IBM i
バージョン 7.2

プログラミング
IBM Rational Development Studio for i
ILE C/C++ プログラマーの手引き

IBM
プログラミング
IBM Rational Development Studio for i
ILE C/C++ プログラマーの手引き

IBM
本書は、IBM Rational Development Studio for i (製品番号 5770-WDS) バージョン 7、リリース 2、モディフィケーション 0、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。このバージョンは、すべての RISC モデルで稼動するとは限りません。また、CISC モデルでは稼動しません。

本書にはライセンス内部コードについての参照が含まれている場合があります。ライセンス内部コードは機械コードであり、IBM 機械コードのご使用条件に基づいて使用権を許諾するものです。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原典： IBM i
Version 7.2
Programming
IBM Rational Development Studio for i
ILE C/C++ Programmer's Guide

発行： 日本アイ・ビー・エム株式会社
担当： トランスレーション・サービス・センター

第1刷 2014.4
© Copyright IBM Corporation 1993, 2013.
目次

ILE C/C++ プログラマーの手引き

「ILE C/C++ プログラマーの手引き」のPDFファイル 1-1

ILE C/C++ プログラマーの手引きについて 2-1
ライセンス交付を受けたプログラムのインストールに関する情報 2-1
サンプルに関する注意 2-1
このガイドで使用する制御言語コマンドおよびプロシージャー 2-2

概要 3-1
ILE C/C++ コンパイラの概要 3-1
複数言語プログラムの作成 3-1
IBM i オペレーティング・システムでサポートされているプログラミング言語 3-1
ILE プログラム作成 3-2
バイナッディング・ディレクトリー 3-2
サービス・プログラム 3-2
プログラムおよび資源管理 3-2
プログラム・フロー 3-3
プログラムおよびプロシージャーの呼び出し 3-3
資源の割り振り 3-3
バイナッディング可能な API 3-3
実行時例外 3-4
プログラムのデバッガ 3-4

プログラムの作成とコンパイル 4-1
プログラムの作成 4-1
プログラム開発プロセス 4-1
プログラムの準備 4-1
プログラムのコンパイル 4-1
モジュールのバイナッディング 4-2
オブジェクトの実行または呼び出し 4-2
プログラムのデバッガ 4-3
ソース・ステートメントの入力 4-3
ソース・ファイルの作成例 4-3
手順 4-3
ソース・コードのサンプル 4-4

1 ステップでのプログラムの作成 4-5
2 ステップでのプログラムの作成 4-7
プログラム入り口プロシージャおよびユーザー入り口プロシージャーの識別 4-7
プログラム・オブジェクトの内部構造の理解 4-8
静的プロシージャー呼び出しの使用 4-8
バイナッディング・ディレクトリーの処理 4-8
バイナッティング・ディレクトリーの作成 4-9
バイナッティングを使用したプログラムの作成 4-9
プログラム作成の準備 4-9
CRTPGM コマンドのパラメータの指定 4-10
インポート要求の解決方法 4-11
バイナッディング・リストの使用 4-11
モジュールまたはプログラム・オブジェクトの更新 4-13
プログラムの更新 4-14
グループの活動化 4-14
メッセージリング・サポート 4-15
サービス・プログラム 4-15
プログラムとサービス・プログラムの違い 4-15
共通インターフェース 4-16
サービス・プログラム作成時の考慮事項 4-16
バイナッターを使用したサービス・プログラムの作成 4-16

CRTSRVPGM コマンドのパラメータの指定 4-17
サービス・プログラムの更新または変更 4-17
サービス・プログラムでの制御言語（CL）コマンドの使用 4-18
サービス・プログラムの作成、コンパイル、およびバイナッティング 4-18
ユーザーヘッダー・ファイル 4-18
ソース・コード・ファイル 4-19
サービス・プログラムのコンパイルおよびバイナッティング 4-20
プログラムへのサービス・プログラムのバイナッティング 4-21

サービス・プログラムからのエクスポートの処理 4-22
サービス・プログラムからのエクスポートの判定 4-22
モジュールの表示コマンドを使用したエクスポートされた定義済みシンポルの表示 4-22
バイナッダー言語ソース・ファイルの作成 4-23
SEU を使用したバイナッダー言語の作成 4-24
RTVBNDSC コマンドを使用したバイナッダー言語の作成 4-24
サービス・プログラム・エクスポート・リストの更新 4-25
デマングル関数の使用 4-25
プログラム作成中の未解決インポート要求の処理 4-27
バイナッターエクスポート・サービス・プログラムの作成 4-27
バイナッターエクスポート・サービス・プログラムの作成 4-27
バイナッター言語を使用したエクスポート・サービス・プログラムの作成 4-28
ソース・ファイルの作成 4-29
ソース・ファイルのモジュールへのコンパイル 4-29
サービス・プログラムを作成するためのバイ
ンダー言語の生成 .......................... 4-30
プログラムへのモジュールのバインド .... 4-30
*UNRSLVREF バラメーターを使用した未解
決インポータ要求の処理 ...................... 4-31
プログラム作成順序の変更による未解決イン
ポート要求の処理 .............................. 4-32
存在しないサービス・プログラムへのプログラ
ムのバインド .................................... 4-33
手順 .......................................... 4-33
コード・サンプル ................................ 4-33
プログラムの実行 ................................. 4-34
サービス・プログラム・エクスポート・リスト
の更新 .............................................. 4-34
プログラムの説明 .................................. 4-35
ソース・ファイルの作成 ........................ 4-36
プログラムおよびサービス・プログラムのコ
ンパイルとバインド ............................. 4-38
プログラムの実行 ................................ 4-38
プログラムの実行 ................................. 4-38
ILE C/C++ 実行時モデル ......................... 4-39
活性化および活性化グループ ................... 4-40
一時変数・ライブラリ関数と活性化グルー
プ .................................................. 4-40
プログラムの呼び出し ............................. 4-41
呼び出し (CALL) コマンドの使用 .............. 4-42
呼び出し先プログラムへのバラメーターの一
変換 .............................................. 4-43
プロセス・コマンド (QCAPCMD) API の
使用 .............................................. 4-44
制御権転送 (TFRCTL) コマンドの使用 .... 4-44
例: TFRCTL コマンドを使用したプログラ
ムの作成および実行 ............................ 4-45
コード・サンプル ................................ 4-45
プログラムを実行する CL コマンドの作成 4-46
プログラムの説明 .................................. 4-46
手順 .......................................... 4-47
コード・サンプル ................................ 4-47
正常および異常のプログラム終了 ............... 4-48
活性化グループの管理 ............................. 4-49
活性化グループの指定 ............................. 4-49
名前付き活動化グループでのプログラムの
実行 .............................................. 4-49
活動化グループ *NEW でのプログラムの
実行 .............................................. 4-50
名前付き活動化グループの更新要件の振
る舞い .............................................. 4-51
活動化グループでのプログラムの実行
(*CALLERK) ...................................... 4-52
呼び出し先プログラムの存在 .............. 4-52
活動化グループの削除 ......................... 4-53
システム・リソースの再利用 ............... 4-53
資源再利用 (RCLRSC) コマンドの使用 4-53
実行時のストレージの管理 ...................... 4-53
デフォルト・ヒープの管理 ...................... 4-54
バインド可能 API を使用したデフォル
ト・ヒープの管理 ......................... 4-54
実行時における動的ストレージ割り振り
置換関数のオーバーライド ...................... 4-55
new または delete 演算子の多重定義 .... 4-56
実行時のバフォーマンスの改善 .................. 4-58
バフォーマンスを改善するためのデータ・タイ
プの選択 .......................................... 4-57
volatile 変数の使用の回避 ...................... 4-57
ビット・フィールドを他のデータ・タイプに
置換 .............................................. 4-57
静的変数およびグローバル変数の使用の最小
化 .............................................. 4-57
レジスター・ストレージ・クラスの使用 .... 4-58
バフォーマンスを改善するためのクラスの作成
バフォーマンス測定の使用可能化 ............. 4-58
コンパイラー・オプションを使用してバフォ
ーマンス測定を使用可能にする ............. 4-58
例外処理の最小化 ................................ 4-59
レコード入出力の戻りコードの有効化 .... 4-59
レコード入出力中に C2M メッセージをオフ
にする ............................................. 4-59
直接モニター・ハンドラーの使用 ............. 4-59
例外のバーコレーショーンの最小化 ............ 4-60
サンプル ILE C ソース・コードの例外バ
ーコレーションの例 ............................. 4-60
関数呼び出しおよび引数の数の削減 ...... 4-61
関数呼び出しのインライン化 ..................... 4-61
静的クラス・メンバ関数またはグローバル
変数の使用 ..................................... 4-62
レジスターでの引数の受け渡し ................ 4-62
プロトタイプ型を使用して関数呼び出し処理を
最小化 .......................................... 4-62
バフォーマンスを改善するための入出力関数の
選択 .............................................. 4-63
レコード入出力関数の使用 ..................... 4-63
ISO C レコード入出力 ............................ 4-63
ILE C レコード入出力 ......................... 4-63
入出力フィードバック情報の使用 ............. 4-64
レコードのブロック化 ......................... 4-64
システム・バッファーの操作 ................... 4-65
入出力の両方でファイルを 1 回開く ........... 4-65
共有ファイルの使用の最小化 .................. 4-66
ファイルのオープンおよびクローズ数の最小
化 .............................................. 4-66
バフォーマンス改善のためのテーブ・ファイル
の定義 .......................................... 4-66
ストリーム入出力関数を使用する際のパフォ
ーマンスの改善 ............................... 4-66
C++ 入出力ストリーム・クラスの使用 .... 4-67
ソース物理ファイルに代わる物理ファイルの
使用 .............................................. 4-67
ライブラリ名の指定 ............................. 4-67
ポインタを使用したバフォーマンス向上 .... 4-67
オブジェクト・ポインタの使用の回避 ....... 4-67
### 目次

#### 5-1 プログラムのデバッグ

<table>
<thead>
<tr>
<th>項目</th>
<th>頁数</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILE ソース・デバッガー</td>
<td>5-1</td>
</tr>
<tr>
<td>デバッグ・データ・オプション</td>
<td>5-1</td>
</tr>
<tr>
<td>デバッグ言語構文</td>
<td>5-2</td>
</tr>
<tr>
<td>デバッグ言語構文の制約</td>
<td>5-2</td>
</tr>
<tr>
<td>デバッグ・コマンド</td>
<td>5-2</td>
</tr>
<tr>
<td>ILE C プログラムでのデバッグ式の使用例</td>
<td>5-4</td>
</tr>
<tr>
<td>プログラム定義および対応するデバッグ式の例</td>
<td>5-4</td>
</tr>
<tr>
<td>エラーを検出および修正するためのポインタの評価</td>
<td>5-4</td>
</tr>
<tr>
<td>エラーを検出および修正するための単純式の評価</td>
<td>5-7</td>
</tr>
</tbody>
</table>

#### 5-2 ソース・デバッガーでのシステム・ポインタおよびスペース・ポインターの表示例

<table>
<thead>
<tr>
<th>項目</th>
<th>頁数</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILE ソース・デバッガーでのシステム・ポインタ</td>
<td>5-10</td>
</tr>
<tr>
<td>デバッグのためのプログラムの準備</td>
<td>5-12</td>
</tr>
<tr>
<td>テスト・ライブラリーのセットアップ</td>
<td>5-12</td>
</tr>
<tr>
<td>デバッグのためのリスト・ビューの作成</td>
<td>5-12</td>
</tr>
<tr>
<td>ソース・デバッグ・セッションの処理</td>
<td>5-13</td>
</tr>
<tr>
<td>ソース・デバッグ・セッションの開始</td>
<td>5-13</td>
</tr>
<tr>
<td>デバッグ・セッションでのプログラムの追加および除去</td>
<td>5-16</td>
</tr>
<tr>
<td>セッション中のデバッグ・オプションの設定または変更</td>
<td>5-17</td>
</tr>
<tr>
<td>ソース・コードのサンプル</td>
<td>5-4</td>
</tr>
<tr>
<td>監査ログ・ファイルのソース・コード</td>
<td>4-83</td>
</tr>
<tr>
<td>ソース・コードによる端末セッション入力の ILE プログラムへの引き渡し</td>
<td>4-83</td>
</tr>
<tr>
<td>セッション・データ収集用 CL コマンドを定義するためのソース・コード</td>
<td>4-83</td>
</tr>
<tr>
<td>ユーザー入力プロシージャー (UEP) のソース・コード</td>
<td>4-84</td>
</tr>
<tr>
<td>税金を計算し、出力用にコストをフォーマットするためのソース・コード</td>
<td>4-86</td>
</tr>
<tr>
<td>監査証跡を書き込むためのソース・コード</td>
<td>4-88</td>
</tr>
<tr>
<td>税率データをエクスポートするためのソース・コード</td>
<td>4-89</td>
</tr>
</tbody>
</table>
| 税率データをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエクスポートするためのデータをエク스
プログラムのステップスルー ............................. 5-30
プログラムのステップオーバー ............................. 5-30
F10 でのプログラムのステップオーバー .......................... 5-30
STEP OVER デバッグ・コマンドの使用 .......................... 5-30
プログラムへのステップイン ........................................ 5-31
F22 を使用したプログラムへのステップイン .................. 5-31
STEP INTO デバッグ・コマンドの使用 ............................ 5-31
呼び出したプログラムのステップイントゥ ...................... 5-31
F22 を使用したプログラムへのステップイン .............................. 5-31
トゥの例 .......................................................... 5-31
OPM プログラムへのステップイン .............................. 5-33
プロシージャーのステップオーバー .............................. 5-33
プロシージャーへのステップイン .................................. 5-34
変数のデバッグ .................................................... 5-36
変数の値の表示 .................................................... 5-36
F11 を使用した変数の表示 ....................................... 5-36
変数の値の変更 .................................................... 5-37
スカラー変数の値の変更 ......................................... 5-38
変数、式、またはデバッグ・コマンドに指定された名前の等価値 .................. 5-39
構造体の表示 ...................................................... 5-40
16 進値としての変数の表示 ..................................... 5-40
ヌル終了文字配列の表示 ........................................ 5-41
文字配列の表示 .................................................... 5-43
ポインタ、変数、およびビット・フィールドの表示 .............. 5-44
システム・ポインターおよびスペース・ポインタを表示するためのサンプル EVAL コマンド .................. 5-44
C++ 構造を表示するためのサンプル EVAL コマンド ........... 5-50
モジュールのソース ............................................... 5-52
モジュール最適化およびプログラム識別情報の変更 ................... 5-54
最適化レベルの変更 ............................................... 5-54
モジュール識別情報の除去 ........................................ 5-56

入出力操作の実行 .................................................. 6-1
IBM i ファイルでの ILE C/C++ ストリームおよびレコード入出力関数の使用 .................. 6-1
ILE C レコード入出力関数 ........................................ 6-1
ストリーム・バッファリング ..................................... 6-2
動的ストリーム・ファイルの作成 ................................ 6-2
動的に作成されたストリーム・ファイルのオープン・モード ........... 6-3
標準入出力テストストリーム・ファイル (<stdio.h>) ............... 6-3
舎棄された標準出力のオーバーライド .......................... 6-3
プログラムによる QILINE を指定した入力ファイルの再読み取りを可能にする ............................. 6-3
IBM i ファイル ..................................................... 6-4
IBM i ファイル・タイプ ............................................ 6-4
ストリーム・ファイルおよび ILE C 入出力操作 .................... 6-4
ファイル位置決めの問題の回避 .................................. 6-4
fopen() 関数の使用 ............................................. 6-5
dopen() メンバー関数の使用 .................................. 6-5
IBM i のファイル命名規則 ....................................... 6-5
テキスト・ストリームおよびバイナリー・ストリームのファイル制御構造 .................. 6-6
テキスト・ストリーム・ファイルの入出力プロセス ................. 6-6
バイナリー・ストリーム・ファイルの入出力プロセス .................. 6-6
オープン・フィードバック域 ....................................... 6-20
入出力フィードバック域 .......................................... 6-20
セッション管理機能の使用 ....................................... 6-20
セッション・ハンドルの入手 ..................................... 6-21
セッション・マネージャー API の使用 .......................... 6-21
例：ILE バインド可能 API を使用した DSM ...................... 6-21
セッションの表示 ............................................... 6-21
オーバーヘッド処理 .............................................. 6-22
コード・サンプル ............................................... 6-22
IBM i 統合ファイル・システムでの ILE C/C++ ストリーム関数の使用 .................. 6-24
統合ファイル・システム (IFS) .................................... 6-24
ルート (/) ファイル・システム .................................. 6-25
ユーザー・アクセス ............................................ 6-25
パス名 ......................................................... 6-25
オープン・システム (QOpenSys) ファイル・システム ............. 6-26
ユーザー・アクセス ............................................ 6-26
パス名 ......................................................... 6-26
ライブラリー (QSYS.LIB) ファイル・システム ................... 6-26
ファイル処理の制約事項 ....................................... 6-26
パス名 ......................................................... 6-26
文書ライブラリー・サービス (QDLS) ファイル・システム ........... 6-27
ユーザー・アクセス ............................................ 6-27
パス名 .......................... 6-27
LAN Server/400（QLANsrv）ファイル・システム .......................... 6-28
システム .......................... 6-28
パス名 .......................... 6-28
光ディスプレイサポート（QOPT）ファイル・システム .......................... 6-28
パス名 .......................... 6-28
ファイル・サーバー（QFileSvr,400）ファイル・システム .......................... 6-29
パス名 .......................... 6-29
統合ファイル・システム・ストリーム入出力の有効化 .......................... 6-30
ラージ・ファイルでのストリーム入出力の使用 .......................... 6-30
ストリーム・ファイルの使用 .......................... 6-30
ストリーム・ファイルとデータベース・ファイルの使用 .......................... 6-31
テキスト・ストリーム .......................... 6-32
バイナリ・ストリーム .......................... 6-32
テキスト・ストリーム・ファイルおよびバイナリ・ストリーム・ファイルのオープン .......................... 6-33
テキスト・ストリームまたはバイナリ・ストリームの入出力の使用 .......................... 6-34
統合ファイル・システム（IFS）の使用 .......................... 6-35
ソース・ファイルの IFS へのコピー .......................... 6-35
ストリーム・ファイルの編集 .......................... 6-35
SRCSTMF バラメーター .......................... 6-35
ヘッダー・ファイルの検索 .......................... 6-35
組込みファイル・リンク .......................... 6-36
インクルード・ディレクトリ検索 .......................... 6-36
インクルード検索パスの規則 .......................... 6-38
ソース・ストリーム・ファイルの指定に関する考慮事項 .......................... 6-41
絶対インクルード・パス名に関する制約事項 .......................... 6-41
ソース・ファイルおよびヘッダー・ファイルに関する推奨事項 .......................... 6-42
プリプロセッサー出力 .......................... 6-42
リスト出力 .......................... 6-42
コード・ページおよび CCSID .......................... 6-43
回避すべき危険性 .......................... 6-43
統合ファイル・システム・ソースの使用例 .......................... 6-44
ストリーム入出力の使用 .......................... 6-44
大きなファイル .......................... 6-45
オープン・モード .......................... 6-45
行終了文字 .......................... 6-45
ファイル・システムおよび装置の処理 .......................... 7-1
プログラムでの外部記述ファイルの使用 .......................... 7-1
外部記述データベース・ファイルの作成 .......................... 7-1
タイプ定義の作成 .......................... 7-2
ヘッダー記述の作成 .......................... 7-2
レコード番号の指定 .......................... 7-3
レコード・フィールド名の指定 .......................... 7-4
型定義へのデータベース・ファイルの組み込み .......................... 7-4
構造体の型（KEY フィールド）の定義 .......................... 7-4
ファイルのロングネームの使用 .......................... 7-7
記述を検証するためのレベル検査 .......................... 7-8
GENCSRC ユーティリティを使用したレベル検査 .......................... 7-9
#pragma mapinc ディレクトリを使用したレベル検査 .......................... 7-11
C/C++構造体におけるフィールド位置合わせの問題の回避 .......................... 7-13
プログラムでの外部フィールド定義の組み込み .......................... 7-13
INPUT オプション .......................... 7-14
OUTPUT オプション .......................... 7-14
BOTH オプション .......................... 7-14
標準の定義および使用 .......................... 7-15
ファイル・バッファーデータの定義 .......................... 7-16
分離標準シンポジウムの作成 .......................... 7-16
プログラムでの物理および論理データベース・ファイルの組み込み .......................... 7-17
プログラムでの装置ファイルの組み込み .......................... 7-18
論理ファイルでの複数の外部記述レコード形式の組み込み .......................... 7-18
プログラムでの外部記述バック 10 進数データの使用 .......................... 7-21
プログラムでのデータベース・ファイルおよび分散ファイルの使用 .......................... 7-22
データベース・ファイルおよび分散ファイル .......................... 7-22
物理ファイルおよび論理ファイル .......................... 7-23
データベース・ファイル内レコードの記述 .......................... 7-23
外部記述ファイルの定義 .......................... 7-23
データ・ファイルとソース・ファイル .......................... 7-24
アクセス・パス .......................... 7-24
キー・フィールドの配列 .......................... 7-25
重複キー値 .......................... 7-25
削除済みレコード .......................... 7-25
ロック .......................... 7-25
共用 .......................... 7-26
ヌル可能フィールド .......................... 7-27
データベース・ファイルおよび DDM ファイルをレコード・フィールドとしてオープンする .......................... 7-27
データベース・ファイルおよび DDM ファイルのレコード関数 .......................... 7-28
DDM ファイルの入出力に関する考慮事項 .......................... 7-29
データベース・ファイルおよび DDM ファイルをバイナリ・ストリーム・ファイルとしてオープンする .......................... 7-29
バイナリ・ストリーム・データベース・ファイルおよび DDM ファイルの入出力に関する考慮事項 .......................... 7-29
データベース・ファイルおよび DDM ファイルのバイナリ・ストリーム関数 .......................... 7-30
データベース・レコード・ファイルをコーディング順に処理する .......................... 7-30
手順 .......................... 7-30
ソース・コードのサンプル .......................... 7-30
IBM i機能を使用した処理 .......................... 8-1
プログラムでの例外の処理 .......................... 8-1
ILE 言語固有のエラー処理 .......................... 8-1
例外メッセージ ................................. 8-2
システムが例外を処理する方法 .......................... 8-3
呼び出しメッセージ待ち行列による ILE プロ
シージャーおよび関数の処理方法 .......................... 8-3
ILE で制御境界が例外処理に与える影響 .......................... 8-3
監視されていない例外および未処理例外 .......................... 8-3
未処理例外を含む ILE C ファイル・コード
の例 ................................. 8-4
ネストされた例外 ................................. 8-4
ストリーム・ファイルおよびレコード・ファイル
のエラーの検出 ................................. 8-5
関数の戻り値の検査 ................................. 8-5
erro 値の検査 ................................. 8-6
Erno の初期化 ................................. 8-6
Erno 値の表示や出力 ................................. 8-6
例: fopen() 関数に対する erro 値のチェック
メジャーマイナーに戻るコードの検査 .......................... 8-7
レコード・ファイルについてのシステム例外
の検査 ................................. 8-7
グローバル変数 _EXCP_MSGID の検査 .......................... 8-8
ILE 例外ハンドラーの使用 .......................... 8-8
例外ハンドラーのタイプ .......................... 8-8
ILE 直接モニター・ハンドラーの使用 .......................... 8-8
プログラム・ディレクトリの使用 .......................... 8-8
通信域変数の使用 ................................. 8-9
直接モニター・ハンドラーのスコープ .......................... 8-9
例外クラスの使用 ................................. 8-10
制御処置の指定 ................................. 8-11
メッセージ ID の指定 ................................. 8-12
直接モニター・ハンドラーを使用するソー
ス・コードの例 ................................. 8-12
直接モニター・ハンドラーの使用方法を説
明するソースの例 ................................. 8-13
直接モニター・ハンドラーを提供するサー
ス・プログラムの例 ................................. 8-16
関数でなくラベルをハンドラーとして使用
する例 ................................. 8-16
ILE 条件ハンドラーの使用 .......................... 8-17
ILE C 条件ハンドラーを使用するタイミ
ング ................................. 8-18
条件ハンドラーを使用する ILE ソースの
例 ................................. 8-18
ILE 条件ハンドラーを提供するサービ
ス・プログラムの例 ............................. 8-18
例外処理の例 ................................. 8-19
ゼロ除算例外の処理例 ............................. 8-20
例外のプロモート例 ............................. 8-22
C/C++ シグナル・ハンドラーの使用 .......................... 8-24
シグナル・ハンドラーの使用タイミング .......................... 8-24
シグナルの発信 ................................. 8-25
シグナル処理の関数プロトタイプ .......................... 8-25
ILE C/C++ ランタイム環境によるシグナルの
処理方法 ................................. 8-26
シグナル・アクションのリセット .......................... 8-26
シグナル・ハンドラーのスタックング .......................... 8-27
例: シグナル・ハンドラーのセットアップ .......................... 8-27
手順 ................................. 8-27
シグナル・ハンドラーをセットアップする
ソース・コードのサンプル .......................... 8-28
C/C++ シグナルと ILE 例外ハンドラーの同時
使用 ................................. 8-30
優先順位 ................................. 8-30
直接モニター・ハンドラーとシグナル・ハン
ドルの併用の例 ................................. 8-31
ネストされた例外の処理 ........................... 8-32
取り消し処理プログラムの使用 .......................... 8-32
例: エラーを検出して例外処理する各種方法
の使用 ................................. 8-34
手順 ................................. 8-35
ソース・コードのサンプル .......................... 8-35
プログラムでのポインタの使用 .......................... 8-37
ILE i ポインタ・タイプ .......................... 8-37
オープン・ポインタの使用 .......................... 8-38
オープン・ポインタ以外のポインタの使用 .......................... 8-39
ポインタ変数の宣言 .......................... 8-39
C および C++ で IBM i ポインタ変数を
宣言する ................................. 8-39
ILE C でパインド済みプロシージャーへの関
数ポインタを宣言する ............................. 8-40
ILE C および ILE C++ で OS リンケージ
付き関数ポインタを宣言する ............................. 8-40
ポインタをキャストする .......................... 8-42
例: IBM i ポインタを動的プログラム呼び
出しの引数として別の ILE C プログラムに
渡す ................................. 8-43
手順 ................................. 8-43
ソース・コードのサンプル .......................... 8-44
ILE C/C++ 呼び出し規則の使用 .......................... 8-45
プログラムおよびプロシージャーの呼び出し
動的プログラム呼び出しの使用 .......................... 8-46
ILE 呼び出しスタックを使用してプログラ
ム・フローを制御する仕組み ............................. 8-46
プログラムおよびプロシージャーの名前変更 .......................... 8-47
ライブラリ修飾のあるプログラムの呼び出し .......................... 8-48
ILE C からの C++ プログラムおよびプロシージャーの呼び出し .......................... 8-50
リンクアプリケーションの指定 .......................... 8-50
例: C++ オブジェクトを使用する ILE C プ
ログラム ................................. 8-51
プログラム構造 ................................. 8-51
プログラム・フロー ................................. 8-53
プログラム出力 ................................. 8-54
ILE C から C++ クラスにアクセスする .......................... 8-54
C 構造体への C++ クラスのマップ .......................... 8-54
例: C++ オブジェクトを使用する ILE C プ
ログラム ................................. 8-55
プログラム・ファイルと構造体 .......................... 8-55
プログラムの説明 .......................... 8-56
プログラム出力 .......................... 8-58
別のプラットフォームから ILE へのプログラムの
移植 ........................................ 8-58
ILE C または C++ へのコード移植の制限 .... 8-58
ファイルの組み込み ................................ 8-59
プラットフォーム固有の拡張機能 .................. 8-59
共用体のメンバー .......................... 8-59
構造体のメンバー .......................... 8-59
10 進数定数 ................................ 8-60
10 進数および case ステートメント .......... 8-60
IFS 内のライブラリ QSYS.LIB .............. 8-60
デラースペースに関する考慮事項 .............. 8-60
ILE C++ オブジェクトの呼び出しの変更 .... 8-60
ヘッダー・ファイルにおける相違点 .......... 8-61
リンクスリンク指定における相違点 .......... 8-61
関数定義における相違点 .................... 8-62
BCD マクロを使用してコード化 10 進数オブジェクトを ILE C++ に移植する .................. 8-62
例 ............................................. 8-62
クラス・テンプレート・インスタンス化の
ILE C 構文へのマッピング ..................... 8-63
乗算および除算の追加精度の処理 ............ 8-63
オブジェクトの指数の決定 .................... 8-64
内部バック 10 進数データ・オブジェクトの
指数の決定 .................................. 8-64
フォーマット設定された C 入出力関数の値
のフォーマット設定 ......................... 8-65
印刷関数フラグ ................................ 8-65
印刷関数のフィールド幅 ...................... 8-65
印刷関数のフィールド精度 .................... 8-65
変換指定子 .................................. 8-66
条件演算子の ILE C または C++ への移植
式間のクラスの相違を解決する明示的キャストの例 ................................. 8-66
一貫性のある変数型の使用例 .................. 8-67
ILE C のバック 10 進数データ・タイプを
DecimalT クラス・テンプレートに移植 .... 8-67
バック構造の使用における相違点 .......... 8-68
エラー検出における相違点 .................... 8-69
無効な 10 進数形式 .......................... 8-69
数学演算子 .................................. 8-69
C と C++ の両方で機能するヘッダー・ファイル ................................. 8-69
二重関数プロトタイプの使用 .................. 8-70
ILE C プログラムで C++ リンクスリンク関数に
アクセスできるようにする .................... 8-70
QSYSINC ヘッダー・ファイルの組み込み .... 8-71
より厳密な C++ 型検査の処理 ................. 8-71
整数データ・タイプのサイズ問題の解決 ..... 8-71
非互換ポインタ型の解決 .................... 8-72
未定義名エラーコの回避するための名前マングリングの無効化 .................. 8-72
C++ 関数プロトタイプとのタイプ不一致の解決 ..... 8-72
関数プロトタイプ不一致の例 .................. 8-73
関数プロトタイプ不一致の処理 ................ 8-73
unsigned char 変数としての unsigned char ポインターの宣言 ................. 8-73
文字配列の初期化 ................................ 8-73
ストリング・リテラルへのアクセス権限の指定 ..... 8-74
単一活動化グループへの有効範囲設定によるキャッシュされない例外の回避 .......... 8-75
複数言語アプリケーションの処理 ................ 8-75
言語間のプロシージャー呼び出し .............. 8-76
任意のプログラム呼び出し (*PGM) 用の ILE
規則 ............................................. 8-77
再帰的呼び出しを非再帰的呼び出しの混合 .... 8-78
ILE プログラムから非 EPM プログラムへの
引数の引き渡し ................................ 8-79
ILE プログラムから EPM プログラムへの引数の引き渡し .................. 8-79
C++ 動的プログラム呼び出しにおけるリネージ指定の使用 .................. 8-80
有効なストリング・リテラル .......................... 8-80
リングスリンク指定 ......................... 8-80
ILE C/C++ からの任意の ILE プログラムの呼び出し .................. 8-80
ILE C++ から別の高水準言語へのパラメターの受け渡し .................. 8-81
異なるリネージ指定の使用 .................... 8-81
異なるリネージ指定の使用 (C++ のみ) ........ 8-83
デフォルト・パラメター受け渡しタイプする使用 ................................. 8-85
操作記述子を使用した不明データ・タイプ
のパラメーターの受け渡し .................. 8-86
例: 操作記述子による関数の呼び出し ..... 8-86
リンクスリンクをオーバーライドせずに関数をオーバーライドする型キャスト .................. 8-87
制御言語プログラムから ILE C++ プログラム
への引数の受け渡し ......................... 8-87
ILE C++ プログラムへの CL 定数の引き渡
し方法 ........................................... 8-87
ILE C++ プログラムへの CL 変数の引き渡
し方法 ........................................... 8-87
CL の例: 複数言語 ILE アプリケーション .... 8-88
プログラムの説明 .......................... 8-88
プログラム構造 ............................ 8-88
プログラムの活用化 .......................... 8-89
アプリケーション・モジュールおよびファイン ................................. 8-90
ILE プログラムの起動 ......................... 8-95
例: ILE C++ プログラムを呼び出すユーザ定義の制御言語プログラム ................. 8-95
プログラムミング・タスク .................... 8-95
ソース・コード ............................. 8-96
CL コマンド SQUARE を使用して、計算
値を戻す .................. 8-97
例: ILE C++ プログラムにパラメーターを渡す制御言語プログラム .................. 8-97
ILE プログラムからの ILE C プロシージャー
へのアクセス .................. 8-99
静的プロシージャー呼び出し .................. 8-99
<table>
<thead>
<tr>
<th>項目</th>
<th>頁碼</th>
</tr>
</thead>
<tbody>
<tr>
<td>テンプレートのインスタンス化の管理</td>
<td>8-160</td>
</tr>
<tr>
<td>テンプレート・インスタンス化管理オプション</td>
<td>8-161</td>
</tr>
<tr>
<td>ILE C++ コンバイラーによるテンプレート・インスタンス化の処理方法</td>
<td>8-162</td>
</tr>
<tr>
<td>静的メンバー定義の生成</td>
<td>8-162</td>
</tr>
<tr>
<td>内部結合</td>
<td>8-162</td>
</tr>
<tr>
<td>外部結合</td>
<td>8-163</td>
</tr>
<tr>
<td>クラス・テンプレート・インスタンス化の例</td>
<td>8-163</td>
</tr>
<tr>
<td>宣言と定義</td>
<td>8-163</td>
</tr>
<tr>
<td>リンケージ</td>
<td>8-164</td>
</tr>
<tr>
<td>デフォルト・テンプレート・インスタンス化管理オプションの使用</td>
<td>8-165</td>
</tr>
<tr>
<td>単一のインスタンス化のためにコードを手動で構築</td>
<td>8-165</td>
</tr>
<tr>
<td>明示的インスタンス化</td>
<td>8-165</td>
</tr>
<tr>
<td>ILE テンプレート・レジストリー・オプションの使用</td>
<td>8-166</td>
</tr>
<tr>
<td>ILE テンプレート登録オプションの機能の仕方</td>
<td>8-166</td>
</tr>
<tr>
<td>TMPLREG バラメーターの値の指定</td>
<td>8-166</td>
</tr>
<tr>
<td>ILE TEMPPINC オプションの使用</td>
<td>8-167</td>
</tr>
<tr>
<td>TEMPPINC オプションの機能の仕方</td>
<td>8-167</td>
</tr>
<tr>
<td>TEMPPINC 管理インスタンス化用のプログラムの構成</td>
<td>8-168</td>
</tr>
<tr>
<td>テンプレート実装ファイル</td>
<td>8-169</td>
</tr>
<tr>
<td>temppinc ファイル</td>
<td>8-170</td>
</tr>
<tr>
<td>ILE C および C++ プログラムでのテラスペースの使用</td>
<td>8-171</td>
</tr>
<tr>
<td>サポートされるテラスペース環境</td>
<td>8-171</td>
</tr>
<tr>
<td>C/C++ のポインタのサポート</td>
<td>8-171</td>
</tr>
<tr>
<td>C/C++ ポインタ型変換</td>
<td>8-171</td>
</tr>
<tr>
<td>テラスペースを使用するためのポインタ可能 API</td>
<td>8-172</td>
</tr>
<tr>
<td>16 バイトの実行時バインディング・ライブラリー</td>
<td>8-173</td>
</tr>
<tr>
<td>8 バイト実行時バインディング (RTBDN) ライブラリー拡張</td>
<td>8-173</td>
</tr>
<tr>
<td>RTBDN を使用した C++ プログラムのパフォーマンス</td>
<td>8-173</td>
</tr>
<tr>
<td>トーマスの最適化</td>
<td>8-174</td>
</tr>
<tr>
<td>要件</td>
<td>8-174</td>
</tr>
<tr>
<td>エラー状態</td>
<td>8-174</td>
</tr>
<tr>
<td>制限</td>
<td>8-174</td>
</tr>
<tr>
<td>各テラスペース・ストレージ・モデルの特性</td>
<td>8-174</td>
</tr>
<tr>
<td>テラスペース環境でコードを移植する際のパフォーマンス</td>
<td>8-174</td>
</tr>
<tr>
<td>ナラリ互換性に関する考慮事項</td>
<td>8-176</td>
</tr>
<tr>
<td>テラスペース環境の指定</td>
<td>8-176</td>
</tr>
<tr>
<td>特定のポインタのサイズの指定</td>
<td>8-176</td>
</tr>
<tr>
<td>一貫した引数宣言の維持</td>
<td>8-177</td>
</tr>
<tr>
<td>ソース・コードのサンプル</td>
<td>8-177</td>
</tr>
<tr>
<td>例: データ・モデルでのフォワード宣言の効果</td>
<td>8-177</td>
</tr>
<tr>
<td>例: new または delete 演算子の再定義</td>
<td>8-177</td>
</tr>
<tr>
<td>例: テンプレートがデータ・モデルを採用する方法</td>
<td>8-179</td>
</tr>
<tr>
<td>例: 関数の多重定義</td>
<td>8-180</td>
</tr>
<tr>
<td>実行時型情報を使用したキャスト</td>
<td>8-180</td>
</tr>
<tr>
<td>RTTI 言語拡張</td>
<td>8-181</td>
</tr>
<tr>
<td>C++ 言語で定義された RTTI の使用</td>
<td>8-181</td>
</tr>
<tr>
<td>dynamic_cast 演算子</td>
<td>8-181</td>
</tr>
<tr>
<td>ポインタ型変換のためのRTTI</td>
<td>8-181</td>
</tr>
<tr>
<td>参照による動的キャスト</td>
<td>8-182</td>
</tr>
<tr>
<td>typeid 演算子</td>
<td>8-183</td>
</tr>
<tr>
<td>typeid 演算の結果</td>
<td>8-183</td>
</tr>
<tr>
<td>式での typeid 演算子の使用</td>
<td>8-183</td>
</tr>
<tr>
<td>type_info クラス</td>
<td>8-184</td>
</tr>
<tr>
<td>コンストラクターおよびデストラクターにおける RTTI の使用</td>
<td>8-185</td>
</tr>
<tr>
<td>ILE に対する C++ の拡張</td>
<td>8-185</td>
</tr>
<tr>
<td>extended_type_info クラス</td>
<td>8-185</td>
</tr>
</tbody>
</table>

国際ロケールおよびコード化文字セットの使用

<table>
<thead>
<tr>
<th>項目</th>
<th>頁碼</th>
</tr>
</thead>
<tbody>
<tr>
<td>プログラムの国際化</td>
<td>9-1</td>
</tr>
<tr>
<td>コード化文字セット ID</td>
<td>9-1</td>
</tr>
<tr>
<td>CCSID へのソース・ファイル変換</td>
<td>9-2</td>
</tr>
<tr>
<td>コード化文字セット ID を使用したソースファイルのフォーマット</td>
<td>9-2</td>
</tr>
<tr>
<td>アイルの作成</td>
<td>9-2</td>
</tr>
<tr>
<td>テラスペース内ストリング・リテラルの変換</td>
<td>9-3</td>
</tr>
<tr>
<td>ワイド文字リテラルのユニコード・サポートの使用</td>
<td>9-4</td>
</tr>
<tr>
<td>ウィンドウ文字リテラルの表現</td>
<td>9-4</td>
</tr>
<tr>
<td>ユニコード文字セット・サポートの有効化</td>
<td>9-5</td>
</tr>
<tr>
<td>#pragma convert() 演算用のユニコードコードの効果</td>
<td>9-5</td>
</tr>
<tr>
<td>GB18030 コードページ・サポート</td>
<td>9-6</td>
</tr>
<tr>
<td>UTF-32 でのワイド文字およびストリング・リテラルの生成</td>
<td>9-6</td>
</tr>
<tr>
<td>考慮事項</td>
<td>9-7</td>
</tr>
<tr>
<td>CCSID のターゲット指定</td>
<td>9-7</td>
</tr>
<tr>
<td>ILE C/C++ コンバイラーによるソース・ファイルのターゲット CCSID への変換方法</td>
<td>9-7</td>
</tr>
<tr>
<td>リテラル、コメント、および ID の制限</td>
<td>9-8</td>
</tr>
<tr>
<td>国際ロケールのサポート</td>
<td>9-9</td>
</tr>
<tr>
<td>言語環境のエレメント</td>
<td>9-9</td>
</tr>
<tr>
<td>ロケール</td>
<td>9-9</td>
</tr>
<tr>
<td>ロケール用の ILE C/C++ サポート</td>
<td>9-9</td>
</tr>
<tr>
<td>*CLD および *LOCALE オブジェクト・タイプ</td>
<td>9-9</td>
</tr>
<tr>
<td>用の ILE C/C++ サポート</td>
<td>9-10</td>
</tr>
<tr>
<td>C ロケール・マイグレーション・テーブル</td>
<td>9-10</td>
</tr>
<tr>
<td>POSIX ロケール定義および *LOCALE サポート</td>
<td>9-12</td>
</tr>
<tr>
<td>LOCALETYPE コンバイラー・オプション</td>
<td>9-13</td>
</tr>
<tr>
<td>ロケールの作成</td>
<td>9-13</td>
</tr>
<tr>
<td>LOCALETYPE(*LOCALE) を使用したモジュールの作成</td>
<td>9-14</td>
</tr>
<tr>
<td>ロケールで使用されるカテゴリー</td>
<td>9-14</td>
</tr>
<tr>
<td>アプリケーションのアクティブ・ロケールの設定</td>
<td>9-14</td>
</tr>
<tr>
<td>項目</td>
<td>頁碼</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>アクティブ・ロケールを設定するための環境</td>
<td>9-15</td>
</tr>
<tr>
<td>変数の使用</td>
<td>9-15</td>
</tr>
<tr>
<td>SAA および POSIX *LOCAL 指定</td>
<td>9-16</td>
</tr>
<tr>
<td>ロケール依存実行時間数</td>
<td>9-16</td>
</tr>
<tr>
<td>商標</td>
<td>A-3</td>
</tr>
<tr>
<td>使用条件</td>
<td>A-3</td>
</tr>
</tbody>
</table>

**GENCSRC ユーティリティと**

<table>
<thead>
<tr>
<th>項目</th>
<th>頁碼</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma mapinc ディレクティブ</td>
<td>10-1</td>
</tr>
<tr>
<td>言語間のデータ・タイプ互換性</td>
<td>11-1</td>
</tr>
<tr>
<td>関連情報</td>
<td>12-1</td>
</tr>
<tr>
<td>特記事項</td>
<td>A-1</td>
</tr>
<tr>
<td>プログラミング・インターフェース情報</td>
<td>A-3</td>
</tr>
</tbody>
</table>
ILE C/C++ プログラマーの手引き

本書は、C および C++ のプログラミング言語を熟知し、ILE C/C++ アプリケーションの作成または保守を行う予定のプログラマーを対象としています。

以下の少なくとも 1 つの使用経験がユーザーに必要です。
• 適用される IBM® i メニューおよび画面
• 制御言語 (CL) コマンド
「ILE C/C++ プログラマーの手引き」の PDF ファイル

この情報の PDF ファイルを表示または印刷できます。

この資料の PDF 版を表示またはダウンロードするには、「ILE C/C++ プログラマーの手引き」を選択します。

PDF ファイルの保存

表示または印刷のために PDF をワークステーションに保存するには、以下のようにします。
1. ご使用のブラウザーで PDF リンクを右クリックする。
2. PDF をローカルに保存するオプションをクリックする。
3. PDF を保存したいディレクトリーに進む。
4. 「保存」をクリックする。

Adobe Reader のダウンロード

これらの PDF を表示または印刷するには、Adobe Reader がご使用のシステムにインストールされている必要があります。Adobe Reader は、Adobe の Web サイト (www.adobe.com/products/acrobat/readstep.html) から無償でダウンロードすることができます。
ILE C/C++ プログラマーの手引きについて

このガイドでは、以下の内容について説明しています。

- ソース・ステートメントの入力
- 2 ステップでのプログラムの作成
- 1 ステップでのプログラムの作成
- プログラムの実行
- プログラムのデバッグ
- ストリームおよびレコード・ファイルの管理
- 以下を行うプログラムの作成
  - 外部記述ファイルの使用
  - データベース・ファイルおよび分散ファイルの使用
  - 装置ファイルの使用
  - 例外の処理
  - プログラムおよびプロシージャーの呼び出し
  - システムにおけるポインターの使用
- プログラムの国際化
- C++ プログラムでのテンプレートの使用
- ILE C++ へのプログラムの移植
- 実行時型情報を使用したキャスト
- テラスペースのサポートの使用
- ロケールを使用したプログラムのカスタマイズ

ライセンス交付を受けたプログラムのインストールに関する情報

ILE C/C++ コンバイラーを使用するシステムに QSYSINC ライブラリーをインストールする必要があります。

サンプルに関する注意

ILE C/C++ コンバイラーの使用方法を示しているサンプルは、簡素化されています。次のことにお注意してください。

- C/C++ 言語構造体の使用については、これらのサンプルですべて説明されているわけではありません。
- 一部の例はコード・フラグメントであり、追加コードがないとコンパイルできません。
- 完全な、実行可能なサンプルはすべて T1520 で始めます。これらは、QCPPLE ライブラリー内のソース・ファイル QACSRC にあります。
- このガイドで紹介するサンプルのほとんどは、CL コマンド行に制御言語 (CL) コマンドを入力することで説明されています。制御言語プログラムを使用すると、ほとんどのサンプルを実行できます。
- サンプルの実行については、QCPPLE/QAINFO のメンバー T1520INF を参照してください。

© Copyright IBM Corp. 1993, 2013
このガイドで使用する制御言語コマンドおよびプロシージャー

このガイドでは、プロシージャーで CL コマンドの入力を指示しています。 CL コマンドを入力するには、コマンド行にコマンドを入力します。Enter キーまたは F4 キーを押すと、オプションおよびパラメーターについてのプロンプトが出されます。

オンライン・ヘルプ情報が必要な場合は、CL コマンド・プロンプトの画面で F1 (ヘルプ) を押します。

Integrated Language Environment® (ILE) CL コマンドを含む CL コマンドのコマンド構文については、「ILE C/C++ コンバイラー参照」を参照してください。 CL コマンドはパッチ・モードまたは対話モードで使用するか、制御言語プログラムから使用することができます。

注: CL コマンドを使用するには、オブジェクト権限が必要です。
概要
このトピックでは、統合化Language Environment® (ILE) および IBM i オペレーティング・システムのプログラミング機能について紹介します。これに、以下の概要が含まれます。

- 混合言語プログラムの作成
- プログラムおよびリソースの管理
- プログラムのデバッグ

### ILE C/C++ コンパイラーの概要
Integrated Language Environment (ILE) は、IBM i オペレーティング・システムとともに、大きなプログラマム開発をさまざまな点からサポートします。C および C++ は、ILE でサポートされる 2 つのプログラミング言語です。表 3-1 に、ILE 言語の完全なセットをリストします。

ILE C/C++ コンパイラーでは、C と C++ の両プログラミング言語でのプログラムの開発がサポートされます。C++は、以下を提供することで、C コンパイラの機能を拡張しています。

- 追加キーワード
- パラメーター化タイプ (テンプレート)
- クラスによるオブジェクト指向プログラミングのサポート
- より厳格なタイプ検査

ILE C/C++ は、プログラム開発の以下の領域で利点をもたらします。

- 検数言語プログラムおよびアプリケーションの作成
- プログラム・フローおよび資源管理
- プログラムのデバッグ

### 複数言語プログラムの作成
任意の ILE プログラミング言語で作成されたモジュールで構成された混合言語プログラムを作成できます。

### IBM i オペレーティング・システムでサポートされているプログラミング言語
コンパイラの ILE ファミリーには、ILE C++, ILE C, ILE RPG, ILE COBOL、および ILE CL が含まれています。表 3-1 は、IBM i オペレーティング・システムがサポートするプログラム言語のリストです。

<table>
<thead>
<tr>
<th>統合化言語環境 (ILE)</th>
<th>オリジナル・プログラム・モデル (OPM)</th>
<th>拡張プログラム・モデル (EPM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>BASIC (PRPQ)</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>CL</td>
<td>FORTRAN</td>
</tr>
<tr>
<td>CL</td>
<td>COBOL</td>
<td>PASCAL (PRPQ)</td>
</tr>
<tr>
<td>COBOL</td>
<td>PL/I (PRPQ)</td>
<td></td>
</tr>
<tr>
<td>RPG</td>
<td>RPG</td>
<td></td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1993, 2013

3-1
ILE プログラム作成

1. ソース・コードのモジュールへのコンパイル
2. 1 つ以上のモジュールの 1 つのプログラム・オブジェクトへのバインド (結合)。

ILE 言語からモジュールをバインドすることができるので、混合言語プログラムを作成および保守することができます。

図 3-1 は、コンパイラおよびバインダーの呼び出しによって ILE プログラムを作成するプロセスを示しています。

図 3-1. ILE におけるプログラム作成

注: 一度プログラムを作成しておくと、プログラムの更新 (UPDPGM) コマンドまたはサービス・プログラムの更新 (UPDSRVPGM) コマンドを使用して、そのプログラムを更新することができます。これらのコマンドは、プログラムの更新時に、新規または変更されたモジュールを使用可能にするだけに、便利です。

バインディング・ディレクトリー

ILE C++ プログラムまたはサービス・プログラムに必要なモジュールおよびサービス・プログラムの名前を入れるためのバインディング・ディレクトリーを作成することができます。各バインディング・ディレクトリーは、バインディング・ディレクトリーの作成 (CRTBNDDIR) コマンドを使用して、特定のライブラリー内に作成されます。バインディング・ディレクトリーを使用すると、プログラム・サイズを縮小することができきます。バインディング・ディレクトリーにリストされたモジュールまたはサービス・プログラムは、必要なときにだけ呼び出されます。

サービス・プログラム

モジュールをサービス・プログラム (+SRVPGM) にバインドすることができます。サービス・プログラムは、呼び出し可能なルーチン（機能またはプロシージャー）を、個別にバインドされたプログラムにパッケージする手段です。サービス・プログラムを使用すると、モジュール性および保守容易性が高まります。ハード・バーティーが開発した既製のモジュールを使用することもできれば、ハード・バーティー用に独自のモジュールをパッケージすることもできます。

プログラムおよび資源管理

ILE は、以下の共通の基盤を提供します。

• プログラム・フローの管理

3-2 IBM i: ILE C/C++ プログラマーの手引き
資源の共用
バインド可能なアプリケーション・プログラミング・インターフェース (API)
プログラムの実行時における例外の処理

プログラム・フロー

プログラムまたはサービス・プログラムを実行可能にする処理は、活動化として知られています。活動化では、その空間で 1 つ以上のプログラムが実行できるように、ジョブ内で資源を割り振ります。プログラムが呼び出されると、ILE は、プログラムに指定された活動化グループを自動的に開始します。プログラムが呼び出されたときに、プログラムの指定された活動化グループが存在していない場合には、プログラムの活動化を保持するために、ジョブ内でそれが作成されます。

注：活動化グループの詳細については、以下を参照してください。

- 4-40 ページの『活動化および活動化グループ』
- 4-49 ページの『活動化グループの管理』

プログラムおよびプロシージャーの呼び出し：ILE では、ILE C++, OPM、および EPM のプログラムが動的プログラム呼び出しを使用して一緒に動作できるプログラムを作成することができます。このような呼び出しを使用する時には、呼び出し側プログラムは呼び出されるプログラムの名前を指定します。この名前は、呼び出し側プログラムが呼び出されたプログラムに制御を渡す直前に実行時のアドレスに解決されます。

動的プログラム呼び出しの使用は、静的プロシージャー呼び出しを使用して最適化することができます。プロシージャー名はバインド時（すなわち、プログラムを作成する時）に解決されるので、静的プロシージャー呼び出しは動的呼び出しよりも速くなります。

さらに、静的プロシージャー呼び出しでは、操作記述子を使用できます。操作記述子は、他の ILE 言語で作成されたバインド可能 API またはプロシージャーを呼び出す場合に使用されます。

注：プロシージャーは、タスクを実行してから呼び出し元に戻る、自己完結型コード・セットです。ILE C++ モジュールは、1 つ以上のプロシージャーから構成されます。

プログラムとプロシージャー間の呼び出しについては、8-45 ページの『ILE C/C++ 呼び出し規則の使用』を参照してください。

資源の割り振り

活動化グループ は、ILE プログラムの資源および動作を管理するキー要素です。コミットメント制御操作の有効範囲を活動化グループ・レベルに設定することができます。ファイル一時変更および共用オープン・データ・ベースの有効範囲を実行中プログラムの活動化グループに設定することができます。終了時のプログラムの動作は、プログラムが実行する活動化グループによって影響を受けます。

注：活動化グループの詳細については、以下を参照してください。

- 4-40 ページの『活動化および活動化グループ』
- 4-49 ページの『活動化グループの管理』

バインド可能な API

ILE は、ILE C/C++ 機能を補足する多くのバインド可能 API を提供しています。バインド可能 API は、プログラム呼び出しおよび活動化機能、条件および記憶域管理、数学関数、および動的画面管理を提供しています。『システム API 解説書』にはバインド可能 API についての情報が含まれています。
実行時例外
C と C++ のランタイム・ライブラリー関数の多くには、エラー検査の目的で戻り値が関連付けられています。例えば、_ftimeov() 関数は、ポリューム間でファイルが移動された場合、1 を戻します。fopen() 関数は、ファイルが正常に開かなかった場合に、NULL を返します。「ILE C/C++ ランタイム・ライブラリー関数」に、ILE C/C++ 関数の戻り値に関する情報があります。

プログラムのデバッグ
ILE では、プログラムがデバッグ情報を含めてコンパイルされていれば、1 つ以上の ILE 言語で作成された任意のプログラムに対してソース・レベルのデバッグを実行できます。

以下のことが可能です。
・プログラムの実行中にデバッグ・コマンドを使用して、プログラムのフローを制御する
・プログラムの実行前に条件付き停止点および無条件停止点を設定する
・指定数のステートメントをステップスルーし、プログラムの呼び出し後に変数を表示または変更する
停止点、ステップ・コマンド、または実行時エラーのためにプログラムが停止したときには、プログラムが停止した時点で、関連するモジュールが表示されます。この時点で追加のデバッグ・コマンドを入力することができます。

ILE C/C++ プログラムのデバッグについては、5-1ページの『プログラムのデバッグ』を参照してください。
プログラムの作成とコンパイル

このトピックでは、以下を行うための方法について説明します。
- コンパイラおよびバインダー・コマンドを使用した ILE C/C++ プログラムの作成
- ILE サービス・プログラムの作成
- サービス・プログラムからエクスポートできるプロシージャーおよびデータ項目の処理
- ILE プログラムの実行
- プログラムのパフォーマンスの向上
- コンパイラおよびバインダー・プログラムを使用したプログラム・モジュールおよび実行可能プログラムの作成

プログラムの作成

このトピックでは以下について説明します。
- プログラム開発プロセス
  - ソース・ステートメントの入力方法
    - 1 ステップでのプログラムの作成方法
    - 2 ステップでのプログラムの作成方法
    - メッセージング・サポート

プログラム開発プロセス

開発プロセスの間、ILE プログラムは、以下の 5 つのステージを経ます。
- 準備
- コンパイル
- バインディング
- 実行
- デバッグ

これらのプロセス・ステップは、必ずしもリストされた順に実行されるとは限りません。プログラムをコンパイルし、コンパイル時エラーを訂正し、プログラムを変更して、再コンパイルするプロセスを数回経てからプログラムをバインドすることができます。

プログラムの準備

プログラムの準備では、ソース・コードの設計、書き込み、および作成が行われます。ソース・コードの作成について詳しくは、4-3 ページの『ソース・ステートメントの入力』を参照してください。

プログラムのコンパイル

ソースに対してコンパイル・コマンドを発行し、発生したコンパイル・エラーを修正します。エラーは、ジョブ・ログまたはリスト（作成を選択した場合）のメッセージで確認できます。

ILE C/C++ コンパイラには、以下のコンパイル・コマンドがあります。
### 4-1. ILE C/C++ コンパイラ・コマンド

<table>
<thead>
<tr>
<th>コンパイラ・コマンド</th>
<th>用途</th>
<th>説明</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRTCMOD</td>
<td>C モジュールの作成</td>
<td>モジュール作成コマンドはモジュール・オブジェクトを作成します。プログラムに複数のソース・ファイルからのオブジェクトが含まれる場合、ソース・ファイルごとにモジュール作成コマンドを使用してから、必須のすべての *MODULE を指定する CRTPGM を実行して、バインド済みプログラムを作成する必要があります。</td>
</tr>
<tr>
<td>CRTCPPMOD</td>
<td>C++ モジュールの作成</td>
<td>モジュール作成コマンドは、1つのステップでモジュール作成とバインドイングの両方の操作を実行し、単一のソース・ファイルから *PGM オブジェクトを作成します。</td>
</tr>
<tr>
<td>CRTBNDC</td>
<td>バインド済み C プログラムの作成</td>
<td>バインド済みプログラムの作成コマンドは、1つのステップでモジュール作成とバインドイングの両方の操作を実行し、単一のソース・ファイルから</td>
</tr>
<tr>
<td>CRTBNDCPP</td>
<td>バインド済 C++ プログラムの作成</td>
<td>バインド済みプログラムの作成コマンドは、1つのステップでモジュール作成とバインドイングの両方の操作を実行し、単一のソース・ファイルから</td>
</tr>
</tbody>
</table>

注: コンパイラ・コマンドは、制御言語プログラムまたは Make サイドから発行される場合もあります。

モジュール作成コマンドおよびバインド済みプログラム作成コマンドとそのオプションについて詳しくは、「ILE C/C++ コンパイラ・参考」を参照してください。

### 4-2. IBM i: ILE C/C++ コンパイラ・手引き

モジュールのバインドイング
コンパイル時にモジュールを作成した場合、プログラムの作成 (CRTPGM) またはサービス・プログラムの作成 (CRTSRVPGM) コマンドを使用して、それらのモジュール・オブジェクトを一緒にバインドする必要があります。結果は実行可能な *PGM または *SRVPGM オブジェクトになります。

バインドイングにより、1つ以上のモジュールが1つのプログラム (*PGM) またはサービス・プログラム (*SRVPGM) に結合されます。ILE の C または C++ で書かれたモジュールは、他の ILE 言語で書かれたモジュールにバインドすることができます。C++ で書かれたモジュールは、C++ クラス・ライブラリー、C クラス・ライブラリー、および ILE サービス・プログラムからの関数を使用できます。バインダーは、各モジュール内のアドレスを解決し、一緒にバインドされるモジュール間で要求のインポートとオファーのエクスポートを行います。

いったんプログラムが作成されると、その後でプログラムの更新 (UPDPGM) コマンドまたはサービス・プログラムの更新 (UPDSRVPGM) コマンドを使用して更新できます。これらのコマンドは、プログラムの更新時に、新規または変更されたモジュールを使用可能にするだけでなく、便利です。

### オブジェクトの実行または呼び出し
*CMD オブジェクトは実行されますが、*PGM オブジェクトは呼び出されます。例えば、HELLO *CMD を実行するには、コマンド行に HELLO と入力して、実行キーを押します。HELLO *PGM を実行するには、QCMD 行に CALL HELLO と入力して、実行キーを押します。
プログラムのデバッグ
デバッグを実行することにより、プログラム内の実行時エラーを検出したり、診断したり、除去したりすることができます。ILE ソース・デバッガーを使用すると、ILE または OPM プログラムのデバッグを行えます。

ILE デバッグに関する考慮事項については、「ILE 概念(SD88-5033-09)」を参照してください。

ソース・ステートメントの入力
編集セッションを開始してソース・ステートメントを入力する前に、ライブラリーとソース物理ファイルを作成しておく必要があります。また、統合ファイル・システム (IFS) ファイルからソース・ステートメントをコンパイルすることもできます。詳細については、6-34 ページの「統合ファイル・システム (IFS) の使用」を参照してください。

プログラム開発管理機能の開始 (STRPDM) コマンドを使用して、編集セッションを開始し、ソース・ステートメントを入力できます。

プログラム開発管理機能 (PDM) の他に、以下に示すように、ソースを入力するためのいくつかの方法があります。
- ファイル・コピー (CPYF) コマンド。
- SEU 開始 (STRSEU) コマンド。
- プログラマーメニュー。

これは、すべてを網羅したリストではありません。ソースを作成して IBM i プラットフォームに配置する方法は他にもあります (NFS や ftp など)。

ソース・ファイルの作成例
この例は、ライブラリー、ソース物理ファイル、メンバを作成し、編集セッションを開始して、ソース・ステートメントを入力し、メンバーを保存する方法を示しています。

手順
1. MYLIB というライブラリを作成するには、次のように入力します。
   `CRTLIB LIB(MYLIB)`
2. QCSRC というソース物理ファイルをライブラリ MYLIB 内に作成するには、次のように入力します。
   `CRTSRCPF FILE(MYLIB/QCSRC) TEXT('Source physical file for all ILE C programs')`

QCSRC は、モジュールおよびプログラムの作成に使用する ILE C コマンドのディフェルトのソース・ファイル名です。ILE C++ コマンドの場合、これに対する省略時値は QCPPSRCです。このファイルを統合ファイル・システムのファイルにコピーする方法については、6-34 ページの「統合ファイル・システム (IFS) の使用」を参照してください。
3. 編集セッションを開始するには、次のように入力します。
   `STRPDM`
4. オプション 3 (メンバの処理) を選択し、ソース・ファイル名 QCSRC とライブラリ MYLIB を指定します。
5. F6 (作成) を押してメンバー名 T1520ALP およびソース仕様タイプ C を入力します。「SEU 編集」画面が表示され、ここでソース・ステートメントを入力することができます。
6. 「ソース・コードのサンプル」で表示されたソースを「SEU 編集」画面に入力します。注：3 文字表記の使用について詳しくは、「ILE C/C++ 解説書」を参照してください。

ソース・コードのサンプル

ILE C コンパイラは、CCSID 290、905、および 1026 以外のあらゆる 1 バイト EBCDIC CCSID (コード化文字セット ID) で作成されたソース・コードを認識します。CCSID については、「プログラムの国際化」を参照してください。

C および C++ の文字セットの文字には、環境によっては使用可能でないものがあります。3 文字表記と呼ばれる 3 文字のシーケンスを使用して、これらの文字を C または C++ ソース・プログラムに入力できます。

注：3 文字表記の使用について詳しくは、「ILE C/C++ 解説書」を参照してください。

> C コンパイラは 2 文字表記もサポートしています。(C++ コンパイラは 2 文字表記をサポートしません。)
/* This program reads input from the terminal, displays characters, */
/* and sums and prints the digits. Enter a "+" character to */
/* indicate EOF (end-of-file). */
#define MAXLEN 60 /* The maximum input line size. */
#include <stdio.h>
#include <ctype.h>

#include <trigraphs.h>

void main(void)
{
  int c;
  int i = 0, j = 0;
  int sum = 0;
  int count, cnt;
  int num[MAXLEN]; /* An array of digits. */
  char letter[MAXLEN]; /* An array of characters. Trigraphs */
  /* replace the square brackets. */

  while ((c = getchar()) != '+')
  {
    if (isalpha(c)) /* A test for an alphabetic */
    { /* character. */
      letter[i++] = c;
    }
    else if (isdigit(c)) /* A test for a decimal digit. */
    { /*
      num[j++] = c - '0'; */
      num[j++] = c - '0'; /* Trigraphs replace the square */
      /* brackets. */
    }
  }
  printf("Characters are ");
  for (count = 0; count < i; ++count)
  {
    printf("%c", letter[count]);
  }
  printf("\nSum of Digits is ");
  for (cnt = 0; cnt < j; ++cnt)
  { /*
    sum += num[cnt]; */
    sum += num[cnt];
  }
  printf("\%d\n", sum);
}

1 ステップでのプログラムの作成
バインド済みプログラムの作成 (CRTBNDC および CRTBNDCPP) コマンドを使用して、プログラム
(*PGM オブジェクト) をワンステップで作成できます。

これらのバインド済みプログラムの作成コマンドでは、コンパイルとバインドのステップが結合されます。
これらを使用するには、モジュールの作成コマンド・ステップで作成されるモジュールが CRTPGM ステ
ップ後に削除されるという点を除いて、まずはモジュールの作成 (CRTCONE または CRTPMOD) コマン
ドを呼び出してから、プログラムの作成 (CRTPGM) コマンドを呼び出すのと同じです。

バインド済みプログラムの作成コマンドを使用するには、ソース・メンバーに main() 関数が含まれている
必要があります。
注: バインド済みプログラムの作成コマンドで CRTPGM パラメーターが指定されていない場合、CRTPGM パラメーターのデフォルトが使用されます。例えば、パラメーター ACTGRP(*NEW) が CRTPGM コマンドのデフォルトであり、バインド済みプログラムの作成コマンドで使用されます。コマンド省略時間変更 (CHGCMDDFT) コマンドを使用して、CRTPGM パラメーターのデフォルトを変更できます。

CRTSQLCI または CRTSQLCPPI コマンドを使用して、ILE C コンパイラーを開始し、プログラム・オブジェクトを作成できます。ILE C/C++ ソースで SQL ステートメントを組み込んだ場合、SQL データベースに ILE C/C++ プログラムからアクセスできます。

例:

1. [4-5ページの図 4-1] に示したソースを使用して、プログラム T1520ALP を作成するには、以下を入力します。

   CRTBNDC PGM(MYLIB/T1520ALP) SRCFILE(QCPPLE/QACSRC)
   TEXT('Adds integers and prints characters') OUTPUT(*PRINT)
   OPTION(*SHOWINC *NOLOGMSG) FLAG(30) MSGLMT(10)
   CHECKOUT(*PARM) DBGVIEW(*ALL)

   指定されているオプションは以下のとおりです。
   • OUTPUT(*PRINT) - コンパイラー・リストが必要であることを指定します。
   • OPTION(*SHOWINC *NOLOGMSG) - コンパイラー・リストで組み込みファイルを展開し、ジョブ・ログにログメッセージを記録しないことを指定します。
   • FLAG(30) - 重大度レベル 30 のメッセージがリストで表示されるように指定します。
   • MSGLMT(10) - 重大度レベル 30 のメッセージが11件出た後にコンパイルを停止するように指定します。
   • CHECKOUT(*PARM) - 使用されていない関数・プロパティ・パラメーターのリストを示します。
   • DBGVIEW(*ALL) は、3つすべてのビューおよびデバッグ・データでこのプログラムをデバッグするように指定します。

2. コンパイラー・リストを表示するには、以下の CL コマンドのいずれかを入力します。
   • DSPJOBで、オプション 4 (スプール・ファイル表示) を選択します。
   • WRKJOB で、オプション 4 (スプール・ファイル処理) を選択します。
   • WRKOUTQ queue-name
   • WRKSQLPFL

   コンパイラー・リストを表示するためのオプションを選択します。

3. プログラムを実行するには、以下を入力します。

   CALL PGM(MYLIB/T1520ALP)

4. a と入力し、Enter を押します。9 と入力し、Enter を押します。b と入力し、Enter を押します。8 と入力し、Enter を押します。

   対話式セッションは、以下に示すようになります。
2 ステップでのプログラムの作成

ILE C/C++ が提供する柔軟性を活用するために、2 ステップでソース・コードを ILE C/C++ プログラムにコンパイルしてバイインドできます。

1. 最初のステップでは、モジュールの作成コマンドを使用して、各ソース・メンバーから 1 つ以上の ILE C/C++ モジュール・オブジェクト (*MODULE) を作成します。

2. 2 番目のステップでは、プログラムの作成 (CRTPGM) コマンドを使用して、これらの 1 つ以上のモジュール・オブジェクトを、実行可能な 1 つの ILE プログラム・オブジェクト (*PGM) にバイインドします。バイインディングは、1 つ以上のモジュールおよびオプション・サービス・プログラムを結合し、それらの間で外部シンボルを解決するプロセスです。モジュールを結合してシンボルを解決するシステム・コードは、バイナーと呼ばれます。

例えば、

CRTMOD HELLO
CRTPGM HELLO
CALL HELLO

モジュールを使用する利点は、以下のとおりです。

- モジュールは保守が容易です。プログラム全体を保守するより、単一の機能を表す小さいモジュールを保守した方が容易です。例えば、モジュール内の 1、2 行だけを変更する場合、プログラム全体ではなく、そのモジュールだけを再コンパイルすれば済みます。
- モジュールはテストが容易です。それぞれの機能のテストを独立して行うことができます。プログラム全体を実行する必要はありません。代わりに、テスト中のモジュールを含むテスト・ハーネスを使用することができます。
- モジュールはコーディングが容易です。単一のソース・ファイルでプログラム全体をコーディングするのではなく、作業を小さなソース・メンバーに分割できます。
- モジュールはさまざまなアプリケーション・プログラムで再利用することができます。

プログラム入りロプロシージャーおよびユーザー入りロプロシージャーの識別

モジュール・オブジェクトの作成時に、プログラム入りロプロシージャー (PEP) およびユーザー入りロプロシージャー (UEP) が生成されることがあります。

ILE C と C++ でともに、main() 関数が必要ですが、ILE C では、ILE プログラムの UEP になります。PEP の実行後に、関連付けられた UEP が呼び出され、ILE プログラムの実行が開始されます。

このバイナードプロセスの一部として、プロシージャーを始動プロシージャー (プログラム入りロプロシージャー (PEP)) として識別する必要があります。プログラムの呼び出し時に、PEP はコマンド行パラメータを受け取り、プログラムの初期制御が与えられます。PEP から制御を受け取るプロシージャーは、ユーザーアリロプロシージャー (UEP) と呼ばれます。
ILE モジュールには、main() 関数が含まれている場合にのみ、プログラム入り口プロシージャーが含まれます。そのため、プログラムにバインドするモジュールのいずれかに、main() 関数が含まれている必要があります。

プログラム・オブジェクトの内部構造の理解

図4-2 に、2 つのモジュール (TRNSRPT および INCALC) をバインドして作成された標準プログラム・オブジェクト MYPROG の内部構造を示します。この例では、TRNSRPT は、UEP に加え、PEP が含まれた入り口モジュールです。モジュール INCALC には、UEP のみが含まれています。

静的プロシージャー呼び出しの使用

バインドされたオブジェクト内では、静的プロシージャー呼び出しを使用してプロシージャーを呼び出すことができます。これらのバインド呼び出しは外部呼び出しより高速です。したがって、多くのバインド呼び出しのある単一のバインド済みプログラムから成るアプリケーションは、多くの外部プログラム間呼び出しがある各プログラムから構成される同様のアプリケーションより高速で実行されるはずです。

パインディング・ディレクトリーの処理

パインディング・ディレクトリーには、ILE プログラムやサービス・プログラムの作成に必要になる可能性のあるモジュールおよびサービス・プログラムの名前が入っています。

パインディング・ディレクトリーは以下のメリットを提供します。

• ILE プログラムやサービス・プログラムの作成に必要になる可能性のあるモジュールまたはサービス・プログラムをパッケージ化する便利な方法。

• パインディング・ディレクトリーにリストされたモジュールまたはサービス・プログラムが必要な場合にだけ使用されることによる、プログラム・サイズの縮小。
バイニング・ディレクトリーはオプションです。それらは、CRTPGM コマンドの BNDDIR パラメータによってシステムに識別されたオブジェクトです。

バイニング・ディレクトリーにリストされているモジュールまたはサービス・プログラムは、現在未解決であるインポート要求を満足するエクスポートを提供する場合にのみ使用されます。バイニング・ディレクトリーの項目は、バイニング・ディレクトリーの作成時点ではまだ存在しないが、その後存在することになるオブジェクトを参照している可能性があります。

バイニング・ディレクトリーの作成
バイニング・ディレクトリーを作成する場合、バイナード・ディレクトリーの作成 (CRTBNDDIR) コマンドを使用して、ILE C/C++ プログラムまたはサービス・プログラムに必要となるモジュールおよびサービス・プログラムの名前を入れることができます。

例えば、次のような場合です。
CRTBNDDIR BNDDIR(MYBNDDIR) MODULES (MOD1, MOD2)
CRTCPMOD PROG (MYPROG) BNDDIR (MYBNDDIR)
または
CRTBNDDIR BNDDIR(MYBNDDIR) MODULES (MOD1, MOD2)
CRTCPMOD PROG (MYPROG) BNDDIR (MYBNDDIR)

バイナーを使用したプログラムの作成
バイナーは、プログラムの作成 (CRTPGM) コマンドまたはサービス・プログラムの作成 (CRTSRVPGM) コマンドによって呼び出されます。CRTPGM コマンドは、1 つ以上のモジュール・オブジェクトから 1つのプログラム・オブジェクトを作成し、また必要に応じて、1つ以上のサービス・プログラムにバインドします。CRTSRVPGM コマンドは、1つ以上のモジュール・オブジェクトから 1つのサービス・プログラム・オブジェクトを作成し、また必要に応じて、1つのサービス・プログラムにバインドします。サービス・プログラムについて詳しくは、「4-15 ページの『サービス・プログラム』」を参照してください。

CRTPGM および CRTSRVPGM コマンドは、バイナーと呼ばれる IBM i コンポーネントを呼び出します。バイナード・プログラムは、指定されたモジュールからのプロシージャー名およびデータ項目名に関するインポート要求を処理します。次にバイナード・プログラムは、指定されたモジュール・サービス・プログラム、およびバイニング・ディレクトリーを調べて一致するエクスポートを探します。エクスポートは、他のモジュールまたはプログラムが使用できる、モジュールまたはプログラムで定義された外部シンボルです。インポートは、現行モジュール・オブジェクトで定義されていないプロシージャーまたはデータ項目の名前の使用または参照です。

コンバイラーで作成されたモジュールを、他のいずれかの ILE モジュール作成コマンド (CRTRPGMOD, CRTCMOD, CRTCBLMOD, CRTCLMOD など) または他の ILE コンバイラーで作成されたモジュールにバインドできます。

注: バインドするモジュールまたはサービス・プログラムは既に作成されている必要があります。

プログラム作成の準備
CRTPGM コマンドを使ってプログラム・オブジェクトを作成する前に、以下のことを実行しなければなりません。
1. プログラム名を決める。
2. プログラム・オブジェクトにバインドしたいモジュール、および必要であればサービス・プログラムを特定する。
3. 動的プログラム呼び出しの際に制御を行うプログラム呼び込みプロシージャーがプログラムに含まれていることを確認する。（つまり、1つのモジュールにプログラムのmain()関数が含まれている必要がある。）

どのモジュールにプログラム呼び込みプロシージャーが入っているかを、ENTMODパラメーターによって指示します。デフォルト値はENTMOD(*FIRST)であり、MODULEパラメーターのリストで見つかった最初のプログラム呼び込みプロシージャーを含むモジュールが呼び込みモジュールであることを意味しています。

複数のILEモジュールを一緒にバインドする場合、ENTMOD(*FIRST)を指定するか、プログラム呼び込みプロシージャーを含むモジュール名を指定します。プログラム・オブジェクトにバインドするモジュールが1つのみである場合、あるいは複数のモジュールをバインドするけれどもプログラム呼び込みプロシージャーを含むモジュールが1つのみである場合は、ENTMOD(*ONLY)を使用できます。例えば、main()関数を持つモジュールをmain()関数を持たないCモジュールにバインドする場合は、ENTMOD(*ONLY)を指定することができます。
4. プログラムを使用する活動化グループを識別する。

ユーザーのプログラムに特殊な要件がないか、あるいは使用するグループが確定していない場合には、ACTGRP(*NEW)を指定してください。

ACTGRP(*NEW)がCRTPGMのデフォルトの活動化グループであることにご注意ください。これはプログラムがそれ自体の活動化グループで実行され、プログラムが終了すると活動化グループも終了することを意味します。この省略値により、プログラムが呼び出される度に、実行に必要な資源のリフレッシュが確実に行われます。

名前のない、および名前付きの活動化グループについて詳しくは、4-14ページの「グループの活動化」を参照してください。

CRTPGMコマンドのパラメーターの指定

表4-2に、CRTPGMコマンドのパラメーターとそのデフォルト値を示します。各パラメーターには、値を指定しない場合に使用されるデフォルト値があります。

注: パラメーターの詳しい説明については、IBM i Information Center のWebサイト(http://www.ibm.com/systems/i/inforcenter)で、「プログラミング」カテゴリーの「CLおよびAPI」のセクションを参照してください。

表4-2. CRTPGMコマンドのパラメーターとそのデフォルト値

<table>
<thead>
<tr>
<th>パラメーター・グループ</th>
<th>パラメーター (デフォルト値)</th>
</tr>
</thead>
<tbody>
<tr>
<td>識別</td>
<td>PGM(ライブラリ名/プログラム名) MODULE(*PGM) TEXT(*ENTMODTXT)</td>
</tr>
<tr>
<td>プログラム・アクセス</td>
<td>ENTMOD(*FIRST)</td>
</tr>
<tr>
<td>バインド</td>
<td>BNDSRVPGM(*NONE)</td>
</tr>
<tr>
<td></td>
<td>BNDDIR(*NONE)</td>
</tr>
<tr>
<td>実行時</td>
<td>ACTGRP(*NEW)</td>
</tr>
<tr>
<td>パラメーター・グループ</td>
<td>パラメーター (デフォルト値)</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>その他</td>
<td>OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF)</td>
</tr>
<tr>
<td></td>
<td>DETAIL(*NONE)</td>
</tr>
<tr>
<td></td>
<td>ALWUPD(*YES)</td>
</tr>
<tr>
<td></td>
<td>ALWLIBUPD(*NO)</td>
</tr>
<tr>
<td></td>
<td>USRPRF(*USER)</td>
</tr>
<tr>
<td></td>
<td>REPLACE(*YES)</td>
</tr>
<tr>
<td></td>
<td>AUT(*LIBCRTAUT)</td>
</tr>
<tr>
<td></td>
<td>TGTRLS(*CURRENT)</td>
</tr>
<tr>
<td></td>
<td>ALWRINZ(*NO)</td>
</tr>
<tr>
<td></td>
<td>STGMDL(*SNGLVL)</td>
</tr>
<tr>
<td></td>
<td>IPA(*NO)</td>
</tr>
<tr>
<td></td>
<td>IPACTLFILE(*NONE)</td>
</tr>
</tbody>
</table>

インポート要求の解決方法
異なるソースのモジュールを 1 つのプログラムに結合する際には常に、コンバイラーは重複した記号の処理を行う必要がある場合があります。

CRTPGM コマンドを入力すると常に、ILE コンバイラーは以下によってインポート要求を解決します。
1. リストされたモジュールをプログラム・オブジェクトになるものにコピーし、サービス・プログラムをこのプログラム・オブジェクトにリンクする。
2. プログラム入力プロシージャーが入っているモジュールを特定し、このモジュールの最初のインポートを見つける。
3. リストされている順序にモジュールをチェックし、最初のインポートをモジュール・エクスポートと突き合わせる。
4. 最初のモジュールに戻り、次のインポートを見つける。
5. 最初のモジュール内のすべてのインポートを解決する。
6. 次のモジュールに進み、すべてのインポートの解決が終わるまで、後続の各モジュール内のすべてのインポートを解決する。

すべてのインポートが解決されると、ILE コンバイラーはバイナド処理を完了し、プログラム・オブジェクトを成形します。いずれかのインポートがエクスポートで解決できない場合、コンバイラーはプログラム・オブジェクトを作成せずにバイナド処理を終了します。

注: 変数をエクスポートするよう (EXPORT キーワードを使用して) バインダー言語で指定した場合には、この変数の名前を、バイナド済みプログラム・オブジェクト内の別のプロシージャーの変数と同じにすることが可能です。 CRTPGM OPTION パラメーターの *DUPPROC オプションを使用して、重複プロシージャー名を許可することができます。この状況の処理方法について詳しくは、「ILE C/C++ コンパイラー参照」を参照してください。

バインダー・リストの使用
バインディング処理では、使用されたリソース、検出した記号およびオブジェクト、およびバインディング処理で解決した問題または未解決の問題を説明するバインダー・リストをオプションで成形することができします。
このリストは CRTPGM コマンドを入力するのに使用するジョブのスプール・ファイルとして作成されます。DETAIL パラメーター値を選択することで、次の 3 つの詳細レベルでリストを生成することができます。

- *BASIC
- *EXTENDED
- *FULL

デフォルトでは、リストの生成は行われません。リストが生成される場合、パインダー・リストには、DETAIL に指定された値に応じて、表 4-3 に記載されたセクションが含まれます。

表 4-3. DETAIL パラメーターに基づくパインダー・リストのセクション

<table>
<thead>
<tr>
<th>セクション名</th>
<th>*BASIC</th>
<th>*EXTENDED</th>
<th>*FULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>コマンド・オプションの要約</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>簡略要約表</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>拡張要約表</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>BIND プログラムの情報リスト</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>相互参照リスト</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>パインド統計</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

このリストの情報は、パイングが正常に行われなかった場合に問題を診断するのを容易にしたり、あるいはパインダーのパインティング処理で検出されたものについてのフィードバックを与えることができます。

4-12 ページの図 4-3 には、プログラム CVTHEXPGM のための基本的なパインダー・リストを記載しています。このリストはコンテキストから取得されたものであることに注意してください。これはパインダー・リストに表示される可能性のある情報のタイプを示しているにすぎません。
モジュールまたはプログラム・オブジェクトの更新
モジュールまたはプログラム・オブジェクトを変更する必要が生じる理由は多数あります。

・ 機能拡張への対応や保守の目的で、オブジェクトの変更が必要となることがあります。

図4.3 基本的なバインダー・リストの例
変更必要箇所は、デバッガー情報または CRTPGM コマンドからのバインダー・リストを使用して、判断することができます。この情報から、どのモジュール、プロシージャー、またはフィールドに変更が必要かを判定することができます。

・モジュールまたはプログラムの最適化レベルあるいはプログラム識別情報を変更する必要が生じる場合もあります。

この状況は、プログラムまたはモジュールをデバッガーした時、あるいはプログラムを実動用にする準備ができている時によく起こります。そのような変更は、該当するオブジェクトを再作成するよりも迅速に行うことができ、システム・リソースの使用量も少なく済みます。

・アプリケーションを完成させた後で、プログラム・サイズを小さくすることもできます。

ILE プログラム・オブジェクトには、そのオブジェクトに追加される付加データがあるため、同様の OPM または EPM プログラム・オブジェクトよりも大きくになります。

上記のアプローチには、変更のためにそれぞれ異なるデータが必要です。

プログラムの更新

一般に、プログラムは必要に応じてモジュールを置き換えることによって更新することができます。プログラム・オブジェクトを再作成する必要はありません。例えば、該当プログラムを既に使用している他のサイトにアプリケーションを提供する場合に、特定モジュールを置き換えられると便利です。ユーザーは変更モジュールを送信するだけでよく、受け取ったサイトでは、UPDPGM および UPDSRVPGM コマンドを使用してアプリケーションを更新することができます。

更新コマンドは、プログラム・オブジェクトとモジュール・オブジェクトの両方を処理します。これらのコマンドのパラメーターは、プログラムの作成 (CRTPGM) コマンドのものによく似ています。例えば、プログラムのモジュールを置き換えるには、MODULE パラメーターにモジュール名を入力し、ライブラリ名を入力します。

UPDPGM コマンドを使用する場合、置き換えられるモジュールは、プログラムの作成時と同じライブラリーに存在する必要があります。すべてのモジュールの置き換え、またはモジュールのサブセットのみの置き換えを指定できます。

グループの活動化

活動化とは、ILE プログラム実行の準備を行うプロセスです。活動化により、ILE プログラムの静的記憶域の割り振りと初期化が行われ、ILE プログラムと ILE サービス・プログラムとのバインドが完了します。CRTPGM および CRTSRVPGM コマンドの ACTGRP パラメーターは、プログラムまたはサービス・プログラムを実行する活動化グループを指定します。

すべての ILE プログラムおよびサービス・プログラムは、活動化グループと呼ばれるジョブのサブストラクチャー内内で活動化されます。このサブストラクチャーは、その ILE プログラムの実行に必要なリソースが含まれます。静的プログラム変数と自動プログラム変数、および動的記憶域に、活動化グループごとの個別のアドレス・スペースが割り当てられます。活動化および活動化グループは、以下を実行します。

・活動化グループに対して資源をスコープ宣言することで、同じジョブ内で実行中の ILE プログラム (例えば、コミットメント制御、指定変更、共有ファイルなど) が互いに干渉されることなく独立して実行できるようにする。

ILE プログラムに対して資源をスコープ宣言する。

ILE プログラムまたはサービス・プログラムが必要とする静的データを固定的に割り振る。

ILE サービス・プログラムに対するシンボリック・リンクを物理アドレスに変更する。

4-14  IBM i: ILE C/C++ プログラマーの手引き
メッセージング・サポート
以下のようにソース・コードのコンパイル時に受け取る可能性のあるコンパイラ・メッセージのレベルについて説明します。

<table>
<thead>
<tr>
<th>重大度 (Severity)</th>
<th>コンパイラ応答</th>
</tr>
</thead>
<tbody>
<tr>
<td>通知 (00)</td>
<td>コンパイルは継続します。メッセージは、コンパイル中に検出された状態を報告します。</td>
</tr>
<tr>
<td>警告 (10)</td>
<td>コンパイルは継続します。メッセージは、有効ではあるが、意図したものではない可能性がある状態を報告します。</td>
</tr>
<tr>
<td>エラー (20)</td>
<td>コンパイルは継続し、オブジェクト・コードが生成されます。コンパイラが修正できるエラー状態が存在しますが、プログラムが正しく実行されない可能性があります。</td>
</tr>
<tr>
<td>重大エラー (30)</td>
<td>コンパイルは継続しますが、オブジェクト・コードは生成されません。コンパイラが修正できないエラー状態が存在します。</td>
</tr>
<tr>
<td>リカバリ不可能エラー (40)</td>
<td>コンパイラは停止します。内部コンパイラ・エラーが検出され、メッセージは、IBM サービス技術員に報告する必要があります。</td>
</tr>
</tbody>
</table>

サービス・プログラム
サービス・プログラムは、タイプ *SRVPGM の IBM i オブジェクトです。サービス・プログラムは通常、1つのアプリケーション内および複数のアプリケーション間で頻繁に他のプロシージャーに呼び出される共通の関数に使用されます。例えば、ILE コンパイラは数学関数や入出力関数などの実行時サービスを提供するためにサービス・プログラムを使用します。

サービス・プログラムは、その単一のコピーのみが保守されて保管されるため、保守を単純化し、ストレージ要件を低減します。

このトピックでは以下について説明します。
- **プログラムとサービス・プログラムの違い**
- **パブリック・インターフェース**
- **サービス・プログラムを作成する際の考慮事項**
- **バインダーを使用したサービス・プログラムの作成方法**
- **サービス・プログラムの作成方法**

プログラムとサービス・プログラムの違い
サービス・プログラムは、以下の 2つの点でプログラムとは異なっています。
- サービス・プログラムは既存のプログラムまたは他のサービス・プログラムにバインドされます。単独では実行できません。
- サービス・プログラムはプログラム入力プロシージャーを含みません。そのため、OS リンケージ指定を使用してサービス・プログラムを呼び出すことはできません。ただし、c リンケージ指定を使用してサービス・プログラムを呼び出すことはできます。これは、少なくとも 1 つのユーザー入力プロシージャーが含まれているためです。サービス・プログラムには、ユーザー入力プロシージャーではなく、データ・エクスポートが含まれる場合があります。
共通インターフェース

サービス・プログラムの共通インターフェース は、他の ILE オブジェクトによって参照可能なエクスポートされるプロシージャーおよびデータ項目の名前からなります。データ項目を ILE サービス・プログラムからエクスポートするには、ILE サービス・プログラムを構成するモジュール・オブジェクトの 1 つからデータ項目をエクスポートする必要があります。

サービス・プログラムの共通インターフェースを指定するには、エクスポート・リストを使用します。バイナダー言語にリストされたプロシージャーおよびデータ項目の名前から、インターフェース識別値が生成されます。このインターフェース識別値を使用して、サービス・プログラムへのインターフェースを検証することができます。

共通インターフェースが変更されない限り、サービス・プログラムを変更しても、サービス・プログラムのクライアントを再コンパイルする必要はありません。

サービス・プログラム作成時の考慮事項

サービス・プログラム作成の際、以下のことを考慮してください。

- 後日プログラムを更新する意図があるかどうか
- 更新にインターフェースの変更が含まれるかどうか

サービス・プログラムに対するインターフェースが変更された場合には、元のサービス・プログラムにバインドされているすべてのプログラムの再バインドが必要なことがあります。しかし、変更とその実装方法によっては、バインダー言語を使用してサービス・プログラムを作成すると、再バインドの量を削減できる場合があります。この場合には、新しいエクスポートを識別するためのバインダー言語ソースの更新後に、その新しいエクスポートを必要とするプログラムだけを再バインドする必要があります。

サービス・プログラムのエクスポートに関する追加情報については、4-23 ページの『モジュールの表示コマンドを使用した、エクスポートされた定義済みシンボルの表示』を参照してください。

バインダーを使用したサービス・プログラムの作成

サービス・プログラムを作成するには、ソースコードをモジュール・オブジェクトにコンパイルしてから、サービス・プログラムの作成 (CRTSRVPGM) コマンドを使用して 1 つ以上のモジュール・オブジェクトを 1 つずつサービス・プログラム・オブジェクトにバインドします。また、ILE C/C++、ILE RPG、ILE COBOL など、他の ILE 言語コンパイラで作成されたモジュールを使用することもできます。

IBM i: ILE C/C++ プログラマの手引き
CRTSRVPGM コマンドのパラメーターの指定

表4-5 に、CRTSRVPGM コマンドのパラメーターおよびそのデフォルト値をリストします。パラメーターの詳細な説明については、CL Reference CHKxxx through CVTxxx Commands (SC41–5724) を参照してください。各パラメーターには、独自の値が指定されていない場合に使用されるデフォルト値があります。

表4-5. CRTSRVPGM コマンドのパラメーターおよびデフォルト値

<table>
<thead>
<tr>
<th>パラメーター・グループ</th>
<th>パラメーター (デフォルト値)</th>
</tr>
</thead>
<tbody>
<tr>
<td>識別</td>
<td>SRVPGM(ライブラリ名/サービス・プログラム名) MODULE(*SRVPGM)</td>
</tr>
<tr>
<td>プログラム・アクセス</td>
<td>EXPORT(*SRCFILE) SRFILE(*LIBL/QSRVSRC) SRCMBR(*SRVPGM)</td>
</tr>
<tr>
<td>バインディング</td>
<td>BNDSRVPGM(*NONE) BNDDIR(*NONE)</td>
</tr>
<tr>
<td>実行時</td>
<td>ACTGRP(*CALLER)</td>
</tr>
<tr>
<td>その他</td>
<td>OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF) DETAIL(*NONE) ALWUPD(*YES) ALWRINZ(*NO) REPLACE(*YES) AUT(*LIBCRTCNTAX) TEXT(*ENTMODTXT) TGTRLS(*CURRENT) USRPRF(*USER) STGMDL(*SNGLVL) IPA(*NO) IPACTLFILE(*NONE)</td>
</tr>
</tbody>
</table>

サービス・プログラムの更新または変更

サービス・プログラムは、プログラム・オブジェクトの変更と同じ方法で更新または変更することができます。すなわち、以下のことが行えます。

- サービス・プログラムの更新 (サービス・プログラムの更新 (UPDSRVPGM) コマンドを使用)
- 最適化レベルの変更 (サービス・プログラムの変更 (CHGSRVPGM) コマンドを使用)
- プログラム識別情報の除去 (CHGSRVPGM を使用)
- サイズの削減 (オブジェクトの圧縮 (CPROBJ) コマンドを使用)

上記の事項について詳しくは、4-13 ページの『モジュールまたはプログラム・オブジェクトの更新』を参照してください。

バインダー言語を使用する場合、サービス・プログラムを更新するために、それを呼び出すプログラムの再コンパイルは必要ありません。例えば、既存のサービス・プログラムに新しいプロシージャーを追加するには、次のようにします。

1. 新しいプロシージャーのモジュール・オブジェクトを作成する。
2. 新しいプロシージャーと関連したインターフェースを処理するためにバインダー言語のソース・ファイルを変更する。既存のステートメントの後に、新しいエクスポート・ステートメントを追加します。バインダー言語ソース・ファイルの変更の詳細については、4-25 ページの『サービス・プログラム・エクスポート・リストの更新』を参照してください。
3. 元のサービス・プログラムを再作成して、新しいモジュールを組み込みます。
これで、既存プログラムは新しい関数にアクセスできます。旧エクスポートが同じ順序なので、既存のプログラムはまだこれを使用することができます。既存のプログラムの更新も必要となるまで、これらを再コンパイルする必要はありません。

サービス・プログラムでの制御言語 (CL) コマンドの使用
サービス・プログラムでは、以下の CL コマンドを使用することができます。
- サービス・プログラムの作成 (CRTSRVPGM)
- サービス・プログラム変更 (CHGSRVPGM)
- サービス・プログラム表示 (DSPSRVPGM)
- サービス・プログラム削除 (DLTSRVPGM)
- サービス・プログラム更新 (UPDSRVPGM)
- サービス・プログラムの処理 (WRKSRVPGM)

サービス・プログラムの作成、コンパイル、およびバインド
このセクションの例では、他のプログラムから呼び出され任意の文字列で文字ストリングを検索するサービス・プログラム SEARCH を作成する方法を示します。

この部では、以下の方法について説明します。
- ソース・ファイルの作成
- サービス・プログラムのコンパイルとバインド
- サービス・プログラムのプログラムへのバインド

ソース・ファイルの作成
SEARCH プログラムは、クラス・オブジェクト Search として実装されます。クラス Search には、以下が含まれます。
- 3 つの private データ・メンバー: skippat、needle_p、および needle_size
- 3 つのコンストラクター。それぞれ異なる引数を取ります。
- デストラクター
- 多重定義関数 where()。4 つの異なる引数セットを取ります。

サービス・プログラムは、以下のファイルで構成されます。
- ユーザー定義ヘッダー・ファイル search.h
- ソース・コード・ファイル search.cpp
- ソース・コード・ファイル where.cpp

ユーザー・ヘッダー・ファイル: クラスと関数の宣言は、次の図に示したような別のヘッダー・ファイル search.h に配置されます。
// header file search.h
// contains declarations for class Search, and inlined function
// definitions

#include <iostream.h>

class Search {  
private:
  char skippat[256];
  char * needle_p;
  int needle_size;
public:

  // Constructors
  Search( unsigned char * needle, int size);
  Search ( unsigned char * needle);
  Search ( char * needle);

  //Destructor
  ~Search () { delete needle_p;}

  //Overloaded member functions
  unsigned int where ( char * haystack) {
    return where (haystack, strlen(haystack));
  }
  unsigned int where ( unsigned char * haystack) {
    return where (haystack, strlen((const char *)haystack));
  }
  unsigned int where ( char * haystack, int size) {
    return where ( (unsigned char *) haystack, size);
  }
  unsigned int where ( unsigned char * haystack, int size);
};

図 4-4. ヘッダー・ファイル (search.h) の例

ソース・コード・ファイル: クラス Search のメンバー関数の定義がクラス宣言でインライン化されていない場合、それは次の 2 つの別ファイルに含まれます。
- ソース・ファイル search.cpp。クラス Search のコンストラクタ定義を含みます。
- ファイル where.cpp。メンバー関数定義を含みます。
これらのファイルを以下の図に示します。
// source file search.cpp
// contains the definitions for the constructors for class Search

#include "search.h"

Search::Search( unsigned char * needle, int size)
    : needle_size(size), needle_p ( new char [size])
{
    memset (skippat, needle_size, 256);
    for (unsigned int i=0; i< size; ++i) {
        skippat[needle[i]] = size -i-1;
    }
    memcpy (needle_p, needle, needle_size);
}

Search::Search (unsigned char * needle) {
    needle_size = strlen( const char *)needle);
    needle_p = new char [needle_size];
    memset (skippat, needle_size, 256);
    for (unsigned int i=0; i< needle_size; ++i) {
        skippat [needle[i]] = needle_size -i-1;
    }
    memcpy(needle_p, needle, needle_size);
}

Search::Search ( char * needle) {
    needle_size = strlen( needle);
    needle_p = new char [needle_size];
    memset (skippat, needle_size, 256);
    for (unsigned int i=0; i< needle_size; ++i) {
        skippat [needle[i]] = needle_size -i-1;
    }
    memcpy (needle_p, needle, needle_size);
}

unsigned int Search::where ( unsigned char * haystack, int size)
{
    unsigned int i, t;
    int j;
    for ( i = needle_size-1, j = needle_size-1; j >= 0; --i, --j ){
        while ( haystack[i] != needle_p[j]) {
            t = skippat [haystack [i]] ;
            i += (needle_size - j > t) ? needle_size - j : t;
            if ( i >= size)
                return size;
            j = needle_size - 1;
        }
    }
    return ++i;
}

これらのソース・ファイルをコンパイルして生成されたモジュール SEARCH と WHERE は、サービス・プログラム SERVICE1 にバインドされます。

サービス・プログラムのコンパイルおよびバインディング
サービス・プログラム SERVICE1 を作成するには、次のコマンドを実行してください。
デフォルトでは、バインダーは現行ライブラリー内部サービス・プログラムを作成します。

パラメーター `export(*all)` は、モジュールからエクスポートされたすべてのデータおよびプロシージャーがサービス・プログラムからもエクスポートされることを指定します。

プログラムへのサービス・プログラムのバインド
以下の例では、プログラム MYPROGA から構築されるごく短いアプリケーションがサービス・プログラムにバインドされています。MYPROGA、MYPROGA.cpp のソース・コードを以下の図に示します。

注: このサンプル・アプリケーションは、最小限の機能に縮小されています。このアプリケーションの主な目的は、サービス・プログラムの作成方法を例示することです。

```c++
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include "search.h"
#define HS "Find the needle in this haystack"

void main () {
  int i;
  Search token("needle");
  i = token.where (HS, sizeof(HS));
  cout << "The string was found in position " << i << endl;
}
```

図4-7. myproga.cpp のソース・コード

このプログラムは、クラス Search のオブジェクトを作成します。これにより、検索対象の文字列 ("needle") を表す値を使用したコンストラクターが呼び出されます。メンバー関数 where() と検索するストリング ("Find the needle in this haystack") が呼び出されます。ストリング "needle" の場所が探索され、ターゲット・ストリング "Find a needle in this haystack" 内のその位置が返され、出力されます。

ライブラリー MYLIB 内にプログラム MYPROGA を作成し、それをサービス・プログラム SERVICE1 にバインドするには、以下を入力します。

```
CRTSRVPGM SRVPGM(MYLIB/SERVICE1) MODULE(MYLIB/SEARCH MYLIB/WHERE) EXPORT(*ALL)
```

4-22 ページの図4-8 は、プログラム MYPROGA とサービス・プログラム SERVICE1 の間の内部および外部関数呼び出しを示しています。
図4.8 プログラムとサービス・プログラムの間の呼び出し

MYPROGA が作成される際、その中にはサービス・プログラムとの対話に使用するインターフェースに関する情報を組み込まれます。

プログラムを実行するには、次のように入力してください。

CALL MYLIB/MYPROGA

MYPROGA を実行可能にする処理の段階で、システムは以下のことを検査します。

- ライブラリー MYLIB にサービス・プログラム SERVICE1 が存在すること。
- 作成時に MYPROGA で使用された共通インターフェースが、実行時にも有効であること。

上のいずれかが真でなければ、エラー・メッセージが出されます。

MYPROGA の出力を以下に示します。
The string was found in position 9

サービス・プログラムからのエクスポートの処理

このセクションでは、サービス・プログラムからエクスポートできるプロシージャーおよびデータ項目を処理する方法について説明します。

ハードウェアからのエクスポートの判別

サービス・プログラムは、他のプログラムでインポートできるプロシージャーおよびデータ項目をエクスポートします。このエクスポートは、サービス・プログラムへのインタフェースを表します。C/C++ プログラミング言語では、プロシージャーおよびデータ項目は、関数および変数に対応しています。

エクスポートに関する情報（特定のサービス・プログラムを形成するモジュールから派生する）を使用して、バイナリ言語ソース・ファイルを作成することができます。このファイルでは当該サービス・プログラムへのインターフェースを定義します。バイナリ言語ソース・ファイルは、サービス・プログラムを呼び出すすべてのプログラムに対して、当該サービス・プログラムが使用可能にするエクスポートを指定します。このファイルを、CRTSRVPGM コマンドの EXPORT パラメーターに指定できます。

バイナリ言語によって、サービス・プログラムのエクスポートに対する制御性が向上します。以下のことを行いたい場合には、この制御が非常に有用です。
・アプリケーション内のエクスポートとインポートの不一致を判別する。
・サービス・プログラムに機能を追加する。
・アプリケーションのユーザーよサービス・プログラムへの変更の影響を軽減する。
・サービス・プログラム・ユーザーや特定のサービス・プログラムのエクスポートをマスクする。つまり、バイナダー言語ソース・ファイルに特定の関数または変数をリストしないことで、呼び出し側プログラムが当該エクスポートにアクセスできないようにすることができます。

モジュールの表示コマンドを使用した、エクスポートされた定義済みシンボルの表示
モジュールで使用可能なエクスポートを判別するには以下を入力します。

DSPMOD MODULE(library-name/module-name)

モジュール名およびモジュールが保管されているライブラリーを指定します。
このコマンドは、モジュール情報の表示画面を開きます。この画面の下部に、モジュールからエクスポートできる各シンボルの名前とタイプがあります。

注：コンパイラはソース・ファイルをコンパイルする際に、関数名および特定の変数をエンコードして、タイプおよびスコープ情報を含めます。このエンコード・プロセスは、名前マングリングと呼ばれています。以下のサンプル画面内のシンボル名は、マングル形式で表示されています。モジュール SEARCH のソース・コードは、4-19ページの『ソース・コード・ファイル』に示されています。

<table>
<thead>
<tr>
<th>モジュール情報の表示</th>
</tr>
</thead>
<tbody>
<tr>
<td>モジュール：SEARCH</td>
</tr>
<tr>
<td>ライブラリー：MYLIB</td>
</tr>
<tr>
<td>詳細：EXPORT</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>エクスポートされた定義済み記号</td>
</tr>
<tr>
<td>__ct__6SearchFPc</td>
</tr>
<tr>
<td>__ct__6SearchFPUc</td>
</tr>
<tr>
<td>__ct__6SearchFPUc1</td>
</tr>
</tbody>
</table>

図4.9 サンプル・モジュール SEARCH のモジュール情報の表示画面

バイナダー言語ソース・ファイルの作成
バイナダー言語は、サービス・プログラムにバインドされたモジュールから使用可能なエクスポートに基づいています。バイナダー言語ソース・ファイルには、以下の項目が含まれている必要があります。

・プログラム・エクスポート・リストの開始 (STRPGMEXP) コマンドは、サービス・プログラムからのエクスポートのリストの始まりを識別します。
・シンボルのエクスポート (EXPORT) コマンドは、サービス・プログラムからエクスポートできるシンボル名を識別します。
・プログラム・エクスポートの終了 (ENDPGMEXP) コマンドは、サービス・プログラムからのエクスポートのリストの終わりを識別します。

以下の図では、バイナダー言語ソース・ファイルの構造を示します。
STRPGEXP PGMLEVEL(*CURRENT)
EX  PORT SYMBOL("mangled_procedure_name_a")
EX  PORT SYMBOL("mangled_procedure_name_b")
...
EX  PORT SYMBOL("mangled_procedure_name_x")
ENDPGMEXP

図4-10. バインダー・ソース・ファイルの例

注: EXPORT コマンドで各シンボルのマングル名を指定する必要があります。これにより、バインダーが、他
のモジュールからのインポート要求を解決しようとする際に、エクスポートのマングル名を検索するからで
す。

サービス・プログラムにバインドするすべてのモジュールが作成されて、以下のいずれかの方法を使用し
て、バインダー・ソース・ファイルを作成できます。
- 原始ステートメント入力ユーティリティ (SEU) を使用してファイルを自身で作成できます。
- バインダー・ソース検索 (RTVBNDSRC) コマンドを使用して、スケルトン・バインド・ソースを生成で
きます。

SEU を使用したバインダー言語の作成
原始ステートメント入力ユーティリティ (SEU) を使用してバインダー言語ソース・ファイルを作成する
には、以下のステップに従います。
1. ライブラリー MYLIB にソース物理ファイル QSRVSRC を作成します。
2. バインダー言語が入るメンバー MEMBER1 を作成します。
3. モジュールの表示 (DSPMOD) コマンドを使用して、各モジュールからエクスポートできる記号を表示
します。
4. 呼び出し経由プログラムから使用可能にするエクスポートを決定します。
5. 原始ステートメント入力ユーティリティ (SEU) を使用して、バインダー言語の構文を入力します。

サービス・プログラムの呼び出し元から使用可能にするエクスポートを含む各プロシージャーに 1 つの
export ステートメントが必要です。呼び出し経由プログラムから使用可能にしない記号をリストしないでく
ださい。

例えば、4-23 ページの図4-9 に示した情報を基に、モジュール SEARCH のバインダー言語ソース・ファイ
ルでは、以下の EXPORT SYMBOL をリストします。

STRPGEXP PGMLEVEL(*CURRENT)
EX  PORT SYMBOL("__ct__6SearchFPc")
EX  PORT SYMBOL("__ct__6SearchFPUc")
EX  PORT SYMBOL("__ct__6SearchFPUci")
ENDPGMEXP

RTVBNDSRC コマンドを使用したバインダー言語の作成
バインダー・ソース検索 (RTVBNDSRC) コマンドでは、バインダー言語ソース・ファイルを自動的に作成
することができます。これにより、モジュールまたはモジュール・セットからエクスポートを検索します。これ
らのエクスポートのバインダー言語を生成し、指定されたファイル・メンバーにエクスポートとバインダー
言語を入力します。このファイル・メンバーを、後で、サービス・プログラム作成 (CRTSRVPGM) コマンド
の EXPORT パラメーターへの入力として使用することができます。
注: バインダー言語を検索してソース・ファイル・メンバーに入れた後で、そのバインダー言語を編集して、必要に応じて（例えば、モジュールを変更する場合や、特定のエクスポートを呼び出し側プログラムから使用不能にする場合など）それを変更できます。

RTVBNDSRC コマンドの構文は次のとおりです。

```
RTVBNDSRC MODULE(MYLIB/SEARCH) SRCFILE(MYLIB/QSRVSRC) SRCMBR(*DFT) MBROPT(*REPLACE)
```

RTVBNDSRC コマンドとそのパラメーターの詳細については、コマンド行で RTVBNDSRC と入力し、F1 を押してヘルプを表示します。

以下の例では、RTVBNDSRC コマンドを使用して、ライブラリー MYLIB にあるモジュール SEARCH のバインダー言語ソース・ファイルを作成する方法を示します。モジュール SEARCH のソース・コードは、4-19ページの『ソース・コード・ファイル』に示されています。

```
RTVBNDSRC MODULE(MYLIB/SEARCH) SRCFILE(MYLIB/QSRVSRC) SRCMBR(ONE)
```

このコマンドは、自動的に以下を行います。
1. ライブラリー MYLIB にソース物理ファイル QSRVSRC を作成します。
2. QSRVSRC にメンバー ONE を追加します。
3. ライブラリー MYLIB のモジュール SEARCH からバインダー言語を生成し、メンバー ONE に入れます。

ファイル MYLIB/QSRVSRC 内のメンバー ONE には、現在、以下のバインダー言語が含まれています。

<table>
<thead>
<tr>
<th>桁</th>
<th>...</th>
<th>1</th>
<th>7</th>
<th>走査検索</th>
<th>MYLIB/QSRVSRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEU=&gt;</td>
<td>...</td>
<td>1</td>
<td>...</td>
<td>2</td>
<td>...</td>
</tr>
</tbody>
</table>

```
0000.01 STRPGMEXP PGMVL(*CURRENT)
0000.02 /*********************************************************************/
0000.03 /* *MODULE SEARCH MYLIB 95/06/10 17:34:41 */
0000.04 /*********************************************************************/
0000.05 EXPORT SYMBOL("__ct__6SearchFPc")
0000.06 EXPORT SYMBOL("__ct__6SearchFPuc")
0000.07 EXPORT SYMBOL("__ct__6SearchFPuc1")
0000.08 ENDPGMEXP
```

図4-11. モジュール SEARCH に生成されたバインダー言語ソース・ファイル

サービス・プログラム・エクスポート・リストの更新

バインダー言語を使用して、サービス・プログラムで利用可能なエクスポートのリストで変更を反映させることができます。バインダー言語を作成すると、インターフェース識別子は、サービス・プログラムを構成するモジュールが処理された順序、およびこれらのモジュールから記号がエクスポートされた順序から生成されます。バインダー言語の EXPORT キーワードは、サービス・プログラムのインターフェース識別子を構成するプロシージャー名とデータ項目名を識別します。

サービス・プログラムのエクスポートを変更した場合、このサービス・プログラムを呼び出すすべてのプログラムを再作成する必要があるとは限りません。後方互換性を維持するようにバインダー言語で変更を実装することが可能です。後方互換とは、変更されないエクスポートに依存するプログラムは再作成する必要がないことを意味します。
デマングル関数の使用

RTVBNDSRC コマンドを利用して、エクスポートされたシンボルのマングル名を取得できます。対応するデマングル名を見つけるのに役立つように、ランタイム・ライブラリーには、名前をデマングルとして名前の結果部分を調べるために使用できる、小さな関数クラス階層が含まれています。インタフェースについては、<demangle.h> ヘッダー・ファイルで文書化されています。

デマングル関数を使用して、マングル名をデマングル名に変換したり、その名前の特性（タイプ修飾子やスコープなど）を判別したりするプログラムを作成できます。例えば、プログラムに関数のマングル名を渡すと、その関数のデマングル名および修飾子の名前が返されます。マングル名がクラス・メンバーを参照している場合、それが static, const, または volatile のいずれであるのかを判別できます。また、マングル名のテキスト全体を取得することもできます。

文字列として表される名前をデマングルするには、Name クラスの動的インスタンスを作成し、文字ストリングをクラスのコンストラクターに渡します。例えば、名前 f__1XF1 をデマングルするには、以下を作成します。

```c
char *rest;
Name *name = Demangle("f__1XF1", rest);
```

デマングル関数は、名前を 5 つの分類（関数名、メンバ関数名、特殊名、クラス名、およびメンバ変数名）に分類します。クラス Name のインスタンスを構成した後に、Name の Kind メンバ関数を使用して、インスタンスがどのような種類の Name なのかを判別できます。返された名前の中の種類に基づいて、名前の別の部分のテキストまたは名前全体のテキストを要求できます。

マングル名 f__1XF1 の場合、以下を判別できます。

```c
name->Kind() == MemberFunction
(MemberFunctionName *) name)->Scope()->Text() is "X"
(MemberFunctionName *) name)->RootName() is "f"
(MemberFunctionName *) name)->Text() is "X::f(int)"
```

Name コンストラクターに渡された文字ストリングがマングル名ではない場合、Demangle 関数は NULL を返します。

デマングル関数について詳しくは、demangle.h ヘッダー・ファイルに含まれている情報を参照してください。デフォルト設定を使用して ILE C/C++ をインストールした場合は、このヘッダー・ファイルは、IFS では /QIBM/include ディレクトリー、DM では QSYSINC/H ディレクトリーにあります。
プログラム作成中の未解決インポート要求の処理

未解決インポート とは、インポートのタイプと名前がエクスポートのタイプと名前にまだ一致しないインポートです。未解決インポート要求ではプログラムおよびサービス・プログラムの作成が必ずしもできないとは限りません。次の 2 つの方法で行うことができます。

・ CRTPGM コマンドまたは CRTSRVPGM コマンドの *UNRSLVREF オプションを指定して、モジュールにインポートがあって、一致するエクスポートが見つからなくても、そのままプログラムまたはサービス・プログラムを作成するようにバイナダーに指示します。

・ プログラム作成順序を変更して、未解決の参照がないようにします。

4-28 ページの『循環参照のあるプログラムの作成』に、この 2 つのアプローチの例を示しています。

コードのすべての断片がまだ利用可能ではないときに、*UNRSLVREF オプションを使用して、コードの断片を変換、作成、またはビルドします。開発または変換のフェーズが終了し、すべてのインポート要求が解決可能になった後、未解決インポートを含むプログラムまたはサービス・プログラムを再作成してください。

*UNRSLVREF オプションを使用する場合は、見つからないプロシージャー名またはデータ項目名を特定するために、DETAIL(*EXTENDED) または DETAIL(*FULL) を指定するか、オブジェクト作成時のジョブ・ログを保持してください。

注：*UNRSLVREF を指定し、作成したプログラムに未解決のインポート要求が含まれる場合、このプログラムを実行しようとすると、エラー・メッセージ（MCH3203）を受け取ります。

バインダー言語を使用したエクスポート・サービス・プログラムの作成

C++ モジュールの作成 (CRTPPMMOD) コマンドは、一度に 1 つのモジュールのみを作成します。ソース・ストリーム・ファイルまたはソース・ファイル・メンバーごとに CRTPPMOD を使用する必要があります。以下の例は、2 つのモジュール SEARCH および WHERE で構成されています。

例：

4-18 ページの『サービス・プログラムの作成、コンパイラ、およびバインダー』で説明されているサービス・プログラムを作成するためにバインダー言語を使用するには、以下のステップに従います。

1. すべてのソース・ファイルからモジュールを作成するには、以下のコマンドを入力します。

   CRTCPMMOD MODULE(MYLIB/SEARCH) SRCSTMF(search.cpp)
   CRTCPMMOD MODULE(MYLIB/WHERE) SRCSTMF(where.cpp)

   注：CRTCPMMOD コマンドは、*MODULE オブジェクトの作成後にコンパイル・プロセスを停止します。バインダーは呼び出されません。

2. 対応するバインダー言語ソース・ファイルを作成するには、以下のコマンドを入力します。

   RTVBNDSRC MODULE(MYLIB/SEARCH MYLIB/WHERE)
   SRCFILE(MYLIB/QRVSRC) SRCMGR(TWO)

   このコマンドは、4-28 ページの図 4-12 に示されているバインダー言語ソース・ファイルを作成します。

3. サービス・プログラム SERVICE2 を作成するには、以下のコマンドを入力します。

   CRTSRVPGM SRVPGM(MYLIB/SERVICE2) MODULE(MYLIB/SEARCH MYLIB/WHERE)
   SRCFILE(MYLIB/QRVSRC) SRCMGR(TWO)
循環参照のあるプログラムの作成

循環参照は、未解決のインポート要求の特殊なケースです。これが行われるのは、例えば、サービス・プログラム SP1 がサービス・プログラム SP2 からのインポートに依存し、同様にサービス・プログラム SP2 がサービス・プログラム SP1 からのインポートに依存する場合などです。

図4-13は、プログラム A と 2 つのサービス・プログラム SP1 および SP2 の間の未解決のインポート要求を示しています。

以下のインポート要求は、プログラム A と、プログラム A によって呼び出される 2 つのサービス・プログラム SP1 および SP2 の間で行われます。
1. プログラム A は、サービス・プログラム SP1 からインポートする関数 func1() を使用します。
2. サービス・プログラム SP1 が func1() をプログラム A に提供するには、サービス・プログラム SP2 によって提供される関数 func2() をインポートする必要があります。
3. 同様に、サービス・プログラム SP2 が func2() を提供するには、まずサービス・プログラム SP1 から func1 をインポートする必要があります。
未解決の循環参照のあるプログラムを作成するには、以下のタスクを実行してください。

1. **ソース・ファイルを作成します。**
2. **ソース・ファイルをコンパイルしてモジュールを作成します。**
3. **バインダー言語を作成します。**
4. **モジュールをプログラムにバインドします。**

**ソース・ファイルの作成**
アプリケーションは、以下の図で示しているように、3 つのソース・ファイル (m1.cpp、m2.cpp、および m3.cpp) で構成されています。

```
// m1.cpp
#include <iostream.h>
int main(void)
{
    void func1(int);
    int n = 0;
    func1(n);  // Function func1() is called.
}
```

図 4-14. m1.cpp — 循環参照があるアプリケーションの 1 つ目のソース・ファイル

```
// m2.cpp
#include <iostream.h>
void func2(int);
void func1(int x)
{
    if (x<5)
    {
        x += 1;
        cout << "This is from func1(), n=" << x << endl;
        func2(x);  // Function func2() is called.
    }
}
```

図 4-15. m2.cpp — 循環参照があるアプリケーションの 2 つ目のソース・ファイル

```
// m3.cpp
#include <iostream.h>
void func1(int);
void func2(int y)
{
    if (y<5)
    {
        y += 1;
        cout << "This is from func2(), n=" << y << endl;
        func1(y);  // Function func1() is called.
    }
}
```

図 4-16. m3.cpp — 循環参照があるアプリケーションの 3 つ目のソース・ファイル

**ソース・ファイルのモジュールへのコンパイル**
ソース・ファイル m1.cpp をモジュール・オブジェクト (後でこのモジュール・オブジェクトからプログラ
ム A を作成します) にコンパイルします。これにより、DSPMOD コマンドでエクスポートを表示すること
や、RTVBND SRC コマンドでバインダー言語ソースを生成することが可能になります。
サービス・プログラムを作成するためのバイナダー言語の生成
サービス・プログラム SP1 を作成する元となるモジュール M2 のバイナダー言語を生成するには、以下のコマンドを入力します。

RTVBNDSRC MODULE(MYLIB/M2) SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG1)

このコマンドにより、ライブラリー MYLIB、ソース・ファイル QSRVSRC、ファイル・メンバー BNDLANG1 に、モジュール M2 の以下のバイナダー言語が作成されます。

プログラムへのモジュールのバインド
プログラム A は M1 から作成されます。サービス・プログラム SP1 は M2 から作成されます。サービス・プログラム SP2 は M3 から作成されます。
4-30 ページの図 4-17 に示すバイナダー言語と次のコンバイラー呼び出しを使用してモジュール M2 からサービス・プログラム SP1 の試行と作成を行う場合、

CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(MYLIB/M2)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG1)

バイナダーは関数 func2() のためのインポートを解決しようとしますが、一致するエクスポートが見つからないため失敗します。このため、サービス・プログラム SP1 は作成されません。

SP1 が作成されない場合、4-30 ページの図 4-18 に示すバイナダー言語と次のコンバイラー呼び出しを使用してモジュール M3 からサービス・プログラム SP2 の試行と作成を行うと、問題が発生します。

CRTSRVPGM SRVPGM(MYLIB/SP2) MODULE(MYLIB/M3)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG2)

バイナダーが、前のステップで作成されなかったサービス・プログラム SP1 での func1() のためのインポートの検索に失敗するため、サービス・プログラム SP2 は作成されません。

次のコンバイラー呼び出しを使用してプログラム A の試行および作成を行う場合、

CRTPGM PGM(A) MODULE(MYLIB/M1) BNDSRVPGM(MYLIB/SP1 MYLIB/SP2)

サービス・プログラム SP1 と SP2 が存在しないため、バイナダーは失敗します。

*UNRSLVREF パラメーターを使用した未解決インポート要求の処理

下記の例では、次のことを示します。

• m2.cpp からサービス・プログラム SP1 を作成する方法。

• パラメーター *UNRSLVREF を使用して、未解決インポート要求を処理する方法。これを使用しない場合、プログラム A は作成できません。

例:

1. m2.cpp からサービス・プログラム SP1 を作成するために、次のように入力します。

   CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(MYLIB/M2)
   SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG1)
   OPTION(*UNRSLVREF)

   *UNRSLVREF オプションが指定されたため、func2() のインポート要求が解決されなくても、サービス・プログラム SP1 は作成されます。

2. モジュール M3 とサービス・プログラム SP1 からサービス・プログラム SP2 を作成するために、次のように入力します。

   CRTSRVPGM SRVPGM(MYLIB/SP2) MODULE(MYLIB/M3)
   SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG2)
   BNDSRVPGM(MYLIB/SP1)

   サービス・プログラム SP1 は存在するため、バイナダーは必要なすべてのインポート要求を解決し、サービス・プログラム SP2 は正常に作成されます。

3. サービス・プログラム SP1 を再作成するために、次のように入力します。

   CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(MYLIB/M2)
   SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG1)
   BNDSRVPGM(MYLIB/SP2)
サービス・プログラム SP1 は存在しますが、func2() のインポート要求は解決されません。そのため、サービス・プログラム SP1 の再作成が要求されます。サービス・プログラム SP2 は存在するため、バインダーは必要なすべてのインポート要求を解決し、サービス・プログラム SP1 は正常に作成されます。

4. プログラム A を作成するために、次のように入力します。

```
CRTPGM PGM(MYLIB/A) MODULE(MYLIB/M1) BNDSRVPGM(MYLIB/SP1 MYLIB/SP2)
```

サービス・プログラム SP1 と SP2 は存在するため、バインダーはプログラム A を作成します。

**プログラム作成順序の変更による未解決インポート要求の処理**

プログラム作成順序を変更し、まず、すべてのモジュールでサービス・プログラムを作成してから、後でこの同じサービス・プログラムを再作成することで、未解決の参照回避することも可能です。

1. サービス・プログラム SP1 の作成元のモジュール M2 と M3 のバインダー言語を生成するために、次のコマンドを実行します。

```
RTVBNDSRC MODULE(MYLIB/M2 MYLIB/M3) SRCFILE(MYLIB/QSRVSRC)
SRCMBR(BNDLANG3)
```

このコマンドにより、図 4-19 に示したバインダー言語が、ライブラリ MYLIB、ソース・ファイル QSRVSRC、ファイル・メンバー BNDLANG3 に作成されます。

2. モジュール M2 とモジュール M3 からサービス・プログラム SP1 を作成するために、次のように入力します。

```
CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(MYLIB/M2 MYLIB/M3)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG3)
```

モジュール M2 と M3 が指定されたため、すべてのインポート要求が解決され、サービス・プログラム SP1 は正常に作成されます。

3. サービス・プログラム SP2 を作成するために、次のように入力します。

```
CRTSRVPGM SRVPGM(MYLIB/SP2) MODULE(MYLIB/M2 MYLIB/M3)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG2)
BNDSRVPGM(MYLIB/SP1)
```

サービス・プログラム SP1 は存在するため、バインダーは必要なすべてのインポート要求を解決し、サービス・プログラム SP2 は正常に作成されます。

4. サービス・プログラム SP1 を再作成するために、次のように入力します。

```
CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(MYLIB/M2)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG1)
BNDSRVPGM(MYLIB/SP2)
```

サービス・プログラム SP1 は存在しますが、func2() のインポート要求は、サービス・プログラム SP2 のものに解決されません。そのため、循環参照を機能させるためにサービス・プログラム SP1 の再作成が必要です。

サービス・プログラム SP2 は存在するため、バインダーはサービス・プログラム SP2 から func2() のインポート要求を解決することができ、サービス・プログラム SP1 は正常に作成されます。

5. プログラム A を作成するために、次のように入力します。

```
CRTPGM PGM(MYLIB/A) MODULE(MYLIB/M1) BNDSRVPGM(MYLIB/SP1 MYLIB/SP2)
```

サービス・プログラム SP1 と SP2 は存在するため、バインダーはプログラム A を作成します。
存在しないサービス・プログラムへのプログラムのバインド

プログラムまたはサービス・プログラムを正常に作成するには、バインダーを呼び出す前に、必要なすべてのモジュールが存在している必要があります。

ただし、存在しないサービス・プログラムにプログラムをバインドする場合、まず、「プレースホルダー」サービス・プログラムを作成することができます。次の例を検討してみます。

プログラム MYPROG では、関数 print() をサービス・プログラム PRINT によってエクスポートする必要があります。このプログラムのコードは、myprog.cpp にあります。ただし、サービス・プログラムのソースはまだ存在しません。

この問題を正すために、サンプル・コードの図に示されているソース・コードを使用して、このセクションの指示に従います。

手順

1. **4-34 ページの図 4-20** に示すソース・コードを使用して、ソース・ファイル dummy.cpp を作成します。
2. dummy.cpp をコンパイルし、サービス・プログラム PRINT にバインドします。
   ```
   CRTCPPMOD MODULE(MYLIB/DUMMY) SRCSTMF(dummy.cpp)
   CRTSRVPGM SRVPGM(MYLIB/PRINT) MODULE(MYLIB/DUMMY) EXPORT(+ALL)
   ```
3. **4-34 ページの図 4-20** に示すソース・コードを使用して、プログラム MYPROG のソース・ファイルを作成します。
4. プログラム MYPROG を myprog.cpp から作成し、それをサービス・プログラム PRINT にバインドします。以下のコマンドを入力します。
   ```
   CRTCPPMOD MODULE(MYLIB/MYPROG) SRCSTMF(myprog.cpp)
   CRTPGM PGM(MYLIB/MYPROG) MODULE(MYLIB/MYPROG)
   BNDSRVPGM(MYLIB/PRINT) OPTION(*UNRSLVREF)
   ```
   オプション *UNRSLVREF を使用すると、MYPROG のインポート void print(char *) と一致するエクスポートがなくても、プログラムは確実にサービス・プログラムとバインドされます。

コード・サンプル
プログラムの実行
プログラム MYPROG を正常に実行するには、以下を行う必要があります。

- dummy.cpp 内にあるプレースホルダー・コードからではなく、実際のソース・コードからサービス・プログラム PRINT を再作成する。
- プログラム MYPROG を再作成し、そのプログラムを新しいバージョンのサービス・プログラム PRINT とバインドして print() に対する参照を解決する。

注: MYPROG は、PRINT が MYPROG のインポート要求と一致する関数を実際にエクスポートする場合のみ、正常に実行されます。

サービス・プログラム・エクスポート・リストの更新
ILE C/C++ サービス・プログラムに後方互換の変更を行う場合は、バイナダー言語を使用します。この言語によって、エクスポート可能なプロシージャー名とデータ項目名のリストを定義することができます。バイナダー言語のシンボルのエクスポート (EXPORT) コマンドは、サービス・プログラム・モジュールのインターフェース識別値を構成するプロシージャー名とデータ項目名を識別します。

変更での互換性を確保するために、新しいプロシージャー名またはデータ項目名は、エクスポート・リストの末尾に追加してください。インターフェース識別値は、モジュールが処理された順番、およびコピー・モジュールから記号がエクスポートされた順序で生成されます。他の ILE C/C++ プログラムによってエクスポートが使用されると、サービス・プログラムの更新はむずかしくなります。サービス・プログラムを変更すると、エクスポートの順序または個数が変更される可能性があります。インターフェース識別値が変更されると、変更されたサービス・プログラムを使用するすべての ILE C/C++ プログラムおよびサービス・プログラムを再作成しなければなりません。

以下の例では、COST からのエクスポートを必要とする既存プログラム COSTDPT1 を再作成せずに、サービス・プログラム COST に新しいプロシージャー cost2() を追加する方法を示しています。
プログラムの説明
次の図は、サービス・プログラム COST の既存のバージョンと更新したバージョンにおけるエクスポートを示しています。

図 4.22. サービス・プログラム COST からのエクスポート

次の図は、既存プログラム COSTDPT1 と新しいプログラム COSTDPT2 におけるインポート要求を示しています。

図 4.23. プログラム COSTDPT1 と COSTDPT2 におけるインポート要求

古いバージョンのサービス・プログラム COST のバインダー言語は、ライブラリー MYLIB のソース・ファイル QSRVSRC のメンバー BND にあります。

```assembly
STRPGMEXP PGMLVL(*CURRENT)
  EXPORT SYMBOL("cost1__Fi9_DecimalTXSP10SP2_")
ENDPGMEXP
```

エクスポート・インターフェース識別値は 94898385315FD06BB65E44D38A852904 です。更新されたバインダー言語には、新しいエクスポート・プローシージャー cost2() が含まれます。これは、ライブラリー MYLIB のソース・ファイル QSRVSRC のメンバー BNDUPD にあります。

```assembly
STRPGMEXP PGMLVL(*CURRENT)
  EXPORT SYMBOL("cost1__Fi9_DecimalTXSP10SP2_")
  EXPORT SYMBOL("cost2__Fi9_DecimalTXSP10SP2_9_DecimalTXSP3SP1_*")
ENDPGMEXP
```

新しいエクスポート・インターフェース識別値は 61E595C21D3EC9FDFD29749FB36B42D0 です。古いサービス・プログラムを定義するバインダー言語ソースで、PGMLVL の値は *CURRENT から *PRV に変更されます。

```assembly
STRPGMEXP PGMLVL(*PRV)
  EXPORT SYMBOL("cost1__Fi9_DecimalTXSP10SP2_")
ENDPGMEXP
```

そのエクスポート・インターフェース識別値は変更されません。

注: 既存プログラムが、再作成なしでそのサービス・プログラムの新しいバージョンを呼び出せるようにする場合は、必ず、以下を行ってください。

1. バインダー言語の記号リストの最後に新しいエクスポートを追加します。
2. サービス・プログラムの古いバージョンのインターフェース識別値と同一のインターフェース識別値を、新しいバージョンに明示的に指定します。

ソース・ファイルの作成
サービス・プログラム COST、モジュール COST2、プログラム COSTOPT1 および COSTOPT2 のソース・コードを次の図に示します。
// cost1.cpp
// contains the export function cost1() for the old service program
#include <iostream.h>
#include <bcd.h>
_DecimalT<10,2> cost1 (int q, // The quantity.
    _DecimalT<10,2> p ) // The price.
{
    _DecimalT<10,2> c; // The cost.
    c = q*p;
    return c;
}

// cost2.cpp
// contains the export function cost2() for the new service program
#include <iostream.h>
#include <bcd.h>
_DecimalT<10,2> cost2 (int quantity, _DecimalT<10,2> price,
    _DecimalT<3,1> discount)
{
    _DecimalT<10,2> c = __D(quantity*price*discount/100);
    return c;
}

// costdpt1.cpp
// This program prompts users (from dept1) to enter the quantity, and price for a product. It uses function cost1() to calculate the cost, and prints the result out.
#include <iostream.h>
#include <bcd.h>
_DecimalT<10,2> cost1(int, _DecimalT<10,2>);
int main(void)
{
    int quantity;
    _DecimalT<10,2> cost;
    _DecimalT<10,2> price;
    cout << "Enter the quantity, please." << endl;
    cin >> quantity;
    cout << "Enter the price, please." << endl;
    cin >> price;
    cost = cost1(quantity, price);
    cout << "The cost is $" << cost << endl;
}

// costdpt2.cpp
// This program prompts users (from dept2) to enter the quantity, price, and discount rate for a product. It uses function cost2() to calculate the cost, and prints the result out.
#include <iostream.h>
#include <decimal.h>
_DecimalT<10,2> cost2(int, _DecimalT<10,2>, _DecimalT<3,1>);
int main(void)
{
    int quantity;
    _DecimalT<10,2> price;
    _DecimalT<10,2> cost;
    _DecimalT<3,1> discount;
    cout << "Enter the quantity, please." << endl;
    cin >> quantity;
    cout << "Enter the price, please." << endl;
    cin >> price;
    cout << "Enter the discount, please.( %)" << endl;
    cin >> discount;
    cost = cost2(quantity, price, discount);
    cout << "The cost is $" << cost << endl;
}
プログラムおよびサービス・プログラムのコンパイルとバインド

1. ソース・ファイル cost1.cpp から、ライブラリー MYLIB のソース・ファイル QSRVSR にあるバインダー
   ー、ソース・メンバー BND を使用して、サービス・プログラム COST を作成します。
   CRTCPPMOD MODULE(MYLIB/COST1) SRCSTMF(cost1.cpp)
   CRTSRVPGM SRVPGM(MYLIB/COST) MODULE(MYLIB/COST1) SRCFILE(MYLIB/QSRVSR)
   SRCMBR(BND) DETAIL(*EXTENDED)

2. ソース・ファイル costdpt1.cpp と、ライブラリー MYLIB にあるサービス・プログラム COST から、プ
   ログラム COSTDPT1 を作成します。
   CRTCPPMOD MODULE(MYLIB/COSTDPT1) SRCSTMF(costdpt1.cpp)
   CRTSRVPGM SRVPGM(MYLIB/COSTDPT1) MODULE(MYLIB/COSTDPT1) BNDSRVPGM(MYLIB/COST)

3. ライブラリー MYLIB のソース・ファイル QSRVSR にある更新されたバインダー言語ソース BNDUPD を
   使用して、モジュール COST2 を組み込むようにサービス・プログラム COST を更新します。
   CRTCPPMOD MODULE(MYLIB/COST) SRCSTMF(costdpt1.cpp)
   CRTSRVPGM SRVPGM(MYLIB/COST) MODULE(MYLIB/COST1) COST1 COST2)
   SRCFILE(MYLIB/QSRVSR) SRCMBR(BNDUPD) DETAIL(*EXTENDED)

サービス・プログラム COST は、新しい cost2() 関数をサポートするために、COST1 および COST2 の
2 つのモジュールを更新されたバージョンのバインダー言語 BNDUPD を使用して再作成することが必要です。再作成の前に COST を使用したプログラム COSTDPT1 は、変更されないままになります。

サービス・プログラム COST を更新するには、更新されたバージョンのバインダー言語 BNDUPD を使用
して、COST1 および COST2 の 2 つのモジュールからこれを再作成する必要があります。DETAIL パ
ラメーターで *EXTENDED オプションを指定すると、拡張出力リストが作成され、COST の現行と以前
のインターフェース識別値を確認することができます。

4. ソース・ファイル costdpt2 からプログラム COSTDPT2 を作成します。
   CRTCPPMOD MODULE(MYLIB/COSTDPT2) SRCSTMF(costdpt2.cpp)
   CRTSRVPGM SRVPGM(MYLIB/COSTDPT2) MODULE(MYLIB/COSTDPT2) BNDSRVPGM(MYLIB/COST)

プログラムの実行

IBM i コマンド行から CL コマンド CALL COSTDPT1 を使用してプログラム COSTDPT1 を実行します。

IBM i コマンド行から CL コマンド CALL COSTDPT2 を使用してプログラム COSTDPT2 を実行します。

プログラムの実行

ILE 環境でプログラムを実行する方法はいくつかあります。以下を使用できます。

- 制御言語 (CL) コマンド:
  - 呼び出し (CALL) コマンド
  - 制御権転送 (TFRCTL) コマンド
  - プログラム開発管理機能開始 (STRPD) コマンド
  - ユーザー定義 CL コマンド
- コマンド処理プログラム (CPP) としての ILE C/C++ プログラム
- 高水準言語 CALL ステートメント

注: 8-45 ページの『ILE C/C++ 呼び出し規則の使用』に、言語間呼び出しに関する情報が含まれいていま
す。

- ICF ファイル内の EVOKE タイトルメント
このトピックでは以下について説明します。

- **ILE C/C++ 実行時モデル**
- 活動化および活動化グループ
- 実行時間数および活動化グループ
- プログラムの呼び出し方法
- 正常および異常のプログラム終了
- 活動化グループの管理方法
- 実行時ストレージの管理方法

### ILE C/C++ 実行時モデル

ILE C/C++ 実行時モデルでは、以下のいずれかが満たされている場合、ISO C/C++ 標準セマンティクスが保証されます。

- アプリケーション内のすべてのプログラムがバインド済みプログラムの作成コマンド (CRTBNDC および CRTBNDCPP) で作成されている。
- プログラムの作成 (CRTPGM) コマンドで以下のオプションが使用されている。

表4-6. CRTPGM コマンド・オプション

<table>
<thead>
<tr>
<th>オプション</th>
<th>説明</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTGRP(*NEW)</td>
<td>作成された *PGM の各呼び出しで新規活動化グループが作成され、その活動化グループがプログラムの終了時に破棄されます。</td>
</tr>
<tr>
<td>OPTION(*NODUPPROC)</td>
<td>同じバインド済みプログラム内での重複したプロシージャー定義は許可されません。</td>
</tr>
<tr>
<td>OPTION(*NODUPVAR)</td>
<td>同じバインド済みプログラム内での重複した変数定義は許可されません。</td>
</tr>
</tbody>
</table>

- サービス・プログラムの作成 (CRTSRVPGM) コマンドで以下のオプションが使用されている。

表4-7. CRTSRVPGM コマンド・オプション

<table>
<thead>
<tr>
<th>オプション</th>
<th>説明</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTGRP(*CALLER)</td>
<td>このサービス・プログラムは呼び出されると、呼び出し側プログラムの活動化グループで活動化されます。</td>
</tr>
<tr>
<td>OPTION(*NODUPPROC)</td>
<td>同じバインド済みプログラム内での重複したプロシージャー定義は許可されません。</td>
</tr>
<tr>
<td>OPTION(*NODUPVAR)</td>
<td>同じバインド済みプログラム内での重複した変数定義は許可されません。</td>
</tr>
</tbody>
</table>

注: バインド済みプログラムの作成コマンドの呼び出しで CRTPGM パラメーターが指定されていない場合、CRTPGM パラメーターがデフォルトになります。例えば、パラメーター ACTGRP(*NEW) が CRTPGM コマンドのデフォルトであり、バインド済みプログラムの作成コマンドで使用されます。コマンド省略時値変更 (CHGCMDFT) コマンドを使用して、CRTPGM パラメーターのデフォルトを変更できます。
活動化および活動化グループ

ILE C/C++ プログラムを正常に作成した後、そのコードを実行することができます。活動化とは、ILE C/C++ プログラムまたはサービス・プログラムを実行可能にするプロセスです。ILE C/C++ プログラムが呼び出されると、システムは活動化を実行します。ILE C/C++ サービス・プログラムは呼び出されないため、それが活動化されるのは、直接のまたは間接的にそのサービスを必要とする ILE C/C++ プログラムが呼び出された時になります。

活動化および活動化グループがもたらす機能と利点は、以下のとおりです。

- 活動化グループに対して資源をスコープ宣言することで、同じジョブ内で実行中の ILE C/C++ プログラムが、他に干渉されることなく独立して実行できるようにする。同じジョブ内で実行中のプログラムとは、例えばコミットメント制御、指定変更、共有ファイルなどです。
- ILE C/C++ プログラムに対して資源をスコープ宣言する。
- ILE C/C++ プログラムまたはサービス・プログラムが必要とする静的データを固有に割り振る。
- ILE C/C++ サービス・プログラムに対するシンボリック・リンクを物理アドレスに変更する。

活動化により、ILE C/C++ プログラムまたはサービス・プログラムで使用される静的変数に必要な記憶域が割り振られると、スペースは活動化グループから割り振られます。ILE C/C++ プログラムまたはサービス・プログラムの作成時に、実行時に使用する活動化グループを指定します。

ILE C/C++ プログラムは、一度活動化されると、その活動化グループが削除されるまでは活動状態のままになります。それらのプログラムは、活動化されていても、実行中でない場合は呼び出しスタックに表示されません。

IBM i ジョブの開始時に、システムは OPM プログラム用の活動化グループを 2 つ作成します。1 つの活動化グループは、IBM i システム・コード用に予約済みです。もう 1 つの活動化グループは他のすべての OPM プログラムに使用されます。この OPM デフォルトの活動化グループを削除することはできません。それらはジョブの終了時にシステムによって削除されます。

注：OPM プログラムは、スレッド・セーブではありません。OPM プログラムは、それらがマルチスレッド・アプリケーションから呼び出される前に、ILE にマイグレーションし、スレッド・セーブにする必要があります。マルチスレッド・アプリケーションからの OPM プログラムを呼び出す必要がある場合は、OPM プログラムを実行する別のプロセスを開始してください。

ILE C/C++ プログラムの main() 関数が呼び出しスタックにない場合でも、活動化グループは存在し続けることができます。この状況は、(CRTPGM コマンドの ACTGRP オプションで名前を指定することで) ILE C/C++ プログラムが名前付き活動化グループで作成され、main() 関数が return を発行した場合に発生します。また、ILE C/C++ プロシージャーで設定されたジャンプ・バッファーを使用して、ILE C/C++ プログラムが制御境界を超えて longjmp() を実行する際にも、この状況が発生する可能性があります。このプロシージャーは、呼び出しスタック内の高い位置と、最も近い制御境界の前にあります。

ランタイム・ライブラリー関数と活動化グループ

ILE C/C++ ランタイム・ライブラリー関数は、アプリケーションが呼び出される活動化グループでアプリケーションにバインドされます。これは次のことを意味します。

- 同じ活動化グループのすべてのプログラム活動化が、ILE C/C++ ランタイム・ライブラリーの 1 つのインスタンスを共有します。
- ILE C/C++ ランタイム環境の状態は、プログラム呼び出し境界を越えて伝搬します。
つまり、活動化グループ内の 1 つのプログラムが ILE C/C++ ランタイムの状態を変更すると、その活動化グループ内の他のすべてのプログラムが影響を受けます。例えば、同じ活動化グループ内の他のプログラムが、アプリケーションのロケール設定やマルチバイト関数のシフトイン/シフトアウト状態によって影響を受けます。

CRTPGM コマンドの ACTGRP パラメータに *NEW 以外の値が設定されると、アプリケーションのランタイムの動作が、ISO C または ISO C++ の規格に準拠しない場合があります。ISO 非準拠の動作は、以下の間に発生する可能性があります。

- プログラムの終了 (exit()、abort()、atexit())
- シグナルの処理 (signal()、raise())
- マルチバイト・ストリーミングの処理 (mblen())
- ロケールに依存するライブラリ関数 (isalpha()、qsort())

デフォルトの活動化グループで、入出力ファイルは自動的にクローズされません。入出力バッファーはフラッシュされません。

ACTGRP が *CALLER に設定されている場合、ILE C/C++ プログラムの複数の呼び出しが、同じ活動化グループ内の ILE C/C++ ランタイム・ライブラリ状態の 1 つのインスタンスを共有します。このオプションにより、ILE C/C++ プログラムを OPM デフォルトの活動化グループ内で実行することができます。OPM デフォルトの活動化グループ内で実行される ILE C/C++ プログラムには、いくつかの制約事項があります。例えば、OPM デフォルトの活動化グループ内で atexit() 関数を登録することはできません。

名前の付いた活動化グループの場合:

- 同じジョブ内のこの活動化グループでのプログラムの呼び出しはすべて、ILE C/C++ ランタイム・ライブラリ状態の同じインスタンスを共有します。
- プログラム内のコンストラクター、デストラクター、および静的初期化は実行されません。

注: 活動化グループの作成時にのみ、コンストラクター、デストラクター、および静的初期化が実行されます。

ACTGRP(*NEW) 以外のオプションでプログラムを作成した ISO 準拠アプリケーションを作成することが可能です。

注: 資源の共有、および活動化グループ内のすべてのプログラムでのランタイム状態が ISO 非準拠の動作にならないようにするには、アプリケーション設計者の責任です。

プログラムの呼び出し

ユーザーがプログラムを呼び出すと、IBM i システムはそれに対応する実行可能コードを見つけて、そのプログラムにある命令を実行します。

注: プログラムのみが、単独で実行できます。サービス・プログラムまたはその他のバインド済みプロシージャーはそのサービスを必要とするプログラムから呼び出す必要があります。

プログラムを呼び出すには、次のような方法があります。

- 呼び出し (CALL) コマンドを使用する
- 制御権転送 (TFRCTL) コマンドを使用する
- CL コマンドを作成してプログラムを呼び出す
呼び出し (CALL) コマンドの使用

呼び出し (CALL) コマンドを使用して、プログラムを対話式に、またはバッチ・ジョブの一環として実行することができます。

このコマンドの構文は以下のとおりです。

```
CALL PGM{(library-name/program-name)}
```

例えば、次のコマンドを考えます。

```
CALL PGM(MYLIB/MYPROG)
```

これはライブラリー MYLIB 内にあるプログラム MYPROG を呼び出します。

program-name によって指定されたプログラム・オブジェクトがライブラリー・リストに含まれるライブラリー内に存在する場合は、コマンド内のライブラリーネームを省略することができます。構文は次のようになります。

```
CALL PGM(program-name)
```

例えば、MYLIB がライブラリー・リストに表示されている場合、次のように入力するだけです。

```
CALL MYPROG
```

注: コマンド・パラメーター用のプロンプトが必要な場合には、CALL を入力し F4 (プロンプト) キーを押してください。コマンドの使用方法についてヘルプが必要な場合には、CALL を入力し F1 (ヘルプ) を押してください。

呼び出し先プログラムへのパラメーターの受け渡し: 呼び出しコマンドでプロンプトを要求すると、呼び出しているプログラムにパラメーターを指定できる画面が表示されます。

また、呼び出しコマンドに続けてパラメーターを直接コマンド行に入力することも可能です。

プログラムに必要なパラメーターが 1 つだけの場合は、以下のように入力します。

```
CALL MYPROG 'parameter 1'
```

プログラムに必要なパラメーターが複数ある場合は、PARM キーワードを使用する必要があります。以下に例を示します。

```
CALL MYPROG PARM('parameter 1' parameter 2')
```

例 1: 以下の例は、実行時にパラメーターを必要とする ILE C/C++ プログラム T1520REP を示しています。

1. ソース・コードがライブラリー QCPPLE のファイル QACSRC にメンバー T1520REP として保管されると想定してください。プログラム T1520REP を作成するには、次のように入力します。

```
CRTBNDC PGM(MYLIB/T1520REP) SRCFILE(QCPPLE/QACSRC)
```

このソース・コードは、4-43 ページの図 4-25 に示されています。

2. プログラム T1520REP を実行するには、次のように入力します。

```
CALL PGM(MYLIB/T1520REP) PARM('Hello, World')
```

出力は次のとおりです。

4-42 IBM i: ILE C/C++ プログラマーの手引き
実行キーボードを押して端末セッションを終了してください。

次の図に、プログラム T1520REP のソース・ファイルを示します。

```c
/* Print out the command line arguments. */
#include <stdio.h>
void main ( int argc, char *argv[] )
{ int i;
  for ( i = 1; i < argc; ++i )
    printf("%s\n", argv[i]);
}
```

図 4-25 T1520REP — バラメーターを ILE C プログラムに渡す ILE C ソース

例 2: 以下の例では、値 'Hello, World' を、実行時にバラメーターを必要とするプログラム XRUN1 に渡す方法について説明しています。

以下のステップに従って、プログラム XRUN1 を作成し、実行してください。

1. 上記に示したソースを、デフォルトのコンパイラ・オプションでコンパイルします。コマンド行から、次のように入力します。
   CRTBNDCPP PGM(MYLIB/XRUN1) SRCSTMF('xrun1.cpp')
   結果のプログラム・オブジェクトがデフォルト・ライブラリー (この例では MYLIB) 内に作成されます。
2. コマンド行からプログラムを実行するには、次のように入力します。
   CALL PGM(MYLIB/XRUN1) PARM('Hello, World')
   プログラム XRUN1 の出力は、次のようになります。
   Hello, World
   実行キーボードを押して端末セッションを終了してください。

次の図に、プログラム XRUN1 のソース・ファイル xrun1.cpp を示します。

```c
// xrun1.cpp
// Prints out command line arguments.
#include <iostream.h>
int main ( int argc, char *argv[] )
{ int i;
  for ( i = 1; i < argc; ++i )
    cout << argv[i] << endl;
}
```

図 4-26 値 'Hello, World' を別のプログラムに渡すプログラムのソース・ファイル

呼び出し (CALL) コマンドのバラメーター変換: CL コマンド行からプログラムを呼び出す際、呼び出しコマンドで渡すバラメーターは、ユーザーがバラメーターを記述する方法に応じて変更されます。4-44 ページの表 4-8 は、バラメーターがどのように変換されるかを示したものです。
表4-8 呼び出し (CALL) コマンドのパラメーター変換

<table>
<thead>
<tr>
<th>変換ルール</th>
<th>例</th>
<th>変換結果</th>
</tr>
</thead>
<tbody>
<tr>
<td>ストリング・リテラルがヌル終了文字と共に渡される。</td>
<td>CALL PGM(T1520REP) PARM(abc)</td>
<td>ABC\0 (大文字に変換、ストリングとして受け渡し)</td>
</tr>
<tr>
<td>数値定数がバック 10 近数として渡される。</td>
<td>CALL PGM(T1520REP) PARM('123.4')</td>
<td>123.4 (バック 10 近数として渡される (15.5))</td>
</tr>
<tr>
<td>単一引用符で囲まれていない文字は、</td>
<td>CALL PGM(T1520REP) PARM(123.4)</td>
<td>123.4\0 (ストリングとして受け渡し)</td>
</tr>
<tr>
<td>・大文字変換される</td>
<td></td>
<td></td>
</tr>
<tr>
<td>・ヌル文字と共に渡される</td>
<td></td>
<td></td>
</tr>
<tr>
<td>単一引用符で囲まれた文字は、変更されない。大小文字混合ストリングはサポートされており、ヌル終了文字と共に渡される。</td>
<td>CALL PGM(T1520REP) PARM(’abc’) および CALL PGM(T1520REP) PARM(’abC’)</td>
<td>abc\0 (ストリングとして受け渡し) および abC\0 (ストリングとして受け渡し)</td>
</tr>
</tbody>
</table>

REXX インタープリターは、すべての REXX 変数をストリング（ヌル終了符なし）として扱います。
REXX はパラメーターを IBM i に渡します。次に IBM i が ILE C/C++ プログラムを呼び出します。バック 10 進データ型への変換はその後も行われ、ストリングはヌル文字で終了します。

注: これらの変更が適用されるのはコマンド行からのプログラム呼び出しのみで、言語間呼び出しには適用されません。ILE C/C++ 呼び出し規則については、8-45 ページの『ILE C/C++ 呼び出し規則の使用』を参照してください。

プロセス・コマンド (QCAPCMD) API の使用: プロセス・コマンド (QCAPCMD) API を使用すると、以下のことが可能です。
・ILE C/C++ プログラムに渡す引数にヌル文字を追加する。
・コマンド・ストリングを実行する前にその構文をチェックし、コマンドをブロントして変更されたコマンド・ストリングを受け取り、HLL (高水準言語) からコマンド実行する。

コマンド・ストリングでコマンド分析プログラムの処理を実行する際は、QCAPCMD API を使用します。
HLL から CL コマンドをチェックまたは実行することも、特定のソース定義タイプの構文をチェックすることもできます。

制御権転送 (TFRCTL) コマンドの使用
制御権転送 (TFRCTL) コマンドを使用して制御権をユーザーのプログラムに転送する制御言語プログラムの中から、アプリケーションを実行することができます。このコマンドは、以下を行います。
1. コマンドで指定されたプログラムに制御権を転送します。
2. 転送元の制御言語プログラムを呼び出しスタックから除去します。

以下の例では、制御言語プログラム RUNCP 内の TFRCTL コマンドが、TFRCTL コマンドで指定された C++ プログラム XRUN2 を呼び出します。RUNCP は XRUN2 に制御権を転送します。転送元のプログラム RUNCP は呼び出しスタックから除去されます。

4-45 ページの図 4-27 は、制御言語プログラム RUNCP の呼び出しと、C++ プログラム XRUN2 への制御権の転送を示しています。
例: TFRCTL コマンドを使用したプログラムの作成および実行: プログラム RUNCP および XRUN2 を作成および実行するには、以下のステップに従います。
1. ソース・ファイル QCLSRC を作成し、図4-29 に示されているソース・コードを入力します。
2. 制御言語プログラム RUNCP を作成します。コマンド行から、次のように入力します。
   CRTCLPGM PGM(MYLIB/RUNCP) SRCFILE(MYLIB/QCLSRC)
3. ソース・ファイル xrun2.cpp (4-46 ページの図4-29 を参照) からライブラリー MYLIB でプログラム XRUN2 を作成するには、以下を入力します。
   CRTBNDCPP PGM(MYLIB/XRUN2) SRCSTMF(xrun2.cpp)
4. コマンド行から、以下のコマンドでストリング「nails」を渡し、プログラム RUNCP を実行します。
   CALL PGM(MYLIB/RUNCP) PARM('nails')

プログラム XRUN2 からの出力は、以下のとおりです。

```
string = nails
実行キーを押して端末セッションを終了してください。
```

コード・サンプル:

```c
/* Source for CL Program RUNCP */
PGM PARM(&STRING)
DCL VAR(&STRING) TYPE(*CHAR) LEN(20)
DCL VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')

/* ADD NULL TERMINATOR FOR THE ILE C++ PROGRAM */
CHGVAR VAR(&STRING) VALUE(&STRING *TCAT &NULL)
TFRCTL PGM(MYLIB/XRUN2) PARM(&STRING)

/* THE DSPJOBLOG COMMAND IS NOT CARRIED OUT SINCE */
/* WHEN PROGRAM XRRUN2 RETURNS, IT DOES NOT RETURN TO THIS */
/* CL PROGRAM. */
DSPJOBLOG
ENDPGM
```

図4-27. TFRCTL コマンドを使用したプログラム XRUN2 の呼び出し

図4-28. 別のプログラムに制御権を転送するソース・コードの例
注: 例「4-45 ページの『例: TFRCTL コマンドを使用したプログラムの作成および実行』」では、プログラム RUNCP は TFRCTL コマンドを使用して ILE C++ プログラム XRUN2 に制御権を渡します。そのプログラムは、RUNCP に制御権を戻しません。

// xrun2.cpp
// Source for Program XRUN2
// Receives and prints a null-terminated character string

#include <iostream.h>
int main(int argc, char *argv[])
{
    int i;
    char *string;
    string = argv[1];
    cout << "string=" << string << endl;
}

図4-29. ヌル終了文字ストリングを受け取り、印刷するソース・コードの例

注: 例「4-45 ページの『例: TFRCTL コマンドを使用したプログラムの作成および実行』」では、プログラ

プログラムを実行する CL コマンドの作成

独自の CL コマンドからプログラムを実行することも可能です。コマンドを作成するには、次のようにし

1. 一連のコマンド・ステートメントをソース・ファイルに入力します。
2. コマンド作成 (CRTCMD) コマンドを使用し、ソース・ファイルを処理してコマンド・オブジェクト

   CRTCMD のコマンド定義は、コマンド名、パラメーター記述、および妥当性検査情報を含み、コマンド

3. コマンドを対話式に、またはバッチ・ジョブで入力します。

   コマンドで呼び出したプログラムが実行されます。

以下の例では、ユーザーが作成したコマンドからプログラムを実行する方法を示しています。

プログラムの説明: 新しく作成されたコマンド COST では、ユーザー入力値のプロンプトを出し、それを受け入れます。その後、C++ プログラム CALCOST を呼び出し、入力値を受け渡します。CALCOST はコマンド COST から入力値を受け入れ、これらの値で計算を実行し、結果を印刷します。4-47 ページの図4-30は、この例を示しています。
図4-30. ユーザ定義コマンド COST からのプログラム CALCOST の呼び出し

手順: 例を作成して実行するには、以下のステップに従ってください。

1. コマンド・プロンプト COST (4-48 ページの図 4-31 に示しています) のソース・コードをライブラリ MYLIB 内のソース・ファイル QCMDSRC に入力し、メンバー COST として保存します。

2. コマンド・プロンプト COST を作成するには、以下のようにします。コマンド行から、次のように入力します。

   CRTCMD CMD(MYLIB/COST) PGM(MYLIB/CALCOST) SRCFILE(MYLIB/QCMDSRC)

3. ソース・ファイル calcost.cpp (4-48 ページの図 4-31 に示しています) からプログラム CALCOST を作成するには、以下を入力します。

   CRTBNDCPP PGM(MYLIB/CALCOST)

4. プログラム CALCOST を実行するには、以下のようにします。

   a. COST と入力して、F4（プロンプト）を押します。プロンプト ITEM、PRICE、および QUANTITY が順に表示されます。

   b. プロンプトが出されたら、以下に示したデータを入力します。

   Hammers
   1.98
   5000

プログラム CALCOST の出力は、以下のようにになります。

   It costs $11385.00 to buy 5000 HAMMERS
   実行キーを押して端末セッションを終了してください。

コード・サンプル:
/* Source for Command Prompt COST */

CMD   PROMPT('CALCULATE TOTAL COST')

PARM  KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
       MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)

PARM  KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
       RANGE(0.01 99999999.99) MIN(1) +
       ALWUNPRT(*YES) PROMPT('Unit price' 2)

PARM  KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
       9999) MIN(1) ALWUNPRT(*YES) +
       PROMPT('Number of items' 3)

図4-31. CALCOST プログラムを実行するコマンド・プロンプトのソース・コード

// calcost.cpp
// Source for Program CALCOST

#include <iostream.h>
#include <string.h>
#include <bcd.h>

int main(int argc, char *argv[])
{
  char *item_name;
  _DecimalT<10,2> *price;
  short int *quantity;
  const _DecimalT<2,2> taxrate=_D("0.15");
  _DecimalT<17,2> cost;
  item_name = argv[1];
  price = (_DecimalT<10,2> *) argv[2];
  quantity = (short *) argv[3];
  cost = (*quantity)*(*price)*(_D(1.00+taxrate));
  cout << "It costs $" << cost << " to buy "
       << *quantity << " " << item_name << endl;
}

図4-32. プログラム CALCOST のソース・コード

注: このプログラムは、コマンド COST から入力引数を受け取り、コストを計算して、値を印刷します。すべての入力引数はポインタです。

正常および異常のプログラム終了

プログラムが正常に終了すると、システムは、制御を呼び出し元に戻します。呼び出し元は、ワークステーション・ユーザーまたは別のプログラムである場合があります。

プログラムが実行時に異常終了し、プログラムが呼び出し元とは異なる活動化グループで実行されていた場合、以下のエスケープ・メッセージ CEE9301 が発行され、制御が呼び出し元に返されます。

アプリケーション・エラー、<msgid> は、<pmm> によってステートメント
<stmtid> 命令 <instruction> に監視されていません。

制御言語プログラムは、メッセージ・モニター (MONMSG) コマンドを使用して、この例外をモニターすることができます。

プログラムとその呼び出し元が同じ活動化グループで実行されている間、プログラムが異常終了した場合、発行されるメッセージは、プログラムがどのように終了したかによって異なります。機能チェックで終了した場合には、CPF9999 が発行されます。
注：エスケープ・メッセージについて詳しくは、IBM i Information Center の Web サイト（http://www.ibm.com/systems/i/infocenter）の『Programming』カテゴリーの『APIs』セクションにある『メッセージの処理 API』トピック内の『Message Handling Terms and Concepts』を参照してください。

活動化グループの管理
活動化グループにより、複数の ILE プログラムと同じジョブで独立して、互いに阻害することなく、実行することが可能になります。

活動化グループは、ジョブの副構造です。ストレージ、コメントメント定義、および開いているファイルなどのシステム資源で構成されます。これらの資源が、1 つ以上の ILE または OPM プログラムを実行するために割り振られます。例えば、プログラムの静的変数のストレージ・スペースが、活動化グループから割り振られます。

プログラム (タイプ *PGM) は呼び出されると、それが実行されている活動化グループが削除されるまで活動化されたままになります。サービス・プログラムは直接呼び出されないため、サービスを必要とするプログラムが呼び出されている間、活動状態となります。

活動化グループの指定
IBM i ジョブの開始時に、システムは OPM プログラムによって使用される活動化グループを 2 つ、自動的に作成します。1 つの活動化グループは、IBM i システム・コード用に予約済みです。もう 1 つの活動化グループは他のすべての OPM プログラムに使用されます。この活動化グループを表すのに使用される記号は、*DFTACTGRP です。この OPM デフォルトの活動化グループを削除することはできません。それらはジョブの終了時にシステムによって削除されます。

注: OPM プログラムは常にデフォルトの値の活動化グループで実行します。これらの活動化グループの仕様書を変更することはできません。

ILE プログラムについては、「プログラムの作成」コマンドまたは「サービス・プログラムの作成」コマンドの ACTGRP パラメータで、実行時に使用する活動化グループを指定します。以下の中から選択できます。

- 名前付き活動化グループでプログラムを実行する。
- デフォルトの活動化グループを受け入れる。
  - *NEW (プログラムの場合)
  - *CALLER (サービス・プログラムの場合)
- 呼び出し側プログラムの活動化グループに対してプログラムを活動化する。

名前の付いた活動化グループでのプログラムの実行: ILE プログラムおよびサービス・プログラムのコレクションを 1 つアプリケーションとして管理するには、ACTGRP パラメータにユーザーディレクトイ名前を指定して、名前の付いた活動化グループを作成します。

名前の付いた活動化グループを指定した最初のプログラムが呼び出されるとすぐに、システムによってこの活動化グループが作成されます。その後、このグループは、この名前が指定されたすべてのプログラムおよびサービス・プログラムによって使用されます。

名前の付いた活動化グループは、活動化グループの再利用 (RCLACTGRP) コマンドで削除されると終了します。このコマンドは活動化グループがもう使用されない場合にのみ使用することができます。また、コードで exit() 関数を呼び出したときにも終了します。
名前の付いた活動化グループが終了すると、グループのプログラムおよびサービス・プログラムに関連したすべての資源がシステムに返されます。

注：名前の付いた活動化グループを使用すると、非 ISO 準拠の実行時の動作になることがあります。名前の付いた活動化グループを使用して作成されたプログラムが return ステートメントによって活動化されたままになっている場合、以下の問題が生じます。

・ 静的変数が再初期化されない。
・ 静的コンストラクターが再び呼び出されない。
・ 静的デストラクターが戻り時に呼び出されない。
・ 独立しているように見えるが、同じ活動化グループで活動化されている他のプログラムが当該プログラムを終了することがある。
・ 名前の付いた活動化グループの動作に依存している場合、当該プログラムが移植不可になる。

図4-33 名前の付いた活動化グループでのプログラムの実行

以下の例では、プログラム PROG1、PROG2、および PROG3 は同じアプリケーションの一部であり、同じ活動化グループ GROUP1 で実行されます。図4-33 に、このシナリオを示します。

同じ活動化グループでこれらのグループを作成するには、各プログラムの作成時に ACTGRP パラメーターで GROUP1 を指定します。

CRTCPMOD MODULE(PROG1) SRCSTMF(prog1.cpp)
CRTPGM PGM(PROG1) MODULE(PROG1) ACTGRP(GROUP1)
CRTCPMOD MODULE(PROG2) SRCSTMF(prog2.cpp)
CRTPGM PGM(PROG2) MODULE(PROG2) ACTGRP(GROUP1)
CRTCPMOD MODULE(PROG3) SRCSTMF(prog3.cpp)
CRTPGM PGM(PROG3) MODULE(PROG3) ACTGRP(GROUP1)

活 動 化 グ ル プ *NEW でのプログラムの実行：プログラムが呼び出されるたびに新規活動化グループを作成するには、ACTGRP パラメーターで *NEW を指定します。この場合、システムにより、活動化グループに対してジョブ内で固有の名前が作成されます。

*NEW は、CRTPGM コマンドの ACTGRP パラメーターのデフォルト値です。*NEW で作成された活動化グループは常に、それに関連付けられた最後のプログラムが終了すると終了します。

注：*NEW は、サービス・プログラムでは無効です。サービス・プログラムは、呼び出し元の活動化グループまたは指定した活動化グループでのみ実行できます。
ACTGRP(*NEW) を指定してプログラムを作成した場合、複数のユーザーが同じ活動化グループを使用せずに、そのプログラムを同時に呼び出すことができます。各呼び出して、プログラムの新規コピーが使用されます。新規コピーはそれぞれ、独自のデータを持ち、ファイルを開きます。

以下の例では、プログラム PROG4、PROG5、および PROG6 が名前のない別個の活動化グループで実行されます。

![図 4.34 名前のない活動化グループでのプログラムの実行](image)

デフォルトでは、各プログラムは、ACTGRP パラメーター (*NEW) によって指定された、異なる活動化グループに作成されます。

- CRTCPMOD MODULE(PROG4) SRCSTMF(prog4.cpp)
- CRTPGM PGM(PROG4) MODULE(PROG4) ACTGRP(*NEW)
- CRTCPMOD MODULE(PROG5) SRCSTMF(prog5.cpp)
- CRTPGM PGM(PROG5) MODULE(PROG5) ACTGRP(*NEW)
- CRTCPMOD MODULE(PROG6) SRCSTMF(prog6.cpp)
- CRTPGM PGM(PROG6) MODULE(PROG6) ACTGRP(*NEW)

*NEW がデフォルトであるため、以下の呼び出しでも同じ結果が得られます。

- CRTBNDCPP PGM(PROG4) SRCSTMF(prog4.cpp)
- CRTBNDCPP PGM(PROG5) SRCSTMF(prog5.cpp)
- CRTBNDCPP PGM(PROG6) SRCSTMF(prog6.cpp)

注: 以下のように、3 つのモジュールを単一のコマンドで呼び出した場合、単一のプログラム・オブジェクト PROG が活動化グループ *NEW に作成されます。

- CRTCPMOD MODULE(PROG7) SRCSTMF(prog7.cpp)
- CRTCPMOD MODULE(PROG8) SRCSTMF(prog8.cpp)
- CRTCPMOD MODULE(PROG9) SRCSTMF(prog9.cpp)
- CRTPGM PGM(PROG) MODULE(PROG7 PROG8 PROG9)

名前付き活動化グループによる非標準の振る舞い: CRTPGM コマンドの ACTGRP パラメーターが *NEW 以外の値として指定されている場合、アプリケーションの実行時の振る舞いが ISO 言語規則に従わないことがあります。実行時ライブラリおよびクラス・ライブラリでは、プログラムが ACTGRP(*NEW) によってビルトされていることが前提となります。

ISO 非準拠の振る舞いは、以下の間に発生する可能性があります。

- プログラム終了処理 - exit(), abort(), atexit()
- シグナル処理 - signal(), raise()
- マルチバイト・ストリング処理 - mblen()
- いずれかのロケール依存ライブラリー関数 - isalpha(), qsort()
デフォルトの活動化グループで、入出力ファイルは自動的にクローズされません。入出力バッファーは、明示的に要求しない限り、フラッシュされません。

活動化グループでのプログラムの実行 (*CALLER): ACTGRP を *CALLER に設定することで、呼び出し側プログラムの活動化グループ内で ILE プログラムまたは ILE サービス・プログラムが活動化されるように指定することができます。この属性を指定すると、プログラムまたはサービス・プログラムが活動化される時点で、新規活動化グループは作成されません。このオプションでは、呼び出し元が OPM プログラムの場合、ILE C/C++ プログラムを OPM デフォルトの活動化グループ内で実行することができます。

OPM デフォルトの活動化グループで実行中の ILE C/C++ プログラムには、一定の制約事項があります。例えば、OPM デフォルトの活動化グループ内で atexit() 関数を登録することはできません。

活動化グループが名前付きである場合、同じジョブ内のこの活動化グループのプログラムに対する呼び出しはすべて、ILE C/C++ ランタイム・ライブラリ状態の同じインスタンスを共有します。

プログラムが ACTGRP(*NEW) 以外のオプションで作成されている ISO 対応のアプリケーションを作成することは可能です。非 ISO の振る舞いが望ましい場合もありますが、アプリケーションの設計者はその責任において、活動化グループ内のすべてのプログラムで資源と実行時の状態を共有することで誤った振る舞いが起こらないようにする必要があります。

以下の例では、サービス・プログラム SRV1 がプログラム PROG7 および PROG8 の各活動化グループに対して活動化されます。PROG7 は名前付きの活動化グループ GROUP2 で実行され、PROG8 は名前のない活動化グループ *NEW で実行されます。

![Activity Group Diagram](image)

図4-35 呼び出し側プログラムの活動化グループ内でのサービス・プログラムの実行

デフォルトでは、サービス・プログラム SRV1 が各呼び出し側プログラムの活動化グループ内に作成されます。

CRTCRRMOD MODULE(SRV1) SRCSTMF(srv1.cpp)
CRTCRRPGM SRVPGM(SRV1) MODULE(SRV1)

呼び出しスタック上のプログラムの存在

プログラムは活動化されても、実行中でない限り呼び出しスタックに出現しません。しかし、活動化グループは、プログラムの main() 関数が呼び出しスタックにない場合でも、存在し続けることができます。

これは、プログラムが名前の付いた活動化グループで作成され、main() 関数が return を実行したときに起こります。また、プログラムが、ILE C または C++ のブラシッシャー（このブラシッシャーは、呼び出しスタックで上位にあり、最も近い制御境界の前にある）で設定されたジャンプ・バッファーを使用することで、制御境界をまたがって longjmp() を実行したときに起こる可能性があります。
活動化グループの削除
活動化グループが削除されると、その資源が再利用されます。この資源には静的記憶域およびオープン・ファイルが含まれます。NEW 活動化グループは、関連付けられているプログラムが呼び出し元に戻ったときに削除されます。

名前の付いた活動化グループはパーシスタントです。明示的に削除する必要があります。そうしないと、ジョブの終了時にのみ終了することになります。名前の付いた活動化グループで実行中のプログラムと関連した記憶域は、これらの活動化グループが削除されるまで解放されません。

OPM デフォルト活動化グループもパーシスタント活動化グループです。デフォルト活動化グループで実行されている ILE プログラムに関連付けられているストレージは、サインオフ時（対話式ジョブの場合）またはジョブの終了時（パッチ・ジョブの場合）に解放されます。

システム・リソースの再利用
以下のような場合には、システム記憶域がすべて使い尽くされた状態になる可能性があります。

- 多数の ILE プログラムが活動状態になっている（つまり少なくとも 1 回呼び出されている）場合。
- OPM デフォルトの活動化グループで、多量の静的記憶域を使用する ILE プログラムが実行されている場合（ジョブが終了するまで記憶域は再利用されない）。
- 多数のサービス・プログラムが名前付き活動化グループに呼び出された場合（リソースが再利用されるのはジョブが終了するときのみ）。

このような状況において、プログラムでは必要でなくなったりも活動化グループが削除されていないため、確実なままとなっているシステム・リソースを再利用したい場合があります。次のオプションを選択できます。
- 活動化グループの再利用 (RCLACTGRP) コマンドを使用して、使用中でない名前付き活動化グループを削除してください。

このコマンドには該当するすべての活動化グループを削除するか、あるいはある名前の活動化グループ 1 つを削除するかのオプションがあります。
- 資源再利用 (RCLRSC) コマンドを使用して、活動状態でなくなったりプログラム用のリソースを解放してください。

資源再利用 (RCLRSC) コマンドの使用: RCLRSC コマンドの動作は、プログラムの作成方法によって次のように異なります。

- OPM プログラムの場合、RCLRSC コマンドはオープンしているファイルをクローズし、静的ストレージを解放します。
- (*CALLER で作成されたために) 活動化されて OPM デフォルト活動化グループに入った ILE プログラムの場合、RCLRSC コマンドは、ファイルをクローズしてストレージを再初期化します。ただし、ストレージは解放されません。
- 名前の付いた活動化グループと関連を持つ ILE プログラムの場合、RCLRSC コマンドは何も作用しません。名前の付いた活動化グループの資源を解放するためには、RCLACTGRP コマンドを使用しなければなりません。

実行時のストレージの管理
ILE では、ヒープを管理することで、プログラムから直接、実行時のストレージを管理できます。ヒープは動的ストレージの割り振りに使用されるストレージ域のことです。アプリケーションに必要な動的記憶域
の量は、ヒープを使用するプログラムおよびプロシージャーによって処理されるデータによって異なります。ILE バインド可能 API を使用して、ヒープを管理します。

実行時のストレージを明示的に管理する必要はありません。ただし、動的に割り振られたストレージを使用する場合には、そのようにすることができます。例えば、配列の大きさが正確に分からない場合、プログラムでその配列の大きさを判別してから、実行時に配列の実際のストレージを獲得できます。

システムでは、以下の 2 つのタイプのヒープを使用することができます。

・ デフォルト・ヒープ
・ ユーザー作成ヒープ

1 つ以上のユーザー作成ヒープを使用して、活性化グループ内の一部のプログラムおよびプロシージャーで必要とされる動的ストレージを分離できます。

このセクションの残りの部分では、プログラムでデフォルト・ヒープを使用して実行時のストレージを管理する方法について説明します。

**デフォルト・ヒープの管理**

活動化グループ内で動的記憶域の最初の要求があると、デフォルト・ヒープが作成され、ここから記憶域割り振りが行われます。動的記憶域の追加の要求は、デフォルト・ヒープからの割り振りによって満たされます。動的記憶域の現行要求を満たすための記憶域がヒープに不足している場合には、ヒープが拡張され、追加の記憶域が割り振られます。

割り振られた動的記憶域は、明示的に解放するか、あるいはヒープが破棄されるまで割り振られたままです。デフォルト・ヒープが破棄されるのは、所有する活性化グループが終了したときだけです。

同一の活性化グループのプログラムはすべて、同じデフォルト・ヒープを使用します。あるプログラムが割り振られた以上の記憶域にアクセスした場合には、別のプログラムに問題を起こすことがあります。

例えば、2 つのプログラム、PGM1 と PGM2 が同じ活性化グループで実行されているものとします。10 バイトが PGM1 に割り振られても 11 バイトがそれによって変更されます。追加バイトが PGM2 に実際に割り振られた場合、PGM2 で問題が生じることがあります。

バインド可能 API を使用したデフォルト・ヒープの管理: デフォルト・ヒープでは次の ILE バインド可能 API を使用することができます。

**ストレージの解放 (CEEFRST)**

ヒープ記憶域の 1 つ前の割り振りを解放します。

**ヒープ記憶域取得 (CEEGTST)**

ヒープ内の記憶域を割り振ります。

**ストレージ再割り振り (CEECZST)**

直前に割り振られた記憶域のサイズを変更します。

実行時における動的ストレージ割り振り:

**C++**

ILE C++ プログラムでは、演算子 new および delete を使用して動的オブジェクトを作成および削除することで、デフォルト・ヒープに属している動的ストレージを管理します。動的オブジェクトは自動的には作成および削除されません。十分なフリー・ヒープ・スペースが使用可能でない場合は、作成は失敗することがあり、プログラムではこの可能性に備える必要があります。
以下の図では、動的ストレージ割り振りのために new および delete 演算子を使用する方法を示します。

```cpp
TClass *p;  // Define pointer
p = new TClass;  // Construct object
if (!p) {
    Error("Unable to construct object");
    exit(1);
}
...  // Define pointer
delete p;  // Delete object
```

図4.36. クラス・オブジェクトのストレージの動的割り振りおよび割り振り解除の例

```cpp
TClass *array;  // Define pointer
array = new TClass[100];  // Construct array of 100 objects
if (!array) {
    Error("Unable to construct array");
    exit(1);
}
...  // Define pointer
delete[] array;  // Delete array
```

注: この例では、delete[] を使用して配列を削除します。ブロックがない場合、delete は配列全体を削除しますが、配列内の最初のエレメントに対してのみデストラクターを呼び出します。デストラクターがない値の配列がある場合は、delete または delete[] を使用できます。

図4.37. オブジェクトの配列のストレージの動的割り振りおよび割り振り解除の例

置換関数のオーバーライド:

> C++

C++ 标準では、多くの置換関数をアプリケーションで再定義することができます。ライブラリーによって提供されるデフォルト・バージョンではなく、プログラムの定義が使用されます。こうした置換は、プログラム開始処理の前に実行されます。

C++ プログラムは、8 つの動的メモリー割り振り関数のいずれかに定義を提供します。以下の関数です。

- void *operator new (size_t) throw(std::bad_alloc);
- void *operator new (size_t, const std::nothrow_t&) throw();
- void *operator new[] (size_t) throw(std::bad_alloc);
- void *operator new[] (size_t, const std::nothrow_t&) throw();
- void operator delete (void*) throw();
- void operator delete (void*, const std::nothrow_t&) throw();
- void operator delete [] (void*) throw();
- void operator delete [] (void*, const std::nothrow_t&) throw();

制限: 置換関数をオーバーライドする際には、次の制約に注意してください。

- main 関数を含むプログラムは、C++ コンパイラとコンパイルする必要があります。
- main() エントリー・ポイントが C++ モジュールでない場合、グローバルな new または delete 演算子の呼び出しは、それらが同じコンパイル単位にある（対応する置換関数の定義が可視である）場合にのみ機能します。
- 置換関数の実装で標準ライブラリー・オブジェクトを使用すると、無限再帰が生じることがあります。ライブラリーが割り振り演算子の呼び出しの使用を拡張するからです。
新またはdelete演算子の多重定義：ISO C++ 標準は、演算子 new および演算子 delete を置換関数として分類しています。つまり、これらは、C++ プログラムで再定義できます。ただし、標準では、プログラム実行時に有効にすることができる演算子の定義は 1 つだけです。

注：プログラムが以下の方を行うと、可視性の問題が生じることがあります。
・ 演算子 new または演算子 delete を多重定義する
・ 複数の C++ 変換単位を使用する
可視性の問題について詳しくは、「ILE C/C++ 解説書」を参照してください。

例：

アプリケーションで以下 3 つの C++ ソース・ファイル (one.cpp、two.cpp、および three.cpp) を使用するものとします。
・ one.cpp には、main 関数が含まれています。
・ two.cpp には、演算子 new または演算子 delete の再定義が含まれています。
・ three.cpp は演算子 new を呼び出します。
アプリケーションをコンパイルした後に、再定義された演算子 new(または演算子 delete) は、すべての変換単位に対して可視であり、すべての変換単位で使用されます。

同じ 3 つの変換単位のシナリオで、one.cpp が C コンパイラでコンパイルされるものとします。再定義された演算子は、変換単位 two.cpp には可視ですが、three.cpp では可視ではありません。変換単位 two.cpp の外側での演算子 new(または演算子 delete) の呼び出しはすべて、演算子のユーザー定義バージョンではなく、標準バージョンを使用することになります。

注：ユーザー定義の演算子と標準の演算子ではシンタキーが異なるため、バイナダー・エラーもコンパイラ警告も生成されません。

実行時のパフォーマンスの改善

パフォーマンスを改善するためにプログラムを調べる際には、プログラムを実行することに重大な影響を及ぼす側面を確認します。

多くの場合、実行時のパフォーマンスは、ソース・プログラムに対する小さい変更で改善できます。各変更で達成される改善の度合いは、以下によって異なります。
・ プログラムの編成
・ プログラムで使用している関数および言語構成要素
プログラムに大幅なパフォーマンスの改善をもたらす変更もあれば、ほとんど改善をもたらさない変更もあります。

注：ヒントの中には、利点のトレードオフが原因で、互いに矛盾しているものもあります。例えば、あるヒントでは、静的変数およびグローバル変数を使用して呼び出しスタックのサイズを削減しますが、別のヒントでは、静的変数およびグローバル変数の使用を削減することで、実行開始パフォーマンスを改善します。
実行時のパフォーマンスを改善しようとする前に、完全な最適化を使用して、プログラムをコンパイルしてベンチマークします。パフォーマンス分析ツールを使用して、パフォーマンス上の問題がある場所を見つけたから、該当する各種ヒントを試して適用し、試行錯誤しながらプログラムの最適なパフォーマンスが得られるようにします。

このトピックでは、以下の点についてパフォーマンスを改善する方法について説明します。

- データ・タイプ
- クラス
- パフォーマンス測定
- 例外処理
- 関数呼び出しのパフォーマンス
- 入出力の考慮事項
- Pointers
- シャロー・コピーおよびディープ・コピー
- スペースに関する考慮事項
- 活動化グループ
- コンパイラ・オプション
- 実行時の制限

パフォーマンスを改善するためのデータ・タイプの選択
データ・タイプによってパフォーマンスを改善する方法がいくつかあります。ビット・フィールドを他のデータ・タイプに置き換える方法、および静的変数およびグローバル変数の使用を最小限に抑える方法は、その例です。

volatile 修飾子の使用の回避
volatile 修飾子は、必要な場合にのみ使用してください。volatile は、変数が、おそらく外部プログラムによって、いつで変更可能であるため、最適化の候補ではないことを指定します。

ビット・フィールドを他のデータ・タイプに置換
ビット・フィールドへのアクセスには、他のデータ・タイプ (short や int など) より時間がかかるため、ビット・フィールドの使用は避けてください。ビット・フィールドは、可能であれば常に他のデータ・タイプに置き換えてください。ビット・フィールドが 16 ビットを使用し、2 パイト境界に位置合わせする場合は、short データ・タイプに置き換えることができます。

注: ビット・フィールドが整数型より小さい場合でも、実行時の向上は得られます。データでのスペース節約によるパフォーマンスの向上は、ビット・フィールドの操作コードに追加で必要な時間によって相殺されます。

静的変数およびグローバル変数の使用の最小限化
可能であれば、静的変数およびグローバル変数の使用は最小限に抑えてください。これらは、明示的に初期化するかどうかに関係なく、初期化されます。静的変数およびグローバル変数を使用しないことで、活動化グループの開始でパフォーマンスの向上が得られます。
レジスター・ストレージ・クラスの使用

頻繁に使用する変数には、レジスター・ストレージ・クラスを使用します。最適化プログラムが、最も頻繁に使用される変数を使用可能なハードウェア・レジスターに入れられるようにするために、レジスター・ストレージ・クラスは乱用しないでください。

レジスター・ストレージ・クラスを使用した場合、デバッガー内から表示される値は信頼できません。これは、ストレージ内に残っている古い値を参照している可能性があるためです。

パフォーマンスを改善するためのクラスの作成

クラス・ライブラリーを使用してクラスを作成する場合、ハイレベル抽象化を使用します。クラスへのアクセスのタイプを設定した後に、より具体的な実装を作成できます。これにより、最少のコードの変更でパフォーマンスが改善されることがあります。

クラス内で構造体またはデータ・メンバーを定義する際には、まず最大のデータ・タイプを定義して最大の自然境界を位置合わせます。まず、ポインタを定義して、4倍長ワード（16バイト）境界で位置合わせるために必要な埋め込みを削減します。その後、ダブルワード項目およびハープワード項目の順で続け、埋め込みを回避し、またロード/保管時間を改善します。

パフォーマンス測定の使用可能化

ネイティブ・コンパイラ・オプションを使用して、生成済みコードにパフォーマンス・フックを組み込むことができます。

コンパイラ・オプションを使用してパフォーマンス測定を使用可能にする

パフォーマンス測定コンパイラ・オプション ENBPFRCOL() を使用すると、コンパイラがコンパイル済みプログラムまたはモジュール内にコード（パフォーマンス・フックと呼ばれることもある）を生成するかどうかを指定することができます。パフォーマンス・フックにより、Performance Explorer でプログラムを分析することが可能になります。このオプションのデフォルトは、プログラム実行プロシージャー・レベルのパフォーマンス測定コードがコンパイル済みモジュールまたはプログラム用に生成されることを指定します。

パフォーマンス測定コードをモジュールまたはプログラムにコンパイルすることで、パフォーマンス・データを収集し、分析することができます。追加の収集コードを挿入することで、モジュールまたはプログラム・オブジェクトのサイズがわずかに大きくなるため、パフォーマンスに多少の影響が出る可能性があります。

収集されるパフォーマンス・データのタイプには、次のようなものがあります。
- 呼び出し前および呼び出し後の情報
  この情報は、任意の指定された関数を呼び出す直前および直後に収集されます。ここで提供されるのは、呼び出しが行われた場所の記録と、呼び出された命令のパフォーマンスに関する情報です。
- プロシージャー入力および出力の情報
  この情報は、プロシージャーの入力および出力の直後に収集されます。プロシージャーの入力で現在のパフォーマンス統計のスナップショットが取得され、そのプロシージャーの出口でそれらの統計の差の計算が行われます。
リーフ・プロシージャー内にパフォーマンス収集コードが生成される際、そのプロシージャーは、リーフ・プロシージャーではなくなるよう変更されます。（リーフ・プロシージャーは、他のプロシージャーを呼び出さないプロシージャーです。）これは、リーフ・プロシージャーにパフォーマンス収集ルーチンを呼び出すフックが組み込まれたためです。この処理には、時間がかかることがあります。

これらのオプションについて詳しくは、「ILE C/C++ コンパイラー参照」を参照してください。

例外処理の最小化
例外処理を最小化するために、以下を試すことができます。

- レコード入出力時の戻りコードをオフにする
- レコード入出力時の C2M メッセージをオフにする
- 直接モニター・ハンドラーを使用する
- 例外のバーコレーションを最小化する

例外処理については、8-1ページの『プログラムでの例外の処理』を参照してください。

レコード入出力での戻りコードの有効化
例外の処理にはコストがかかります。

レコード入出力を使用する場合、_Ropen() で rtncode=y オプションを使用することで、例外を最小限に抑えることができます。以下の条件の場合に例外が生成されません。

- "Record not found" (CPF5006)
- "End-of-File" (CPF5001)

これらの条件が発生した場合、_RIOFB_T 構造の num_bytes フィールドが更新され、errno が設定されますが、例外は生成されません。"Record not found" の条件の場合、num_bytes フィールドはゼロに設定されます。"End-of-File" の条件の場合、num_bytes フィールドは EOF に設定されます。

レコード入出力中に C2M メッセージをオフにする
レコード入出力中に C2M メッセージをオフにするには、変数_C2M_MSG (<recio.h> 内) をゼロに設定します。_C2M_MSG が別の値に設定されている場合、レコード入出力は、C2M3003、C2M3004、C2M3005、C2M3009、C2M3014、C2M3015、C2M3040、C2M3041、C2M3042、および C2M3044 のいずれかのエラーを検出すると、C2M メッセージをプログラムに送信します。

注：レコード入出力中に C2M メッセージをオフにする場合に、シグナル・ハンドラーおよびメッセージ・ハンドラーの付いたデータ切り捨てメッセージを削除する必要はなくなりました。

直接モニター・ハンドラーの使用
例外が発生すると、コンパイラーはまず、直接モニター・ハンドラーを使用しようとします。直接モニター・ハンドラーがない場合、例外は実行時にシグナルにマップされ、対応するシグナル・ハンドラーが呼び出されます。  pragma exception_handler ディレクティブを使用して直接モニター・ハンドラーを使用可能にすることで、シグナル・マッピングとシグナル・ハンドラーの検索の両方の処理を回避できます。

#pragma exception_handler ディレクティブによって指定されたすべての例外について、直接モニター・ハンドラーは、各例外を処理済みとマークします。そうでない場合、例外は再度バーコレートされます。

#pragma exception_handler については、「ILE C/C++ コンパイラー参照」を参照してください。
例外を処理する例

### ILE C ソース・コードの例外処理例

以下の図は、例外を処理するための ILE C ソース・コードの例を示しています。図の下には、発生する可能性のある例外および例外を処理するためにコードで実行されるステップの例があります。

```c
#include <stdio.h>
#include <except.h>
#include <signal.h>
#include <lecond.h>

void handler1(_INTRPT_Hndlr_Parms_T * __ptr128 parms) {
    printf("In handler1: will not handle the exception\n");
}
void handler2(_INTRPT_Hndlr_Parms_T * __ptr128 parms) {
    printf("In handler2: will not handle the exception\n");
}
void handler3(_FEEDBACK *condition, _POINTER *token, _INT4 *result_code,
              _FEEDBACK *new_condition) {
    printf("In handler3: will not handle the exception\n");
}
void handler4(_INTRPT_Hndlr_Parms_T * __ptr128 parms) {
    printf("In handler4: will not handle the exception\n");
}

void fred(void) {
    _HDLR_ENTRY hdlr = handler3;
    char *p = NULL;
    #pragma exception_handler(handler2, 0, 0, \
        _C2_MH_ESCAPE | _C2_MH_FUNCTION_CHECK)
    CEEHDLR(&hdlr, NULL, NULL);  
    #pragma exception_handler(handler1, 0, 0, _C2_MH_ESCAPE)
    *p = 'x';  /* exception */
}

int main(void) {
    signal(SIGSEGV, SIG_DFL);
    #pragma exception_handler(handler4, 0, 0, \
        _C2_MH_ESCAPE | _C2_MH_FUNCTION_CHECK)
    fred();
}
```

図4-38 T1520XH7 - 例外処理用の ILE C ソース

図4-38 内のソース・コードの実行時に発生する例外および処理アクションのシーケンスは、以下のとおりです。

1. 関数 fred() でエスケープ例外が発生します。
2. handler1 が制御権を獲得します。これは、エスケープ・メッセージをモニターしていて、例外に最も近いネストされたモニターであり、しかも直接ハンドラーであるため最高優先順位を持っているからです。
3. handler2 が制御権を獲得します。これは、エスケープ・メッセージをモニターティしていて、CEEHDLR よりも高い優先順位を持っているからです。
4. handler3 が (CEEHDLR から) 制御権を獲得します。
5. signal ハンドラーが制御権を獲得します。main で登録されていますが、signal は活動化グループがスコープであるため、制御権を獲得します。これは、直接ハンドラーまたは CEEHDLR より低い優先順位を持っているので、handler1, handler2, および handler3 の後に制御権を獲得します。アクションが SIG_DFL であるので、例外は処理されません。
6. 例外は main() にパーコレートされます。
7. handler4 が制御権を獲得します。
8. 例外はまだ処理されていません。したがって、制御境界 (main() の PEP) に達すると、関数チェックになり、再駆動されます。
9. handler1 は、関数チェックをモニタしていないので、制御権を獲得しません。
10. handler2 は、関数チェックをモニタしているので、制御権を獲得します。
11. handler3 は、CEEHDLR が *ESCAPE, *STATUS, *NOTIFY, および関数チェックのすべてのメッセージについて制御権を獲得するので、制御権を獲得します。
12. signal ハンドラーは、signal が関数チェックを認識しないので、制御権を獲得しません。
13. 関数チェックは main() にパーコレートされます。
14. handler4 は、関数チェックをモニタしているので、制御権を獲得します。
15. 関数チェックは制御境界にパーコレートされ、終了になります。
16. (CEE9901) *ESCAPE が main() の呼び出し元に送られます。

関数呼び出しおよび引数の数の削減

プログラム呼び出しの戻り値にアクセスする際に、追加の処理が必要になります。

以下によって、関数呼び出しおよび引数の数を削減できます。

- 関数呼び出しのインライン化
- 静的クラス・メンバーまたはグローバル・変数の使用
- レジスターでの引数の受け渡し
- プロトタイプを使用した関数呼び出し処理の最小化

関数呼び出しのインライン化

1 つの関数が少ない場所で呼び出されるが何度も実行される場合には、通常、その関数をインライン関数に変更すると、多数の関数呼び出しが削除され、パフォーマンスが改善されます。関数呼び出しをインライン関数またはマクロ式に変更することで、パフォーマンスを改善できることがあります。ただし、その変更でプログラム・オブジェクトのサイズが増加せず、プログラムをスローガウンするほどのページ不在が生じない場合に限ります。パフォーマンスを最適化するには、プログラムのサイズとインライン化またはマクロ式の間でバランスを取ります。4-75ページの表 4-9 を参照してください。

注: C++ C++ では、マクロ式はお勧めできません。代わりに、inline キーワードを使用し、インライン化をオンにします。

INLINE コンパイル時オプションにより、コンバイラーが関数呼び出しを、その関数呼び出しの代わりにその関数のコードに置き換えるように要求できます。コンバイラーでインライン化を行うことが許可される場合、関数呼び出しは、関数定義のソース・コードを表示マシン・コードで置き換えられます。
インライン化は、関数呼び出しのオーバーヘッドを除去することで、C または C++ プログラムの実行時のパフォーマンスを改善することができる方法です。インライン化により、最適化のためのプログラムの拡張ビューが可能になります。グローバル・スケールで定数およびフロー構造体を公開することで、コンパイラは最適化時に、より良い選択を行うことができます。

インライン化およびマクロの拡張については、以下を参照してください。
- ILE C/C++ 解説書
- ILE C/C++ コンパイラ参照

静的クラス・メンバー関数またはグローバル変数の使用

静的クラス・メンバーを使用して関数に引数を受け渡すことで、C++ プログラムの実行時のパフォーマンスを向上できる場合があります。

C または C++ のプログラムでは、関数に引数を渡す代わりに、変数をグローバルとして定義して、関数でグローバル変数を使用することができます。

注: 使用するグローバル変数の数が増えると、グローバル変数の割り振りと初期化のために活動化グループの開始で必要な作業量が増大し、最適化が妨げられる可能性があります。

クラス・メンバー関数とグローバル変数については、以下を参照してください。
- 4-57 ページの『静的変数およびグローバル変数の使用の最小限化』
- ILE C/C++ 解説書
- ILE C/C++ コンパイラ参照

レジスターでの引数の受け渡し

システムで全ての引数がレジスターで受け渡される場合、関数呼び出しのパフォーマンスを向上させることができます。レジスターは限られた数しかないため、すべての引数をレジスターで受け渡す可能性を上げるために、複数の引数を組み合わせて 1 つのクラスに入れ、クラスのアドレスを関数に渡します。アドレスが受け渡されるため、参照による受け渡しのセマンティクスが使用されます。これは、引数を個別の変数として受け渡した場合とは異なる可能性があります。

レジスターでの引数の受け渡しについて詳しくは、以下を参照してください。
- ILE C/C++ 解説書
- ILE C/C++ コンパイラ参照

ストレージ・クラスについて詳しくは、以下を参照してください。
- 4-58 ページの『レジスター・ストレージ・クラスの使用』
- ILE C/C++ 解説書

プロトタイプを使用して関数呼び出し処理を最小限化

関数プロトタイプは、関数の戻りの型、関数の名前、およびパラメータ・リストで構成されます。非プロトタイプ関数では、最初の参照時に有効なデータ・モデルによってシグニチャが推定されます。

注: C++ には、完全なプロトタイプ宣言が必要です。ISO C では、非プロトタイプ関数が認められていません。

extern OS リンケージを使用して C++ プログラムから動的にプログラムを呼び出すときは、int ではなく void を戻すようにプログラムをプロトタイプ化します。プログラム呼び出しの戻り値にアクセス
する際に、追加の処理が必要になります。パフォーマンスの観点からは、戻り値を保持できるストレージのアドレスを呼び出しの引数リストで受け渡すほうが適切です。

関数プロトタイプについては、「ILE C/C++ 解説書」を参照してください。

パフォーマンスを改善するための入出力関数の選択
このセクションでは、入出力に関するいくつかの問題について説明します。

レコード入出力関数の使用
ストリーム入出力関数の代わりにレコード入出力関数を使用すると、入出力パフォーマンスが大幅に改善します。1度に1バイトずつアクセスするのではなく、レコード入出力関数では1度に1レコードずつアクセスできます。

ILE C/C++ ランタイム・ライブラリーによってサポートされるレコード入出力のタイプには、ISO C レコード入出力と ILE C レコード入出力の2つがあります。

ISO C レコード入出力
プログラムで ISO C レコード入出力を使用する場合、ファイルを開くときに、fopen()のオープン・モード・バラメーターに type = record を指定する必要があります。さらに FILE データ・タイプを使用する必要があります。次の図に例を示します。

```
#include <stdio.h>
define MAX_LEN 80
int main(void)
{
    FILE *fp;
    int len;
    char buf[MAX_LEN + 1];
    fp = fopen("MY_LIB/MY_FILE", "rb, type = record");
    while ((len = fread(buf, 1, MAX_LEN, fp)) != 0)
    {
        buf[len] = '\0';
        printf("%s\n", buf);
    }
    fclose(fp);
    return 0;
}
```

図4-39. 例: ISO C レコード入出力の使用

ILE C レコード入出力
プログラム内に ILE C レコード入出力を使用する場合、以下のようにする必要があります。

• ILE C レコード入出力関数 (例えば _R で始まる関数) を使用します。
• _RFILE データ・タイプを使用します。

図4-39 の例は、以下のように書き直すことができます。
入出力フィードバック情報の使用

_RIOFB_T は、ILE C レコード関数からの入出力フィードバック情報 (例えば、読み取られたか、書き込まれたバイト数) を含む構造です。デフォルトでは、ILE C レコード入出力関数は、レコード入出力操作の実行後に _RIOFB_T のフィールドを更新します。

プログラムがこれらの値の一部を使用しない場合、以下の図に示すようにファイルを開いて、アプリケーションのパフォーマンスを改善することができます。

```c
fp = _Ropen("MY_LIB/MY_FILE", "rr, riofb = N");
```

図4-40 例: ILE C レコード入出力の使用

 riofb = N を指定すると、num_bytes フィールド (_RIOFB_T 構造での読み取りまたは書き込みバイト数)のみが更新されます。rio fb = Y を指定すると、_RIOFB_T 構造のすべてのフィールドが更新されます。

レコードのブロック化

レコードのブロック化によって、レコード入出力のパフォーマンスを改善することができます。ブロック化を指定すると、最初の読み取りでレコードのブロック全体がバッファーに入れられます。後続の読み取り操作では、バッファーが空になるまで、バッファーからレコードが戻されます。バッファーが空になった時点で、次のブロックが取り出されます。

```c
FILE データ・タイプを使用する際にレコードをブロック化したい場合、blksize=value を指定してファイルを開きます。この場合、value はブロック・サイズを示します。blksize に値ゼロを指定した場合、ファイルを開くときに、ブロック・サイズが計算されます。
```

```c
_RFILE データ・タイプを使用する際にレコードをブロック化したい場合、ファイルを開くときに、blkrcd = Y を指定します。
```

書き込み操作でのレコードのブロック化の際にも同様の規則が適用されます。

--

4-64  IBM i: ILE C/C++ プログラマーの手引き
システム・バッファーの操作
システム・バッファーに対して読み書き操作を直接実行することで、ILE C/C++ プログラムの入出力パフォーマンスを改善することができます。アプリケーション定義のバッファーは必要ありません。このシステム・アクセスは、位置指定モードと呼ばれています。以下は、ソース物理ファイルを読み取るときに、システム・バッファーを直接操作する方法を示しています。

```c
fp = _Ropen("MY_LIB/MY_FILE", "rr", blkrcd = Y, riofb = N);
while ( (_Rreadn(fp, NULL, 92, __DFT))->num_bytes != EOF )
{
    printf("%75.75s\n", ((char *) (*(fp->in_buf))) + 12);
}
_Rclose(fp);
```

図4-42. システム・バッファーの使用
上の例のコードは、ファイルに含まれる各レコードの最大 75 文字を印刷します。_Rreadn() の 2 番目のパラメーター (NULL) は、システム・バッファーのレコードの操作を可能にします。_RFILE 構造には、in_buf と out_buf のフィールドが含まれます。これらはそれぞれシステム入力バッファーとシステム出力バッファーを指します。上の例では、システムの入力バッファーにアクセスすることで各レコードが印刷されます。

システム・バッファーの直接操作では、非常に長いレコードを処理するときに、パフォーマンスが向上します。また、システム間通信機能 (ICF) ファイルを使用する際にも、明らかなパフォーマンスの向上をもたらします。通常、ICF ファイルの最後の数バイトにアクセスするだけで済み、レコード内の他のすべてのデータにはアクセスする必要はありません。システム・バッファーを直接使用すると、ICF ファイルを使用しないデータはコピーする必要がありません。

システム・バッファーは、常に<_reio.h> ヘッダー・ファイルにある _RFILE 構造内の in_buf および out_buf ポイントを介してアクセスする必要があります。システム・バッファーが in_buf および out_buf ポイントを介さずにアクセスされると、予測不能な結果が生じることがあります。

入出力の両方でファイルを 1 回開く
アプリケーションがデータをファイルに書き込み、その後でそのデータを読み取る場合、通常では入力および出力の両方を実行するために 2 回ファイルが開かれますが、代わりにファイルを 1 回のみ開くことで、パフォーマンスを改善することができます。以下は、ファイルがどのように 2 回開かれ、2 回閉じられるかを示しています。

```c
fp = _Ropen("MY_LIB/MY_FILE", "wr"); /* Output only.*/
_Rclose(fp);
/* Code to write data to MY_FILE */

fp = _Ropen("MY_LIB/MY_FILE", "rr"); /* Input only.*/
_Rclose(fp);
/* Code to read data from MY_FILE. */
```

図4-43. 例: ファイルを 2 回開く
この例を以下のように変更すると、_Ropen と _Rclose のそれぞれ 1 回の呼び出しを省けます。
ファイルの使用の最小化

ファイルのオープンおよびクローズ数の最小限化

バフォーマンス改善のためのテープ・ファイルの定義

ストリーム入出力関数を使用する際のバフォーマンスの改善

ILE C のレコード入出力関数を使用すると、ストリーム入出力関数に比べてバフォーマンスが大幅に改善しますが、ストリーム入出力を使用する際にもバフォーマンスを改善する方法があります。

CRTCMTD または CRTBNDC コマンドに SYSIFCOPX (*IFSIO) を指定すると、レコード入出力関数と同様のバフォーマンスで IFS ストリーム・ファイルを使用できます。

ファイルから文字を読み取るのに、fgetc() の代わりに getc() のマクロ・バージョンを使用してください。4-62 ページの「静的クラス・メンバー関数またはグローバル変数の使用」を参照してください。getc() のマクロ・バージョンは、バッファーが空になるまで、バッファー内ですべての文字を読み取ります。バッファーが空になると、getc() は fgetc() を呼び出して、次のレコードを取得します。

同じ理由から、fputc() の代わりに putc() を使用してください。putc() のマクロ・バージョンは、バッファーがいっぱいになるまで、バッファーにすべての文字を書き込みます。バッファーがいっぱいになると、putc() は fprintf() を呼び出してレコードをファイルに書き込みます。

ストリーム入出力関数では多くの関数呼び出ししか行われないので、その使用を削減すると、アプリケーションのバフォーマンスが改善されます。以下は、printf() の呼び出しを示しています。

printf("Enter next item.");
printf("When done, enter 'done'.\n");

4-45. printf() の使用

以下のように printf() の 2 つの呼び出しを 1 つの呼び出しに結合すると、1 つ呼び出しを省くことができます。
printf("Enter next item.\n" "When done, enter 'done'.\n");

図 4-46. 関数呼び出し削減のための printf() の使用

### C++ 入出力ストリーム・クラスの使用

標準ストリームでは、C 同等物ではなく多重定義シフト << >> 演算子を使用してください。

#### ソース物理ファイルに代わる物理ファイルの使用

パフォーマンスを改善するには、データにソース物理ファイルの代わりに物理ファイルを使用します。

ストリーム入出力にソース物理ファイルを使用すると、各レコードの最初の 12 バイトがアプリケーションに可視になりません。これは、レコード番号と更新時刻の保持に使用されます。これらの 12 バイトは、ILE C ストリーム入出力関数で対処する必要がある余分の負荷となります。以下に例を示します。

- 出力の実行時には、これらの 12 バイトをゼロに初期化する必要があります。
- 入力の実行時には、アプリケーションには渡されませんが、これらの 12 バイトを取り出す必要があります。

出力用にまだ存在しないテキスト・ファイルを開くときに、ILE C ストリーム入出力関数はソース物理ファイルを動的に作成します。この理由からファイルを物理ファイルとして作成してから、アプリケーションを開始します。

#### ライブラリ名の指定

ファイルが存在するライブラリの名前を指定する必要があります。ファイルの処理時にライブラリ名を指定しない場合には、ライブラリ・リストがそのファイルを見つけするために検索されます。ライブラリの数やその中のオブジェクトの数によっては、検索時間が長くなる可能性があります。

#### ポインターを使用したパフォーマンス向上

ポインターの使用および比較は、パフォーマンスに影響する可能性があります。

#### オープン・ポインターの使用の回避

オープン・ポインターを使用することは、避けてしまってください。オープン・ポインターでは最適化を行えません。void へのポインター(void*) は、ILE C/C++ においてオープン・ポインターであることに注意してください。

#### ポインターの比較の回避

ポインターは 16 バイトのスペースを消費するため、ポインターの比較は、他のデータ・タイプを使用した比較よりも、効率が低下します。ポインターの比較は、int などの他のデータ・タイプを使用した比較に置き換えるとよい場合があります。

次の図は、リンク・リストを構成し、リスト内の全エレメントを処理した後、リンク・リストを解放するプログラムを示しています。
#include <string.h>  
#include <stdlib.h>  
#include <recio.h>  

#define MAX_LEN 80  

struct link  
{  
    struct link *next;  
    char record[MAX_LEN];  
};  

int main(void)  
{  
    struct link *start, *ptr;  
    _RFILE *fp;  
    return 0;  
    int i;  

    // Construct the linked list and read in records.  
    fp = _Ropen("MY_LIB/MY_FILE", "rr, blkrcd = Y");  
    start = (struct link *) malloc(sizeof(struct link));  
    start->next = NULL;  
    ptr = start;  
    for (i = (_Ropenfbk(fp))->num_records; i > 0; i--)  
    {  
        _Readn (fp, NULL, MAX_LEN, _DFT);  
        ptr = ptr->next = (struct link *) malloc(sizeof(struct link));  
        memcpy(ptr->record, (void const *) *(fp->in_buf), MAX_LEN);  
        ptr->next = NULL;  
    }  
    ptr = start->next;  
    free(start);  
    start = ptr;  
    _Rclose(fp);  
    return 0;  

    // Process all records.  
    for (ptr = start; ptr != NULL; ptr = ptr->next)  
    {  
        // Code to process the element pointed to by ptr.  
    }  

    // Free space allocated for the linked list.  
    while (start != NULL)  
    {  
        ptr = start->next;  
        free(start);  
        start = ptr;  
    }  

図4-47 リンク・リストを使用するプログラムの例

リンク・リストの各エレメントには、ファイルからの1つのレコードが保持されます。

上記のプログラムでは、エレメントの処理時とリンク・リストの解放時にポインタの比較が使用されています。このプログラムは、リンク・リストの終わりを示すためにshort型のメンバーを使用して書き直すことができます。結果として、次の図に示すように、ポインタの比較を整数の比較に変更します。
#include <string.h>
#include <stdlib.h>
#include <recio.h>

#define MAX_LEN 80
int i;

struct link
{
    struct link *next;
    short last;
    char record[MAX_LEN];
};

int main(void)
{
    struct link *start, *ptr;
    _RFILE *fp;
    return 0;

    // Construct the linked list and read in records.
    fp = _Ropen("MY_LIB/MY_FILE", "rr, blkrcd = Y");
    start = (struct link *) malloc(sizeof(struct link));
    start->next = NULL;
    ptr = start;
    for (i = (_Ropnfbk(fp))->num_records; i > 0; --i)
    {
        _Rreadn(fp, NULL, MAX_LEN, __DFT);
        (struct link *) malloc(sizeof(struct link));
        memcpy(ptr->record, (void const *) *(fp->in_buf), MAX_LEN);
        ptr->last = 0;
    }
    ptr->last = 1;
    ptr = start->next;
    free(start);
    start = ptr;
    _Rclose(fp);

    // Process all records.
    if (start != NULL)
    {
        for (ptr = start; !ptr->last; ptr = ptr->next)
        {
            // Code to process the element pointed to by
            // code to process the element
            // (last element) pointed.
            // Free space allocated for the linked list.
            while (!start->last)
            {
                ptr = start->next;
                free(start);
                start = ptr;
            }
            free(start);
        }
    }
}

図4-48. short 型のメンバーを使用してリンク・リストを終了するソース・コードの例
ポインターによる間接アクセスの削減

ポインターによる間接アクセスの削減により、パフォーマンスを向上させることができます。各間接レベルで、オーバーヘッドが追加されます。

```c
for (i = 0; i < n; i++)
{
    x->y->z[i] = i;
}
```

上記の例は、次のように書き直すとパフォーマンスが向上します。

```c
temp = x->y;
for (i = 0; i < n; i++)
{
    temp->z[i] = i;
}
```

ディープ・コピーの代わりにシャロー・コピーを使用

シャロー・コピーだけが必要な場合は、ディープ・コピーを実行しないようにしてください。他のオブジェクトへのポインターを含むオブジェクトについて、シャロー・コピーはポインターだけをコピーし、それらが指すオブジェクトをコピーしません。その結果、同じものが含まれたオブジェクトを指す 2 つのオブジェクトができます。ディープ・コピーは、ポインターとそれが指すオブジェクトのほか、そのオブジェクト内に含まれているすべてのポインターやオブジェクトなどもコピーします。

注: プログラムが 2 回以上オブジェクトを指す場合は、ディープ・コピーを使用する必要があります。シャロー・コピーを使用するオブジェクトは、2 回以上指されたオブジェクトを破棄する可能性があります。

スペース所要量の最小限化

必要なスペースを削減することで、プログラムのパフォーマンスを向上させることができます。スペース所要量を削減すると、ページ不在、セグメント不在、および有効アドレス・オーバーフローを減らすことができます。

適切なデータ型の選択

適切なデータ型を選択することで、プログラムのスペース所要量を減らすことができ、また、プログラムのパフォーマンス向上にもつながります。データ型を選択する際は、ご使用のコードがサポートする必要のあるすべてのプラットフォームを考慮してください。コード設計の最初の段階では、すべてのデータ型およびサイズは把握されていない可能性があります。データ型はさまざまなプラットフォームで同じサイズ・データを保持できるため、データ型の用途に基づいて、typedef、enum、または class を使用することができます。コンパイラーは、short に 2 バイトを、int に 4 バイトを、double に 8 バイトを使用します。

動的メモリー割り振り呼び出しの最小限化

メモリーを動的に割り振る回数を削減することで、パフォーマンスを向上させることができます。new 演算子を呼び出すたびに、ヒープから一定量のスペースが割り振られます。このスペースは常に、どのオブジェクトも型のストレージにも適した 16 バイトで位置合わせます。さらに、ブックキープのためには動的ヒープから追加で 32 バイトを消費します。つまり、1 バイトしか必要な場合でも、ブックキープの 32 バイトと埋め込みの 15 バイトのために、動的ヒープから 48 バイトが割り振られます。ヒープ内の現行スペース割り振りを使い切ると、ストレージ割り振りは遅くなります。

```c
ptr1 = new char[12];
ptr2 = new char[4];
```
上記のコードでは、ヒープから 96 バイト (ブックキービンの 64 バイトと埋め込みの 16 バイトを含む) を消費し、new を 2 回使用しています。このコードは、以下のように書き直すことができます。

```c
ptr1 = new char[16];
ptr2 = ptr1 + 12;
```

ヒープから消費するのは 48 バイトだけで、new 演算子は 1 回しか使用していません。動的スペース割り振りの所要量が減るため、ヒープから消費するストレージが減ります。ページ不在の削減といった、他の効果もあるかもしれません。new 演算子の呼び出しを減らすため、関数呼び出しのオーバーヘッドも削減されます。

注: ポインタのインクリメントによりスペースを割り振る場合、ポインターには 16 バイトの位置合わせが必要なため、ポインター (またはポインターを含むことができる集約) を割り振る際には、必ず、適切な位置合わせが保証されるようにしてください。float などのデータ・タイプでは、ワードまたはダブルワードで本来位置合わせするため、これらの本来の境界で割り振られないと、パフォーマンスが低下します。

**変数の配置による埋め込みの削減**

変数の再配置によって埋め込みで浪費したスペースを削減することでも、プログラムのスペース所要量を減らすことができます。

バック 10 進数データ・タイプを除き、変数は、C と C++ のどちらでも、次のように同じように埋め込みが行われます。

- `char` タイプの変数は 1 バイトを使用
- `short` タイプの変数は 2 バイトを使用
- `int` タイプの変数は 4 バイトを使用
- `long` タイプの変数は 4 バイトを使用
- `long long` タイプの変数は 8 バイトを使用
- `float` タイプの変数は 4 バイトを使用
- `double` タイプの変数は 8 バイトを使用
- ポインターは 16 バイトを使用
- `_DecimalT` テンプレート・クラス・オブジェクトは 1 バイトから 16 バイトを使用

C のバック 10 進数タイプの変数のサイズには 1 バイトから 32 バイトが可能です。

C++ のバック 10 進数タイプの変数のサイズには 1 バイトから 16 バイトが可能です。

第 4-72 ページの図 4-49 に示すように、変数を再配置することによって、埋め込みで生成される無駄なスペースを最小限に抑えることができます。
class OrderT
{
    float value;  // Four bytes.
    char flag1;   // One byte plus one byte.
    short num;    // Two bytes.
    char flag2;   // One byte plus three bytes.
}
orderT;
class ItemT
{   
    char *name;    // 16 bytes.
    int number;    // 4 bytes plus 12 bytes.
    char *address; // 16 bytes.
    double value;  // 8 bytes plus 8 bytes.
    char *next;    // 16 bytes.
    short rating;  // 2 bytes plus 14 bytes.
    _DecimalT<25,5> tot_order; // 13 bytes plus 3 bytes.
    int quantity;  // 4 bytes.
    _DecimalT<12,5> unit_price; // 7 bytes plus 5 bytes.
    char *title;   // 16 bytes.
    char flag;    // 1 byte plus 15 bytes.
}
itemT;
注：ItemT クラスの構造は 176 バイトを使用します。このうち、57 バイトが埋め込みに使用されます。

ItemT クラスは、次のように再配置することができます。
class ItemT
{   
    char *name;    // 16 bytes
    char *address; // 16 bytes
    char *next;    // 16 bytes
    char *previous; // 16 bytes
    char *title;   // 16 bytes
    double value;  // 8 bytes
    int quantity;  // 4 bytes
    int number;    // 4 bytes
    short rating;  // 2 bytes
    char flag;    // 1 byte
    _DecimalT<25,5> tot_order; // 13 bytes
    _DecimalT<12,5> unit_price; // 7 bytes plus 9 bytes
}
itemT;
注：再配置の後、ItemT クラスが使用するのは 128 バイトのみで、埋め込みには 9 バイトです。類似した構造タイプの配列を再配置すると、さらに大幅にスペースを節約することができます。

図 4-49 変数の再配置によって埋め込みを最小限にする例
一般的な規則として、16 バイトの変数を最初に宣言し、8 バイトの変数を 2 番目に、4 バイトの変数を 3 番目に、2 バイトの変数を 4 番目に、1 バイトの変数を 5 番目に宣言すると、埋め込みに使用されるスペースを最小限に抑えられます。_DecimalT テンプレート・クラス・オブジェクトは、他のすべての変数を宣言後の後、最後に宣言します。構造またはクラスの定義にも、同じ規則を適用できます。
モジュール構造のレイアウト（埋め込みを含む）を表示するには、パックと通常のどちらの位置合わせでも、*AGR または *STRUCREF のコンバイラー・オプションを使用します。
注：*AGR (すべての構造のマップを示す) は、*STRUCREF (参照される構造のマップを示す) をオーバーライドします。
プログラム識別情報の除去
再コンパイルなしにモジュールを変更可能にするデータがモジュールに含まれる場合、そのモジュールは識別情報を持ちます。以下の 2 つのタイプのデータで、モジュールを識別可能にすることができます。

作成データ
このデータはコードを機械語命令に変換するのに必要です。最適化レベルを変更する前に、オブジェクト化がなければいけません。これは、プログラム変更 (CHGPGM) コマンドの RMVOBS パラメーターの値 *CRTDTA で表されます。

デバッグ・データ
このデータにより、オブジェクトがデバッグ可能になります。これは、CHGPGM コマンドの RMVOBS パラメーターの値 *DBGDTA で表されます。

これらのタイプのデータを追加すると、オブジェクトのサイズが増します。したがって、オブジェクトのサイズを減らすために、ある時点でデータを除去したくなるかもしれませんが。しかし、データを除去してしまうと、オブジェクトのプログラム識別情報も除去されます。これを取戻すためには、ソースを再コンパイルしてプログラムを再作成する必要があります。

どちらの種類のデータでも、プログラムまたはモジュールからの除去には、CHGMOD または CHGPGM コマンドを使用してください。既に説明したとおり、このデータを一度除去したら、オブジェクトを再作成しない限り、オブジェクトを変更することはできません。したがって、プログラムの作成に必要なすべてのソースに対するアクセス権限をもっているか、あるいは作成データと同等のプログラム・オブジェクトをもっていることを確認してください。

オブジェクトの圧縮
ILE プログラムまたはモジュールに関連した作成データ (*CRTDTA) の値は、オブジェクト・サイズの半分以上を占める場合があります。このデータを除去するかまたは圧縮することにより、プログラムのための 2 次的記憶域の必要量を大きく減らすことになります。

代わりに、オブジェクトの圧縮 (CPOBJ) コマンドを使用して、オブジェクトを圧縮する方法もあります。圧縮されたオブジェクトは圧縮されていないオブジェクトより小さなシステム記憶域を占めます。圧縮プログラムが呼び出された場合には、実行可能コードが入っているオブジェクトの一部が自動的に圧縮解除されます。また、オブジェクトの圧縮解除 (DCPJOB) コマンドを使用することによって、オブジェクトを圧縮解除することができます。

活動化グループの使用の最適化
活動化グループの使用は、パフォーマンスに影響を及ぼす場合があります。

他の活動化グループでの関数の呼び出し
同じジョブ内で、別の活動化グループで実行される関数を呼び出すとき、呼び出しのパフォーマンスが大幅に低下します (呼び出しにかかる時間が約 2 倍になります)。

サービス・プログラムが (CRTSRVPGM コマンドの ACTGRP(name) パラメーターを使用して) 名前付き活動化グループで実行されるよう作成されている場合、プログラムまたはサービス・プログラムからその関数に対するすべての呼び出しは、活動化グループを超えて呼び出しを行うため、速度が遅くなります。(記憶域の分離、例外処理などの目的で) 他の活動化グループでプログラムまたはサービス・プログラムを実行するのが妥当な場合もありますが、その配置では呼び出しのパフォーマンスが低下することに注意してください。
プログラム開始処理時間の短縮

新規 ILE プログラムが最初に呼び出されたとき、システムは、プログラムの実行準備のために初期化を実行する必要があります。この初期化の中では、全プログラム・ストレージ用の活動化グループの作成、プログラムにバインドされたすべてのサービス・プログラムの解凍、プログラム引数の取得などが必要です。こうした初期化ステップで、開始処理時間を改善するために、以下の推奨事項が挙げられます。

- グローバル変数の使用を削減します。
- プログラムにバインドされたサービス・プログラムの数を削減します。ILE プログラムで使用されるサービス・プログラムが多いほど、プログラムの開始に必要な時間は増大します。多数の小規模なサービス・プログラムがあるより、少数の大規模なサービス・プログラムがあるほうが、多くの場合適切です。例えば、C ランタイム・ライブラリーは、少数のサービス・プログラムから成ります。

仮想関数の使用の最小限化

仮想関数はコンパイルされること、直接呼び出しが遅い関数呼び出しになるため、仮想関数を使用した場合、パフォーマンスの影響があります。プログラム設計によっては、仮想関数で使用するパラメーター数を最小におくことで、このパフォーマンスの影響を最小限に抑える場合があります。

速度またはサイズの最適化用コンパイラ・オプションの選択

コンパイル時のパフォーマンスを改善する手段がいくつかあります。これらには、フロントエンドとバックエンドの両方のコンパイル時アクティビティが含まれます。

[4-75 ページの表 4-9] では、プログラムの実行速度を速くしたり、コンパイル済みプログラムを小さくしたりするための各種コンパイラ・オプションについて説明します。なお、プログラム・サイズとプログラム速度のどちらが重要なのかを決定する必要が生じることもあります。場合によっては、一方の側面について最適化すると、他方の側面に悪影響が出ます。

最適化 は、出力の生成に必要なシステム・リソース量を削減する処理のショートカットをシステムが検索する際に使用するプロセスです。処理のショートカットはマシン・コードに変換され、モジュール内のプロシージャーの実行効率を上げることができます。高度に最適化されたプログラムまたはサービス・プログラムは、最も適用されていない場合よりも高速に実行されます。

最適化レベルを制御するには、モジュールの作成コマンドおよびバインド済みプログラムの作成コマンドで OPTIMIZEオプションを使用します。目的の最適化レベルを変更するには、ソース・コードを再コンパイルする必要があります。モジュールの変更 (CHGMOD) コマンドを使用してモジュールの最適化を変更することもできます。

注: モジュールの変更 (CHGMOD) コマンドを使用して、ソース・コードを再コンパイルすることなく最適化レベルを変更することはできません。

最適化されたコードを処理する際には、以下の制限に注意する必要があります。
- 一般に、最適化の要求が高くなるにしたがって、オブジェクトの作成にかかる時間が長くなります。
- 最適化レベルが高くなると、フィールドの値が、デバッグ・セッションで表示される際またはプログラムが例外から復旧した後に、正確でないことがあります。
- 最適化によって一部のステートメントが再配置されたり削除されたりすることがあるため、最適化されたコードでは、ソース・デバッガーによって使用される停止点およびステップ位置が変更されることがあります。
デバッグ時のこの制限を回避するには、プログラムをデバッグする時にはフィールドを正しく表示するためにモジュールの適化レベルを低くし、その後プログラムを実行用に使用可能にする時に、プログラムの効率を高めるためにレベルを再び上げることができます。

矛盾する場合を除いて、表 4-9 のガイドラインを使用してください。組み込み関数はパフォーマンスを改善する可能性がありますが、モジュールのサイズが増加します。

表 4-9. パフォーマンス用コンパイラ・オプション

<table>
<thead>
<tr>
<th>オプション</th>
<th>速度の最適化</th>
<th>サイズの最適化</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTIMIZE 10/*Default value</td>
<td>はい</td>
<td>はい</td>
</tr>
<tr>
<td>OPTIMIZE 20</td>
<td>いいえ</td>
<td>はい</td>
</tr>
<tr>
<td>OPTIMIZE 30</td>
<td>いいえ</td>
<td>はい</td>
</tr>
<tr>
<td>OPTIMIZE 40</td>
<td>いいえ</td>
<td>はい</td>
</tr>
</tbody>
</table>

最適化をオンにします。

INLINE(*OFF)

インライン化をオフにします。インライン化関数が少ないコード部分で構成されている場合には特に、モジュール・サイズが削減されることがあります。

INLINE(*ON)

インライン化をオンにします。1つの関数を複数の場所で呼び出される場合でも実行される場合に、多数の関数呼び出しを省くことができます。

DBGVIEW(*NONE)

モジュール・サイズを大きくするとデバッグ情報を生成しません。

実行時間の設定

・単一変数 (ストリングや配列など) のストレージ最大量は、16,773,104 バイトです。
・システム関数に受けるコマンドの最大長は、32,702 バイトです。
・動的ヒープ・ストレージの最大サイズは、4 ギガバイトです。

システムに十分な補助ストレージがない場合、非常に大きなメモリーの割り振りを行うと、システムがクラッシュする可能性があります。4 ギガバイトのメモリーを割り振るには、4 ギガバイトを超える使用可能な DASD が必要です。システム状況処理 (WRKSYSSTS) コマンドで、補助ストレージの使用状況が表示されます。

・単一ヒープ割り振りの最大サイズは、16,711,568 バイトです。
・最大自動ストレージは 16 MB で、約 21,743 レベルの深さという再帰の限度があります。

例: ILE C アプリケーションの作成

このセクションの例では、サンプル ILE C アプリケーションを作成する標準的なステップをいくつか示します。

このアプリケーションは、品名、価格、および数量を入力として取る小規模トランザクション処理プログラムです。出力として、このアプリケーションは画面に品目の総コストを表示し、トランザクションの監査証跡を更新します。

このトピックでは、サンプル・アプリケーションについて以下を説明します。
プロセス・フロー

図4.50 サンプル・アプリケーション: 入力/処理/出力フローの概要

図4.50 は、以下を示しています。

セッション入力
端末セッション時に入力されるデータ
• 発注対象の品目の名前
• 単価
• 発注対象の単位数量

CL CMD プログラム T1520CM1
ユーザーアクセスを受け入れた制御言語プログラム T1520CL1 に渡す、開発者が作成した CL コマンド。

制御言語プログラム T1520CL1
入力を処理して ILE C プログラム T1520PG1 に渡す、制御言語プログラム。

ILE C プログラム T1520PG1
入力を処理して出力をユーザーの端末および外部記述ファイルに送信する ILE C プログラム。ILE C プログラムは、4-77ページの図4.51 に示されているように 2 つのモジュール (T1520IC1 および T1520IC2) で構成されています。モジュール T1520IC1 は、calc_and_format() プロシージャーを呼び出すユーザー入り口プロシージャー main() を提供します。

サービス・プログラム T1520SP1
4-77ページの図4.51 に示すように、プログラムで write_audit_trail() プロシージャーをインポートできるようにする ILE サービス・プログラム。
サービス・プログラム T1520SP2

図4.51に示すように、プログラムで税率データ項目をインポートできるようにする ILE サービス・プログラム。

セッション出力
画面に以下が表示されます。
・ ステートメント："(発注対象の単位数量) (発注対象の品目名) plus tax = (ユーザーへの計算されたコスト)"
・ プロンプト："Press ENTER to end terminal session."

監査ファイル T1520DD1
トランザクションごとに更新されるログ。4-83 ページの図4-52に示しているような DDS ソースにより、監査ファイルのデータ・フィールドおよび関係 (つまり、レイアウト) が定義されます。

ILE 活動化グループ

CL コマンド・プログラムが制御言語プログラムを呼び出すと、これらのプログラムを実行するために必要なすべてのリソースが、デフォルトの活動化グループに割り振られます。

制御言語プログラムが ILE C プログラムを呼び出すと、図4-51に示しているように、新規活動化グループが開始されます。これは、ILE C プログラムが、ACTGRP(*NEW) パラメーターを指定して作成されているためです。ILE C サービス・プログラムも、ACTGRP(*CALLER) パラメーターを指定して作成されているため、この新規活動化グループで活動化されます。

言い換えると、ILE C プログラムおよび ILE C サービス・プログラムは、1つの連携アプリケーションとして開発されているため、1つの活動化グループ内で活動化されます。

注: 同じジョブで実行しているプログラムから一部のプログラムを分離するために、異なる活動化グループを選択できます。例えば、4つの異なるベンダーからのソフトウェア・パッケージを統合することによって、顧客の完全なソリューションを提供できます。活動化グループを分けると、各ベンダー・パッケージに関連付けられたリソースを分離することで、統合が容易になります。

資源要件
サンプル・アプリケーションを作成するには、以下の資源を作成する必要があります。
・ILE C アプリケーション・モジュールおよびサービス・プログラム用のバインディング・ディレクトリ

以下にコード例を列挙します。

出力ファイル用のデータ記述仕様 (DDS) T1520DD1
制御言語プログラム T1520CL1
CL CMD プログラム T1520CM1
ILE C プログラム T1520PG1 モジュール T1520C1 および T1520C2
サービス・プログラム T1520SP1 モジュール T1520C3 およびそのバインダー言語 QSRVSRC
サービス・プログラム T1520SP2 モジュール T1520C4 およびそのバインダー言語 QASRVSRC

注: この例は、例示用です。このサイズの ILE C プログラム用のバインディング・ディレクトリを作成する必要はありません。示されているようにデータと関数で ILE C サービス・プログラムを分断しないことも可能です。

タスクの要約

「IBM i: ILE C/C++ プログラマの手引き」の4-78ページに、サンプル ILE アプリケーションを作成するために実行する必要があるタスクをリストします。以下に各タスクの内容を示します。

<table>
<thead>
<tr>
<th>タスク</th>
<th>コンポーネント</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 監査ログが含まれる物理ファイルを作成します。DDS ソースは、監査ファイルのフィールドを定義します</td>
<td>T1520DD1</td>
</tr>
<tr>
<td>2. 必要なパラメーターを ILE C プログラム T1520PG1 に渡す制御言語プログラムを作成します</td>
<td>T1520CL1</td>
</tr>
<tr>
<td>3. ユーザーの端末セッションからデータを収集する CL コマンド・プロンプトを作成します</td>
<td>T1520CM1</td>
</tr>
<tr>
<td>4. UEP (main() 関数) を提供するモジュールを作成します。これは、以下の関数を行います。</td>
<td>T1520C1</td>
</tr>
</tbody>
</table>
| 4-1. ユーザー入力を制御言語プログラム T1520CL1 から受け取ります
| 4-2. モジュール T1520C2 の calc_and_format() 関数を呼び出す。この関数は以下の関数を呼び出します
| 4-2-1. 品目合計コストを計算します
| 4-2-2. モジュール T1520C3 の write_audit_trail() 関数を呼び出します | T1520C2 |
| 5. 呼び出し関数 calc_and_format() を提供するモジュールを作成します。この関数は、以下によって統合の計算を完了します。 | T1520C3 |
| 5-1. モジュール T1520C1 から関数を受け取ります
| 5-2. ILE C サービス・プログラムから税率データ項目をインポートします | T1520C4 |
| 6. write_audit_trail() 関数を提供するモジュールを作成します。このモジュールは、ILE C サービス・プログラム T1520SP1 を作成します。 | |
| 7. 税率データをエクスポートするモジュールを作成します。このモジュールは、ILE C サービス・プログラム T1520SP2 を作成します。 | T1520C5 |
| 8. ボリュームにある write_audit_trail を ILE サービス・プログラム T1520SP1 からエクスポートするためのバインダー言語が含まれたソース物理ファイル QSRVSRC を作成します。 | QSRVSRC |
### サンプル・アプリケーションを作成するための説明

1. 以下のコマンドを入力して、監査ログ項目が含まれる物理ファイル T1520DD1 を作成します。
   
   CRTPF FILE(MYLIB/T1520DD1) SRCFILE(QCPPLE/QADDSSRC) MAXMBRS(*NOMAX)
   
   注: 4-83 ページの図 4-52 に、T1520DD1 のソース・コードを示します。

2. 以下のコマンドを入力して、ILE C プログラム T1520PG1 にパラメーターを渡す制御言語プログラム T1520CL1 を作成します。
   
   CRTCLPGM PGM(MYLIB/T1520CL1) SRCFILE(QCPPLE/QACLSRC)
   
   注: 4-83 ページの図 4-53 に、T1520CL1 のソース・コードを示します。

3. 以下のコマンドを入力して、品目名、価格、および数量のデータを収集する CL コマンド・ブロンプト T1520CM1 を作成します。
   
   CRTCMD CMD(MYLIB/T1520CM1) PGM(MYLIB/T1520CL1) SRCFILE(QCPPLE/QACMDSRC)
   
   注: 4-84 ページの図 4-54 に、T1520CM1 のソース・コードを示します。

4. 以下のコマンドを入力して、main() 関数を提供するモジュール T1520IC1 を作成します。
   
   CRTCMOD MODULE(MYLIB/T1520IC1) SRCFILE(QCPPLE/Q CSRC) OUTPUT(*PRINT) DBGVIEW(*ALL)
   
   注: 4-85 ページの図 4-55 に、T1520IC1 のソース・コードを示します。
   
   OUTPUT(*PRINT) は、コンパイラ・リストが必要であることを指定します。

### 表 4-1O サンプル ILE アプリケーションを作成するために必要なタスクの要約 (続き)

<table>
<thead>
<tr>
<th>タスク</th>
<th>コンポーネント</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>データ項目 taxrate を ILE サービス・プログラム T1520SP2 からエクスポートするためのバインダー言語が含まれたソース物理ファイル QASRVSRC を作成します。</td>
</tr>
<tr>
<td>1</td>
<td>サービス・プログラム T1520SP1 および T1520SP2 が含まれたバインディング・ディレクトリーを作成し、サービス・プログラム名をそのディレクトリーに追加します。</td>
</tr>
<tr>
<td>2</td>
<td>以下から ILE C サービス・プログラム T1520SP1 を作成します。</td>
</tr>
<tr>
<td>3</td>
<td>以下のコンポーネントから ILE プログラム T1520PG1 を作成します。</td>
</tr>
<tr>
<td>4</td>
<td>プログラムを実行してテストします。</td>
</tr>
</tbody>
</table>

1. QUSRTOOL ライブラリーに、1 つ以上のモジュール用のバインダー言語の生成に役立つツールが用意されていま す。ファイル QUSRTOOL/QATTINFO のメンバー TBNINFO を参照してください。
5. 以下のコマンドを入力して、税金を計算して出力用に合計コストをフォーマット設定するモジュール T1520IC2 を作成します。

CRTC MOD MODULE(MYLIB/T1520IC2) SRCFILE(QCPPLE/QACSRC) OUTPUT(+PRINT)
DBG VIEW(*ALL)

注:
- 4-87 ページの図 4-56 に、T1520IC2 のソース・コードを示します。
- DBG VIEW(*ALL) は、このモジュールをデバッグするために、デバッグ・データとともにルート・ソース・ビューおよびリスト・ビューが必要であることを指定します。

6. 以下のコマンドを入力して、監査ファイル T1520DD1 で監査証跡を更新するモジュール T1520IC3 を作成します。

CRTC MOD MODULE(MYLIB/T1520IC3) SRCFILE(QCPPLE/QACSRC)
OUTPUT(+PRINT) DBG VIEW(+SOURCE) OPTION(*SHOWUSR)

注:
- 4-88 ページの『監査証跡を書き込むためのソース・コード』 に、T1520IC3 のソース・コードを示します。
- DBG VIEW(*SOURCE) OPTION(*SHOWUSR) パラメーターは、このモジュールをデバッグするために、ルート・ソース・メンバー、ユーザー組込みファイル、およびデバッグ・データが含まれた組み込みビューが必要であることを指定します。
- OPTION(*SHOWUSR) パラメーターは、#pragma mapinc ディレクティブで指定されているように、DDS ソース・ファイル MYLIB/T1520DD1 からコンパイラによって生成されたタイプ定義をコンパイラ・リストおよび組込みデバッグ・ビューに展開します。

7. 以下のコマンドを入力して、税率データをエクスポートするモジュール T1520IC4 を作成します。

CRTC MOD MODULE(MYLIB/T1520IC4) SRCFILE(QCPPLE/QACSRC)
OUTPUT(+PRINT) DBG VIEW(*SOURCE)

注:
- 4-89 ページの図 4-57 に、T1520IC4 のソース・コードを示します。
- OPTION(*XREF) パラメーターは、ソース・コード内の ID とそれが現れる行番号のリストが入った相互参照表を生成します。この表は、変数 taxrate のクラス、長さ、およびタイプを提供します。このクラスは外部定義です。長さは 2 です。タイプは、定数 decimal(2,2) です。この例におけるこのオプションの使用は、例示用です。通常、このオプションを使用するのは、複数の変数参照または実行可能ステートメントがある場合です。
- DBG VIEW(*SOURCE) パラメーターは、このモジュールをデバッグするために、ルート・ソース・ビューおよびデバッグ・データを作成します。DBG VIEW(*SOURCE) を指定しなかった場合、taxrate を参照するモジュールをデバッグできますが、そのデバッグ・セッション時に taxrate を表示できず、また taxrate を定義しているこのモジュールをデバッグすることもできません。

8. 以下のコマンドを入力して、ILE サービス・プログラム T1520SP1 からプロシージャー write_audit_trail() をエクスポートするためのバインダー言語が含まれたソース物理ファイル QSRVSRCS を作成します。

CRTC SCPF FILE(MYLIB/QSRVSRCS) MBR(T1520SP1)
注: 4.89 ページの図 4-58 に、MYLIB/QSVSRC のソース・コードを示します。
9. 以下 の コ マンド を入力して、ILE サービス・プログラム T1520SP2 からデータ 項目 taxrate をエクスポートするためのバインダー言語が含まれたソース物理ファイル QASRVSRC を作成します。
CRTSRCPF FILE(MYLIB/QASRVSRC) MBR(T1520SP2)
注: 4.90 ページの図 4-59 に、MYLIB/QASRVSRC のソース・コードを示します。
10. ライブラリー MYLIB にバインディング・ディレクトリー T1520BD1 を作成し、2 つの サービス・プログラム名 (T1520SP1 および T1520SP2) をそのディレクトリーに追加します。
a. バインディング・ディレクトリーを作成するには、以下のコマンドを入力します。
CRTBNDDIR BNDDIR(MYLIB/T1520BD1)
b. サービス・プログラム名を追加するには、以下のコマンドを入力します。
ADDBNDDIRE BNDDIR(MYLIB/T1520BD1) OBJ((MYLIB/T1520SP1 *SRVPGM))
ADDBNDDIRE BNDDIR(MYLIB/T1520BD1) OBJ((MYLIB/T1520SP2 *SRVPGM))
注: サービス・プログラムオブジェクトがまだ存在していないでも、サービス・プログラム名 T1520SP1 および T1520SP2 を追加できます。
注: これらの命令では、ライブラリー MYLIB が既に存在しているものと想定しています。
11. 以下のコマンドを入力して、モジュール T1520IC3 および QSRVSRC のバインダー・ソース言語から
ILE C サービス・プログラム T1520SP1 を作成します。
CRTSRVPAG SRVPG(MYLIB/T1520SP1) MODULE(MYLIB/T1520IC3 MYLIB/T1520IC4)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(*SRVPGM) BNDDIR(MYLIB/T1520BD1) DETAIL(*FULL)
注:
• 4.88 ページの『監査証跡を書き込むためのソース・コード』に、T1520IC3 のソース・コードを示し、4.89 ページの図 4-57 に、T1520IC4 のソース・コードを示します。
• サービス・プログラム T1520SP1 は、モジュール T1520IC3 とモジュール T1520IC4 の両方を必要とします。これは、このサービス・プログラムはプロシージャー write_audit_trail をエクスポートしてモジュール T1520IC1 での関数 write_audit_trail に対するインポート要求を満たし、write_audit_trail プロシージャーが、モジュール T1520IC4 で定義されているデータ項目 taxrate を使用するからです。
12. 以下のコマンドを入力して、モジュール T1520IC4 および QASRVSRC のバインダー・ソース言語から
ILE C サービス・プログラム T1520SP2 を作成します。
CRTSRVPAG SRVPG(MYLIB/T1520SP2) MODULE(MYLIB/T1520IC4) SRCFILE(MYLIB/QSRVSRC)
SRCMBR(*SRVPGM) BNDDIR(MYLIB/T1520BD1) DETAIL(*FULL)
注: サービス・プログラム T1520SP2 は、モジュール T1520IC2 の変数 taxrate のインポート要求を満たすためにデータ項目 taxrate をエクスポートします。
13. 以下のコマンドを入力して、コンポーネント T1520IC1 および T1520IC2 から ILE C プログラム
T1520PG1 を作成します。
CRTPGM PGM(MYLIB/T1520PG1) MODULE(MYLIB/T1520IC1 MYLIB/T1520IC2) ENTMOD(*ONLY)
BNDDIR(MYLIB/T1520BD1) DETAIL(*FULL)
注:
• 4.85 ページの図 4-55 に、T1520IC1 のソース・コードを示し、4.87 ページの図 4-56 に、T1520IC2 のソース・コードを示します。
• モジュール T1520IC1 には、シンボル write_audit_trail を使用している T1520SP1 のエクスポート要求に一致している、write_audit_trail という名前のプロシージャーのインポート要求が含まれています。バインダーは、以下を一致させます。
  - プロシージャー write_audit_trail に対する T1520IC1 からのインポート要求と T1520SP1 からの対応するエクスポート
  - データ項目 taxrate に対する T1520IC2 からのインポート要求と T1520SP2 からの対応するエクスポート
• ENTMOD(*ONLY) バラメーターは、モジュールのリストにある 1 つのモジュールだけが PEP を持つことができることを指定します。複数のモジュールが PEP を持っていることが検出された場合、エラーが発行されます。ENTMOD(*FIRST) バラメーターを使用した場合、モジュールのリストで検出された、PEP を持っている最初のモジュールが、PEP として選択されます。PEP を持つ他のすべてのモジュールは無視されます。
• 新規活動化グループを定義するために、デフォルトの ACTGRP(*NEW) バラメーターが使用されます。プログラム T1520PG1 は、新規活動化グループに関連付けられます。サービス・プログラム T1520SP1 および T1520SP2 は、活動化グループ *CALLER で作成されました。これらのサービス・プログラムは、1 つの連携プログラムを開発するために、1 つの活動化グループの資源を使用する呼び出し元活動化グループの一部になります。サービス・プログラム T1520SP1 および T1520SP2 は、活動化対象のプログラムにバインドされます。これらのサービス・プログラムも、動的呼び出し処理の一部として活動化されます。
14. 以下のように、プログラム T1520PG1 を実行します。

注: ライブラリー MYLIB が LIBL ライブラリー・リストに含まれているようにしてください。
  a. コマンド T1520CM1 を入力し、F4 (プロンプト) を押します。
  b. T1520CM からプロンプトが出されたら、以下のデータを入力します。

<table>
<thead>
<tr>
<th>Hammers</th>
<th>1.98</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>Nails</td>
<td>0.25</td>
</tr>
<tr>
<td>2000</td>
<td></td>
</tr>
</tbody>
</table>

出力は次のとおりです。

```
5000 HAMMERS plus tax = $11,385.00
実行キーを押して端末セッションを終了してください。
> 2000 NAILS plus tax = $575.00
実行キーを押して端末セッションを終了してください。
```

物理ファイル T1520DD1 が、以下のデータを使用して更新されます。

| SMITHE HAMMERS 0000000198500015 | $11,385.00072893 |
| SMITHE NAILS 0000000025200015   | $575.00072893    |

注: 各項目は、ユーザーの ID で識別されます。この場合、SMITHE がユーザーです。

**ソース・コードのサンプル**

このセクションの図には、ILE サンプル・アプリケーションを作成するために使用されるソース・コードが含まれています。
監査ログ・ファイルのソース・コード

監査ファイル T1520DD1 を 4-76 ページの図 4-50 に示しています。DDS ソースは、監査ファイルのフィールドを定義します。

R T1520DD1R
USER 10  COLHDG('User')
ITEM 20  COLHDG('Item name')
PRICE 10S 2  COLHDG('Unit price')
QTY 4S  COLHDG('Number of items')
TXRATE 2S 2  COLHDG('Current tax rate')
TOTAL 21  COLHDG('Total cost')
DATE 6  COLHDG('Transaction date')
K USER

図 4-52. 監査ファイル T1520DD1 の DDS ソース

ソース・コードによる端末セッション入力の ILE プログラムへの引き渡し

制御言語プログラム T1520CL1 を 4-76 ページの図 4-50 に示します。このプログラムは、必要なパラメータを ILE C プログラム T1520PG1 に渡します。

PGM  PARM(&ITEMIN &PRICE &QUANTITY)  1
DCL VAR(&USER) TYPE(*CHAR) LEN(10)
DCL VAR(&ITEMIN) TYPE(*CHAR) LEN(20)
DCL VAR(&ITEMOUT) TYPE(*CHAR) LEN(21)
DCL VAR(&PRICE) TYPE(*DEC) LEN(10 2)
DCL VAR(&QUANTITY) TYPE(*DEC) LEN(2 0)
DCL VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
/* ADD NULL TERMINATOR FOR THE ILE C PROGRAM */
CHGVAR VAR(&ITEMOUT) VALUE(&ITEMIN *TCAT &NULL)  2
/* GET THE USERID FOR THE AUDIT TRAIL */
RTVJOBA USER(&USER)  3
/* ENSURE AUDIT RECORDS WRITTEN TO CORRECT AUDIT FILE MEMBER */
OVRDBF FILE(T1520DD1) TOFILE(*LIBL/T1520DD1) +
MBR(T1520DD1) OVRSCOPE(*CALLLVL) SHARE(*NO)
CALL PGM(T1520PG1) PARM(&ITEMOUT &PRICE &QUANTITY +
&USER)
DLT0VR FILE(*ALL)
ENDPGM

図 4-53. T1520CL1 — ILE C プログラムに変数を渡すための CL ソース

注:
1. このプログラムは、品名、価格、数量、およびユーザー ID の CL 変数を（参照によって）ILE C プログラム T1520PG1 に渡します。制御言語プログラム内の変数が参照によって渡されると、ILE C プログラムは制御言語プログラム内の内容を変更できるようになります。
2. 変数 ITEMOUT は、制御言語プログラム T1520CL1 内でnull終了です。CL から ILE C に渡される CL 変数を引き渡しても、コンパイル済み CL 呼び出しで自動的にnull終了となるわけではありません。コンパイル済み CL 呼び出しおよびコマンド行 CL 呼び出しでのnull終了ストリングについては、8-45 ページの「ILE C/C++ 呼び出し規則の使用」を参照してください。
3. ジョブ属性検索 (RTVJOBA) コマンドは、監査証跡のユーザー ID を取得します。

セッション・データ収集用に CL コマンドを定義するためのソース・コード

CL CMD プログラム T1520CM1 を 4-76 ページの図 4-50 に示します。

この開発者定義コマンドは、以下のことを行います。
・Item name、Unit price、および Number of items の順序でユーザーに入力を求めるプロンプトを出します。
・入力データをキーワード・パラメーター ITEM、PRICE、および QUANTITY に保管します。

注: これらのキーワード・パラメーターは、DDS ファイル T1520DD1 に定義されていたものです。

<table>
<thead>
<tr>
<th>CMD</th>
<th>PROMPT('CALCULATE TOTAL COST')</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARM</td>
<td>KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +</td>
</tr>
<tr>
<td>PARM</td>
<td>MIN(I) ALWUNPRT(*NO) PROMPT('Item name' 1)</td>
</tr>
<tr>
<td>PARM</td>
<td>KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +</td>
</tr>
<tr>
<td>PARM</td>
<td>RANGE(0.01 99999999.99) MIN(I) +</td>
</tr>
<tr>
<td>PARM</td>
<td>ALWUNPRT(*YES) PROMPT('Unit price' 2)</td>
</tr>
<tr>
<td>PARM</td>
<td>KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +</td>
</tr>
<tr>
<td>PARM</td>
<td>9999) MIN(I) ALWUNPRT(*YES) +</td>
</tr>
<tr>
<td>PARM</td>
<td>PROMPT('Number of items' 3)</td>
</tr>
</tbody>
</table>

図 4-54. T1520CM1 を入力データを受け取るための CL コマンド・ソース

ユーザーエンタープロシージャー (UEP) のソース・コード

T1520IC1 のソース・コードでは、main() 関数は以下のことを行います。
・制御言語プログラムからユーザー ID、品名、数量、および価格を受け取ります。
・モジュール T1520IC2 内の calc_and_format() 関数を呼び出します。この関数は以下のことを行います。
  - 品目の合計コストを計算します。
  - モジュール T1520IC2 から write_audit_trail() 関数を呼び出します。この関数は、トランザクションを監査ファイルに書き込みます。
/* This program demonstrates how to use multiple modules, service */
/* programs and a binding directory. This program accepts a user ID, */
/* item name, quantity, and price, calculates the total cost, and */
/* writes an audit trail of the transaction. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>

int calc_and_format (decimal(10,2),
    short int,
    char[]);

void write_audit_trail (char[],
    char[],
    decimal(10,2),
    short int,
    char[]);

int main(int argc, char *argv[]){
    /* Incoming arguments from a CL program have been verified by */
    /* the *CMD and null ended within the CL program. */
    char *user_id;
    char *item_name;
    short int *quantity;
    decimal (10,2) *price;
    char formatted_cost[22];
    /* Incoming arguments are all pointers. */
    item_name = argv[1];
    price = (decimal (10, 2) *) argv[2];
    quantity = (short *) argv[3];
    user_id = argv[4];
    /* Call an ILE C function that returns a formatted cost. */
    /* Function calc_and_format returns true if successful. */
    if (calc_and_format (*price, *quantity, formatted_cost))
    {
        write_audit_trail (user_id,
            item_name,
            *price,
            *quantity,
            formatted_cost);
        printf("%d %s plus tax = %-s
", *quantity,
            item_name,
            formatted_cost);
    }
    else
    {
        printf("Calculation failed
");
    }
    return 0;
}

図4.55 他のモジュール内の関数を呼び出すための ILE C ソース

注:
1. このモジュールの main() 関数はユーザー入リロジシージャー (UEP) です。このプロシージャー
は、制御言語プログラム T1520CL1 からの動的プログラム呼び出しのターゲットです。 UEP は、プロ
グラム入リロジシージャー (PEP) から制御権を受け取ります。このモジュールには、コンパイル時に
ILE C コンパイラによって生成された PEP があります。 PEP は、制御言語プログラム T1520CL1
からの動的プログラム呼び出しにおける ILE C/C++ プログラムのエントリー・ポイントです。 PEP
は呼び出しスキャットに _C_pep として示されます。
2. このモジュールの main() 関数は、CL コマンド・プロンプト T1520CM1 によって検証される入力引数
を制御言語プログラム T1520CL1 から受け取ります。
3. すべての入力引数はポインタです。変数 item_name は制御言語プログラム T1520CL1 内でヌル終了します。

4. このモジュールの main() 関数は、フォーマットされたコストを戻すためにモジュール T1520IC2 内の calc_and_format を呼び出します。calc_and_format が成功を戻すと、サービス・プログラム T1520SP1 内の write_audit_trail によって監査証跡にレコードが書き込まれます。

5. 関数 write_audit_trail は、このモジュール (T1520IC1) に定義されていないので、インポートする必要があります。

税金を計算し、出力用にコストをフォーマットするためのソース・コード
モジュール T1520IC2 を4-77ページの図4-51に示します。このモジュールは calc_and_format() 関数を提供します。
/* This function calculates the tax and formats the total cost. */
/* The function calc_and_format() returns 1 if successful and 0 */
/* if it fails. */
#include <stdio.h>
#include <string.h>
#include <decimal.h>
/* Tax rate is imported from the service program T1520SP2. */
const extern decimal (2,2) taxrate;
int calc_and_format (decimal (10,2) price,
                     short int quantity,
                     char formatted_cost[22])
{
    decimal (17,4) hold_result;
    char hold_formatted_cost[22];
    int i,j;
    memset(formatted_cost, ' ', 21);
    hold_result = (decimal(4,0))quantity * price * (1.00D+taxrate); /* Calculate the total cost. */
    if (hold_result < 0.01D || hold_result > 1989800999801.02D)
    {
        printf("calc out of range:%17.4D(17,4)\n", hold_result);
        return(0);
    }
    /* Format the total cost. */
    sprintf(hold_formatted_cost, "%21.2D(17,4)", hold_result);
    j = 0;
    for (i=0; i<22; ++i)
    {
        if (hold_formatted_cost[i] != ' ' &
            hold_formatted_cost[i] != '0')
        {
            hold_formatted_cost[j] = '$';
            break;
        }
        j = i;
    }
    for (i=j=21; i>=0; --i)
    {
        if (j < 0) return(0);
        if (hold_formatted_cost[i] == '$')
        {
            formatted_cost[j] = hold_formatted_cost[i];
            break;
        }
        if (i<16 & !(i-2)%3)
        {
            formatted_cost[j] = ',';
            --j;
        }
        formatted_cost[j] = hold_formatted_cost[i];
        --j;
    }
    /* End of for loop, 21->0. */
    return(1);
}

図4.56 税金を計算し、出力用にコストをフォーマットするためのサンプル ILE C ソース

注:
1. このモジュールの関数 calc_and_format は合計コストを計算してフォーマットします。計算を行うため
に、データ項目 taxrate がサービス・プログラム T1520SP2 からインポートされます。このデータ項目
をインポートする必要があるのは、このモジュール (T1520IC2) に定義されていないためです。
This function writes an audit trail. To write the audit trail
/* the file field structure is retrieved from the DDS file */
/* T1520DD1 and the taxrate data item is imported from service */
/* program T1520SP2. Retrieves the file field structure. */
#pragma mapinc("myinc", "MYLIB/T1520DD1(*all)", "both", "p z",""
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <decimal.h>
#include <recio.h>
#include <xxcvt.h>
/* These includes are for the call to QWCCVTDT API to get */
/* the system date to be used in the audit trail. */
#include <QSYSINC/H/QWCCVTDT>
#include <QSYSINC/H/QUSEC>
/* DDS mapping of the audit file, T1520DD1. */
#include "myinc"
/* Tax rate is imported from service program T1520SP2. */
const extern decimal (2,2) taxrate;
void write_audit_trail (char user_id[10],
char item_name[],
decimal (10,2) price,
short int quantity,
char formatted_cost[22])
{
char char_item_name[21];
char char_price[11];
char temp_char_price[11];
char char_quantity[4];
char char_date[6];
char char_taxrate[2];
/* Qus_EC_t is defined in QUSEC. */
Qus_EC_t errcode;
char get_date[16];
int i;
double d;
/* File field structure is generated by the #pragma */
/* mapinc directive. */
MYLIB_T1520DD1_T1520DD1R_both_t buf1;
_RFILE *fp;
/* Get the current date. */
errcode.Bytes_Provided = 0;
QWCCVTDT ("*CURRENT ", "", "*MDY ", get_date, &errcode);
memcpy (char_date, &get_date[1]), 6);
/* Loop through the item_name and remove the null terminator. */
for (i=0; i<=20; i++)
{
if (item_name[i] == '\0') char_item_name[i] = ' ';
else char_item_name[i] = item_name[i];
}
/* Convert packed to zoned for audit file. */
d = price;
QXXDTOZ (char_price, 10, 2, d);
QXXITOZ (char_quantity, 4, 0, quantity);
d = taxrate;
QXXDTOZ (char_taxrate, 2, 2, d);
注:
1. このソースには、QSYSINC/H ファイルにある QUSEC と QWCCVTDT という 2 つのメンバーが必要です。CRTBNDC または CRTCMOD コマンドに OPTION(*STDC) バラメータ (デフォルト) が指定されている限り、システム組み込みファイルを見つけるために QSYSINC ライブラリが自動的に検索されます。
2. 組み込み名 myinc は、コンパイラーが #pragma mapinc ディレクティブの型定義を生成するときに作成する一時ソース・メンバーに関連付けられます。#pragma mapinc ディレクティブの使用法については、7-1 ページの「プログラムでの外部記述ファイルの使用」を参照してください。
3. 監査証跡を書き込むために、税率がサービス・プログラム T1520SP2 からインポートされます。

税率データをエクスポートするためのソース・コード
モジュール T1520IC4 は、サービス・プログラム T1520SP2 を作成するために使用されます。

```c
/* Export the tax rate data. */
#include <decimal.h>
const decimal (2,2) taxrate = .15D;
```

図 4-57. T1520IC4 — 税率データをエクスポートするための ILE C ソース

注: データ項目 taxrate は、別個のモジュールにコーディングされているため、以下の両方またはいずれかを使用してインポートすることができます。
- サービス・プログラム T1520IC2 内の calc_and_format 関数
- T1520IC3 内の write_audit_trail

税率データをエクスポートするためのバインダー言語

```c
STRPGMEXP PGMLVL(*CURRENT) EXPORT SYMBOL('taxrate')
ENDPGMEXP
```

図 4-58. 税率データをエクスポートするためのバインダー言語ソース

注:
1. プログラム・エクスポート開始 (STRPGMEXP) コマンドは、サービス・プログラム T1520SP1 からのエクスポートのリストの始まりを識別します。
2. プログラム記号のエクスポート (EXPORT) コマンドは、サービス・プログラム T1520SP1 からエクスポートされる記号名 taxrate を識別します。
3. 記号名 taxrate は、STRPGMEXP PGMLVL(*CURRENT) と ENDPGMEXP のペアの間で識別され、サービス・プログラム T1520SP2 に対するパブリック・インターフェースを定義します。ILE C サービス・プログラムを構成するモジュール・オブジェクトからエクスポートされるプロシージャーおよびデータ項目のみが、このサービス・プログラムからエクスポートすることができます。  

4. 記号名 taxrate は、小文字形式を維持するためにアポストロフィで囲まれています。アポストロフィがないと、大文字に変換されます。(バイナダーは、見つからないことがない TAXRATE を検索してしまいます。)

5. 記号名 taxrate は、シグニチャーを作成するためにも使用されます。このシグニチャーは、活動化時にサービス・プログラム T1520SP2 に対するパブリック・インターフェースを妥当性検査します。これにより、ILE C サービス・プログラム T1520SP1 および ILE C サービス・プログラム T1520PG1 は、再作成することなくサービス・プログラム T1520SP2 を使用できるようになります。

write-audit-trail プロシージャーをエクスポートするためのバイナダー言語

```
STRPGMEXP PGMLVL(*CURRENT) EXPORT SYMBOL('write_audit_trail')
ENDPGMEXP
```

図4.59. write_audit_trail プロシージャーをエクスポートするためのバイナダー言語ソース

注:

1. プログラム・エクスポート開始 (STRPGMEXP) コマンドは、サービス・プログラム T1520SP2 からのエクスポートのリストの始まりを識別します。
2. プログラム記号のエクスポート (EXPORT) コマンドは、サービス・プログラム T1520SP2 からエクスポートされる記号名 write_audit_trail を識別します。
3. 記号名 write_audit_trail は、STRPGMEXP PGMLVL(*CURRENT) と ENDPGMEXP のペアの間で識別され、サービス・プログラム T1520SP2 に対するパブリック・インターフェースを定義します。ILE C サービス・プログラムを構成するモジュール・オブジェクトからエクスポートされるプロシージャーおよびデータ項目のみが、このサービス・プログラムからエクスポートすることができます。パブリック・インターフェースを制御できない場合には、実行時エラーまたは活動化エラーが発生する可能性があります。

4. 記号名 write_audit_trail は、小文字形式を維持するためにアポストロフィで囲まれています。アポストロフィがないと、大文字に変換されます。(バイナダーは、見つからないことがない WRITE_AUDIT_TRAIL を検索してしまいます。)
5. 記号名 write_audit_trail は、シグニチャーを作成するためにも使用されます。このシグニチャーは、活動化時にサービス・プログラム T1520SP2 に対するパブリック・インターフェースを妥当性検査します。これにより、ILE C サービス・プログラム T1520SP1 および ILE C サービス・プログラム T1520PG1 は、再作成することなくサービス・プログラム T1520SP2 を使用できるようになります。
プログラムのデバッガ

このトピックでは、以下を行うための方法について説明します。

- ILE ソース・デバッガー (そのオプション、言語構文、およびコマンド) を使用する
- デバッグ・データが含まれたモジュールをプログラムにバインディングし、デバッグ用のリスト・ビューを作成する
- デバッグ・データを組み込むようにプログラムを準備してコンパイルする
- デバッグ・セッションを使用する
- デバッグを支援するために停止点を使用する
- デバッグを支援するためにウォッチを使用する
- プログラムをステップスルーする
- 変数をデバッグする

ILE ソース・デバッガー

ILE ソース・デバッガーは、ILE C/C++ プログラムおよびサービス・プログラムのプログラミング・エラーを見つけるのに役立ちます。

ILE ソース・デバッガーを使用する前に、ソース・ファイルのコンパイル時にいずれかの非デフォルト・デバッガー・オプション (DBGVIEW) を使用しておく必要があります。停止点または他の ILE ソース・デバッガー・オプションを設定した後に、デバッグ・セッションを開始して、プログラムを呼び出すことができます。

このトピックでは以下について説明します。

- デバッグ・データ・オプション
  - デバッグ言語構文およびその制限
  - デバッグ・コマンド
  - ILE ソース・デバッガーおよび ILE C アプリケーションで実行できる内容を示すデータ定義例

デバッグ・データ・オプション

モジュールに関連付けることができるデバッグ・データのタイプは、デバッグ・ビューとして参照されます。

含まれるデバッグ・データのタイプによって、モジュールまたはプログラムのストレージ要件が多少異なります。デバッグ・オプションを以下にリストします。2 次ストレージ要件は、リストの下にいくほど増大します。

1. DBGVIEW(*NONE) (デバッグ・データなし)
2. DBGVIEW(*STMT) (ステートメント・ビュー)
3. DBGVIEW(*SOURCE) (ソース・ビュー)
4. DBGVIEW(*LIST) (リスト表示ビュー)
5. DBGVIEW(*ALL) (すべてのビュー)
デバッグ言語構文
C デバッグ式文法の制約には、以下のものがあります。
・タイプ・キャスト: 配列および関数指定子のタイプ・キャストは禁止されています。
・関数呼び出し: 関数呼び出ちは、デバッグ式で使用できません。
・10 進数タイプ: 10 進数タイプは、表示用にのみサポートされます。デバッグ式で使用することはできません。

演算子の優先順位および混合タイプのタイプ変換は、ISO C 標準に準拠しています。詳しくは、「ILE C/C++ 解説書」を参照してください。

デバッグ言語構文の制約
ILE ソース・デバッガーには、以下の制約があります。
・デバッグ式で関数呼び出しを使用することはできません。これは、デバッグ式文法の制約です。
・演算子の優先順位および混合タイプのタイプ変換は、C および C++ 言語標準に準拠しています。
・表示可能な変数の最大サイズは 65535 文字です。
  - :c および :x フォーマット設定オーバーライドでは、カウントが入力されていない場合、コマンドは1 バイト後に停止します。
  - :s フォーマット設定オーバーライドでは、カウントが入力されていない場合、コマンドは 30 バイト後または NULL のうち早い方の後に停止します。
  - :f フォーマット設定オーバーライドでは、カウントが入力されていない場合、コマンドは 1024 バイト後または NULL のうち早い方の後に停止します。
・単一のコンパイル単位で仮想基底クラスとして継承できるクラスの最大数は 512 です。

デバッグ・コマンド
ILE ソース・デバッガーとともに多数のデバッグ・コマンドを使用できます。
例えば、デバッグ・コマンド行で break 10 を入力し、行 10 がデバッグ可能ステートメントである場合、デバッガーは、ソースの行 10 に無条件停止点を追加します。

注:
1. 行 10 がプランク行またはコメント・ステートメントである場合、デバッガーはエラーを出します。
2. 行 10 が、typedef ステートメントなど、デバッグ可能ステートメントではない場合は、直後のデバッグ可能ステートメントに停止点を設定します。
3. (カーソルがデバッグ可能コマンド行上にあるときに) F6 キーを押すと、停止点が設定またはクリアされます。

以下のいずれかのデバッグ・オプションを使用してモジュールをコンパイルした場合、デバッグ・データが作成されます。
  • *STMT
  • *SOURCE
  • *LIST
  • *ALL
デバッグ・コマンドおよびそのパラメーターは、「モジュール・ソースの表示」画面または「評価式」画面の下部に表示されたデバッグ・コマンド行に入力します。大文字、小文字、または大小文字混合で入力できます。

オンライン情報デバッグ・コマンドの説明があり、許可される省略形が示されています。

デバッグ・コマンドは以下のとおりです。

**ATTR** 変数の属性を表示します。属性は、デバッグ・シンボル・テーブルに記録された変数のサイズおよびタイプです。

**BREAK**
テストするプログラムの特定の場所へ条件付きまたは無条件のジョブ停止点を入れられるようにします。条件付きジョブ停止点を入力するには、BREAK 行番号 WHEN 式を入力します。

**CLEAR**
条件付きまたは無条件の停止点を除去します。1つまたはすべての活動中のウォッチ条件を除去します。

**DISPLAY**
Equate コマンドを使用して割り当てた名前と定義を表示します。また、現在「モジュール・ソースの表示」画面に示されているのは別のソース・モジュールを表示することができます。モジュール・オブジェクトは現行のプログラム・オブジェクト中に存在していないければなりません。

**EQUATE**
式、変数、またはデバッグ・コマンドに短縮名を割り当てます。

**EVAL** 変数の値を表示または変更します。式、レコード、構造体、または配列の値を表示します。

**QUAL** 後続の EVAL または WATCH コマンドに現れる変数のスコープを定義します。

**SET** 実動ファイルの更新機能などのデバッグ・オプションを変更します。検索操作で大小文字の区別を行うかどうかを指定します。OPM ソース・デバッグ・サポートを使用可能にします。

**STEP** デバッグするプロシージャーの 1つ以上のステートメントを実行します。

**TBREAK**
テスト中のプログラムで、現行スレッドの特定の位置に条件付きまたは無条件のいずれかの停止点を入力することができます。

**THREAD**
「デバッグ済みスレッドの処理」画面を開くか、現行スレッドを変更できます。

**WATCH**
指定した保管場所の内容が現行値から変更された時点で停止点を要求します。

**FIND** 現在表示されているモジュールで、指定した行番号またはテキスト・ストリングを検索します。テキスト検索では、表示されているビュー・テキスト上のカーソルの位置から前方または後方への方向を指定できます。カーソルがビュー・テキスト上にない場合は、検索は、現行画面上のテキストの先頭行の先頭位置から開始されます。ストリングが見つかった場合、カーソルの位置は、見つかったストリングの先頭文字になります。

F16 検索コマンド・キーを使用することで、入力された最後の Find コマンドを繰り返すことができます。

**UP** 表示されているソースのウィンドウを入力した行数だけ、ビューの先頭の方に移動します。

**DOWN** 表示されているソースのウィンドウを入力した行数だけ、ビューの末尾の方に移動します。
ILE C プログラムでのデバッガ式の使用例

プログラミングのエラーを診断および修正するために使用できる ILE C ソース・コードおよび ILE デバッガ式の例を示します。

例では、次のようにになっています。

- 評価する ILE C ソース・コードは、図内に示されています。
- ILE ソース・デバッガー・セッションで示される情報は、画面の図に示されています。各画面で、EVAL デバッガー・コマンドおよびそれによって取得および表示される情報を示しています。

プログラム定義および対応するデバッガ式の例

以下の ILE C ソース・コードには、ポインタのデータ定義、単純変数、構造、共用体、および列挙が含まれています。ILE ソース・デバッガーおよび ILE C アプリケーションによって何が可能であるかを示すために、以下のセクションに、対応するデバッガ式および情報を示します。

- 5-5 ページの「エラーを検出および修正するためのポインターの評価」
- 5-7 ページの「エラーを検出および修正するためのポインターの評価」
- 5-7 ページの「エラーを検出および修正するためのビット・フィールドの評価」
- 5-8 ページの「エラーを検出および修正するための構造体および共用体の評価」
- 5-9 ページの「エラーを検出および修正するための列挙型の評価」

#include <stdio.h>
#include <decimal.h>
#include <pointer.h>
/** POINTERS **/
_SYSPTR pSys; /* System pointer */
_SPCPTR pSpace; /* Space pointer */
int (*fncptr)(void); /* Function pointer */
char *pc1; /* Character pointer */
char *pc2; /* Character pointer */
int *pi1; /* Integer pointer */
char arr1[] = "ABC"; /* Array */
/** SIMPLE VARIABLES **/
int i1; /* Integer */
unsigned u1; /* Unsigned Integer */
char c1; /* Character */
float f1; /* Float */
.Decimal(3,1) dec1; /* Decimal */
/** STRUCTURES **/
struct {
  int b1 : 1;
  int b4 : 4;
}
struct x { /* Tagged structure */
    int x;
    char *p;
};
struct y { /* Structure with */
    int y; /* structure member */
    struct x x;
};
typedef struct z { /* Structure typedef*/
    int z;
    char *p;
} z;
z *pZZ;
/* Same */
typedef struct c { /* Structure typedef */
    unsigned a;
    char *b;
} c;
c d;
/** UNIONS **/
union u{ /* Union */
    int x;
    unsigned y;
};
union u u;
union u *pU;
/** ENUMERATIONS **/
enum number {one, two, three};
enum color {red, yellow, blue};
enum number number = one;
enum color color = blue;
/** FUNCTION **/
int ret100(void) { return 100; }
main(){
struct y y, *pY;
bits.b1 = 1;
bits.b4 = 2;
i1 = ret100();
c1 = 'C';
f1 = 100e2;
dec1 = 12.3;
pc1 = &c1;
pi1 = &i1;
d.a = 1;
pZZ = &zz;
pZZ->z=1;
pY = &y;
pY->x.p=(char *)&y;
pU = &u;
pU->x=255;
number=color;
fncptr = &ret100;
pY->x.x=1; /* Set breakpoint here */
}
int main(void) { .......
return(0); }

エラーを検出および修正するためのポインターの評価： ポインター 型変数は、データ・オブジェクトまたは関数のアドレスを保持します。ポインターは、(ビット・フィールドまたは参照を除く) 任意の 1 つのデータ・タイプのオブジェクトを参照できます。ポインターは、一度に 1 つの値しか保持できないという意味で、スカラー型として分類されます。
プログラムでのポインタの使用については、以下を参照してください。

- 8-37 ページの『プログラムでのポインタの使用』
- 4-67 ページの『ポインタを使用したパフォーマンス向上』
- ILE C/C++ 解説書

EVAL デバッグ・コマンドを使用して、ポインタ変数または配列の値を表示または変更します。複数の行応答が含まれたメッセージは、「評価式」画面を起動します。この画面には、入力したデバッグ・コマンドの履歴およびそのコマンドの結果も表示されます。「モジュール・ソースの表示」画面に戻るには、Enter キーを押します。画面間のトグル・スイッチとして Enter キーを使用できます。

注: 単一行の応答は、「モジュール・ソースの表示」のメッセージ行に表示されます。

以下の図に、ポインタのデバッグ式の例を示します。

```
前のデバッグ式
> eval pcl
   pcl = SPP:C0260900107C0000
> eval pc2=pcl
   pc2=pcl = SPP:C0260900107C0000
> eval *pcl
   *pcl = ’C’
> eval &pc1
   &pc1 = SPP:C026090010400000
> eval *&pc1
   *&pc1 = SPP:C0260900107C0000
> eval *(short *)pc1
   *(short *)pc1 = -15616

評価式
Displaying pointers
Assigning pointers
Dereferencing pointers
Taking an address
Can build expressions with	normal C precedence
Casting

デバッグ . . 終了
```

図 5-1. デバッグ・セッションでのポインタの使用例 (画面 1)

```
前のデバッグ式
> eval arr1
   arr1 = SPP:C026090010700000
> eval *arr1
   *arr1 = ’A’
> eval *arr1:s
   *arr1:s = ”ABC”
> eval pcl=0
   pcl=0 = SYP:*NULL
> eval fncptr
   fncptr = PRP:A0CD0004F0100000
> eval *pY->x.p
   *pY->x.p = ’1’

評価式
Unqualified arrays are treated
as pointers
Dereferencing applies the array type
(character in this example)
If the expression is an lvalue
you can override the formatting
Setting a pointer to null by assigning 0
Function pointers
Using the arrow operator

デバッグ . . 終了
```

図 5-2. デバッグ・セッションでのポインタの使用例 (画面 2)
エラーを検出および修正するための単純式の評価: 式は、計算を指定する演算子、オペランド、および区切り子のシーケンスです。式に含まれている演算子と、それらの演算子が使われるコンテキストに基づいて式の評価が行われます。

プログラムでの式および演算子の使用については、以下を参照してください。
- 8-37 ページの『プログラムでのポインターの使用』
- 8-62 ページの『BCD マクロを使用してコード化 10 進数オブジェクトを ILE C++ に移植する』
- 8-126 ページの『C プログラムでのパック 10 進数データの使用』
- 8-139 ページの『C++ プログラムでのパック 10 進数データの使用』
- ILE C/C++ 解説書

EVAL デバッグ・コマンドを使用して、単純式または演算子の値を表示または変更します。複数の行応答が含まれたメッセージは、「評価式」画面を起動します。この画面には、入力したデバッグ・コマンドの履歴およびそのコマンドの結果も表示されます。「モジュール・ソースの表示」画面に戻るには、Enter キーを押します。画面間のトグル・スイッチとして Enter キーを使用できます。

注: 単一行の応答は、「モジュール・ソースの表示」のメッセージ行に表示されます。

以下の図では、単純演算 (例えば、代入演算、算術演算、関係演算) のデバッグ式の例を示します。

<table>
<thead>
<tr>
<th>評価式</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;eval i1==u1</td>
<td></td>
</tr>
<tr>
<td>i1==u1</td>
<td></td>
</tr>
<tr>
<td>&gt;eval i1++</td>
<td>Unary operators occur in proper order</td>
</tr>
<tr>
<td>i1++ = 100</td>
<td></td>
</tr>
<tr>
<td>&gt;eval i1</td>
<td>Increment has happened after i1 was used</td>
</tr>
<tr>
<td>i1 = 101</td>
<td></td>
</tr>
<tr>
<td>&gt;eval ++i1</td>
<td>Increment has happened before i1 was used</td>
</tr>
<tr>
<td>++i1 = 102</td>
<td></td>
</tr>
<tr>
<td>&gt;eval u1 = -10</td>
<td>Implicit conversions happen</td>
</tr>
<tr>
<td>u1 = -10</td>
<td>4294967286</td>
</tr>
<tr>
<td>&gt;eval (int)u1</td>
<td></td>
</tr>
<tr>
<td>(int)u1 = -10</td>
<td></td>
</tr>
<tr>
<td>&gt;eval dec1</td>
<td>Decimal types are displayed but cannot</td>
</tr>
<tr>
<td>decl = 12.3</td>
<td>be used in expressions</td>
</tr>
<tr>
<td>終り</td>
<td></td>
</tr>
</tbody>
</table>

図 5.3. デバッグ式で使用されている単純演算の例

エラーを検出および修正するためのビット・フィールドの評価: C および C++ では、両方とも、コンパイラが通常許容するよりも小さなメモリ・スペースに整数メンバーを保管することができます。このようなスペース節約構造体メンバーはビット・フィールドと呼ばれ、その幅はビット数で示示的に宣言することができます。ビット・フィールドは、データ構造を固有のハードウェア表現に対応させなければならない、移植できそうにないプログラムで使用します。

ビット・フィールドの使用については、以下を参照してください。
- 4-57 ページの『パフォーマンスを改善するためのデータ・タイプの選択』
- ILE C/C++ 解説書

EVAL デバッグ・コマンドを使用して、ビット・フィールドの値を表示または変更します。複数の行応答が含まれたメッセージは、「評価式」画面を起動します。この画面には、入力したデバッグ・コマンドの履

プログラムのデバッグ 5-7
歴およびそのコマンドの結果も表示されます。「モジュール・ソースの表示」画面に戻るには、Enter キーを押します。画面中のトグル・スイッチとして Enter キーを使用できます。

注: 単一行の応答は、「モジュール・ソースの表示」のメッセージ行に表示されます。

以下の図に、ビット・フィールドのデバッグ式の例を示します。

![評価式](image)

エラーを検出および修正するための構造体および共用体の評価: 構造体 は、class_key struct を使用して宣言されたクラスです。構造体のメソッドおよび基底クラスは、デフォルトにより public になります。

共用体 は、class_key union を使用して宣言されたクラスです。共用体のメンバーは、デフォルトにより public になります。共用体は、一時的に 1 つのデータ・メンバーのみ保持します。

以下の場面を除いて、構造体タイプと共用体タイプ間の変換は許可されません。C では、右オペランドのタイプが左オペランドのタイプと互換性がある場合は、互換性のある構造体タイプまたは共用体タイプ間での代入変換が許可されます。

プログラムでの構造体および共用体の使用については、以下を参照してください。

- 8-58 ページの「別のプラットフォームから ILE へのプログラムの移植」
- 8-62 ページの「BCD マクロを使用してコード化 10 進数オブジェクトを ILE C++ に移植する」
- 7-1 ページの「プログラムでの外部記述ファイルの使用」
- 11-1 ページの「言語間のデータ・タイプ互換性」
- ILE C/C++ 解説書

EVAL デバッグ・コマンドを使用して、構造体または共用体の値を表示または変更します。複数の行応答が含まれたメッセージは、「評価式」画面を起動します。この画面には、入力したデバッグ・コマンドの履歴およびそのコマンドの結果も表示されます。「モジュール・ソースの表示」画面に戻るには、Enter キーを押します。画面中のトグル・スイッチとして Enter キーを使用できます。

注: 単一行の応答は、「モジュール・ソースの表示」のメッセージ行に表示されます。
以下の図に、構造体および共用体のデバッグ式の例を示します。

評価式

前のデバッグ式
>eval (struct z *)&zz
(struct z *)&zz = SPP:C005AA001000000000 You can cast with typedefs
>eval *(c *)&zz
*(c *)&zz.a = 1 You can cast with tags
*(c *)&zz.b = SYP:*NULL
>eval u.x = -10
u.x = -10 = -10 You can assign union members
>eval u
u.y = 4294967286 You can display and the union will be formatted for each definition
u.x = -10

デバッグ . . 終わり

F3= 終了 F9= コマンドの複写 F12= 取り消し F16= 検索の繰り返し F19= 左
F20= 右 F21= コマンド入力 F23= 出力の表示

図 5.5 デバッグ式での構造体および共用体の使用例

エラーを検出および修正するための列挙型の評価：列挙型は、名前付き整数定数である値のセットから構成されているデータ型です。これは、値ごとに名前を作成するときにそれぞれの値をリスト（列挙）しなければならないため、列挙された型とも呼ばれます。列挙型の名前付き値は、列挙型定数とも呼ばれます。列挙型は、整数定数のセットを定義しグループ化する方法を提供するのに加え、少数の可能な値を持つ変数にとって役立ちます。プログラムでの列挙型の使用については、以下を参照してください。

• [11-1 ページの『言語間のデータ・タイプ互換性』
• ILE C/C++ 解説書

以下の図では、列挙型のデバッグ式を示します。
ILE ソース・デバッガーでのシステム・ポインターおよびスペース・ポインターの表示例

ILE ソース・デバッガーでのシステム・ポインターおよびスペース・ポインターの表示例

評価式

前のデバッグ式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

デバッグ...

終わり

評価式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

デバッグ...

終わり

評価式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

デバッグ...

終わり

評価式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

デバッグ...

終わり

評価式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

デバッグ...

終わり

評価式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

デバッグ...

終わり

評価式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

評価式

前のデバッグ式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

デバッグ...

終わり

評価式

前のデバッグ式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

デバッグ...

終わり

評価式

前のデバッグ式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

デバッグ...

終わり

評価式

前のデバッグ式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

デバッグ...

終わり

評価式

前のデバッグ式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

デバッグ...

終わり

評価式

前のデバッグ式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

デバッグ...

終わり

評価式

前のデバッグ式

> eval color
color = blue (2)
> eval number
number = three (2)
> eval (enum color)number
(enum color)number = blue (2)
> eval number = one
number = one (2)
> eval number = three
number = three = three (2)
> eval arrl[one]
arrl[one] = 'A'

Both the enumeration and its value are displayed
Casting to a different enumeration
Assigning by number
Assigning by enumeration
Using in an expression

DE デバッガーでの列挙型の使用例

ILE ソース・デバッガーでのシステム・ポインターおよびスペース・ポインターの表示例

5-11 ページの図 5-7 の ILE C ソース・コードは、デバッガーによってシステム・ポインターおよびスペース・ポインターをどのように表示できるかの例としてそれらのポインターをセットアップします。対応するデバッグ式および情報は 5-12 ページの図 5-8 に示しています。
#include <stdio.h>
#include <miscpace.h>
#include <pointer.h>
#include <miscpobj.h>
#include <except.h>
#include <lecond.h>
#include <leenv.h>
#include <qtedbgs.h> /* From qsysinc */

/* Link up the Create User Space API */
#pragma linkage(CreateUserSpace,OS)
#pragma map(CreateUserSpace,"QUSCRTUS")
void CreateUserSpace(char[20],
    char[16],
    long int,
    char,
    char[10],
    char[10],
    char[50],
    _TE_ERROR_CODE_T *
);

/* Link up the Delete User Space API */
#pragma linkage(DeleteUserSpace,OS)
#pragma map(DeleteUserSpace,"QUSDLTUS")
void DeleteUserSpace(char[20],
    _TE_ERROR_CODE_T *
);

/* Link up the Retrieve Pointer to User Space API */
#pragma linkage(RetrievePointerToUserSpace,OS)
#pragma map(RetrievePointerToUserSpace,"QUSRUTIL")
void RetrievePointerToUserSpace(char[20],
    char **,
    _TE_ERROR_CODE_T *
);

int main (int argc, char *argv[])
{
    char *pBuffer;
    _SYSPTR pSYSptr;
    _TE_ERROR_CODE_T errorCode;
    errorCode.BytesProvided = 0;
    CreateUserSpace("QTEUSERSPCQTEMP ",
        "QTESSPC ",
        10,
        0,
        "ALL ",
        ",
        "YES ",
        &errorCode
    );

    RetrievePointerToUserSpace("QTEUSERSPCQTEMP ",
        pBuffer,
        &errorCode);

    /* convert the space pointer to a system pointer */
pSYSptr = _SETSPFP(pBuffer);

    printf("Space pointer: %p\n",pBuffer);
    printf("System pointer: %p\n",pSYSptr);

    return 0;
}

図 5-7. ILE C ソース・コードでのシステム・ポインターおよびスペース・ポインター

以下の図は、デバッガーによるシステム・ポインターおよびスペース・ポインターの表示方法を示しています。

プログラムのデバッグ  5-11
デバッグのためのプログラムの準備
このトピックでは、以下のための方法について説明します。

- 実動ファイルを変更しないようにするためにテスト・ライブラリをセットアップする。
- デバッグ・データが含まれたモジュールをプログラムにバインドし、デバッグ用のリスト・ビューを作成する。

ILE ソース・デバッガーを使用するには、その前にプログラムにデバッガー・データが含まれていないか確認が必要。その後、デバッグ・セッションを開始できます。ブレーキポイントまたは他の ILE ソース・デバッガー・オプションを設定した後に、プログラムを呼び出すことができます。

注: ブレーキポイントまたは他の ILE ソース・デバッガー・オプションの設定については、5-20ページの「停止点を使用したデバッグの支援」を参照してください。

テスト・ライブラリのセットアップ
重要: プログラムのデバッグおよびテスト中、既存の実動データが影響を受けることもないように、テスト・オブジェクトの入っているテスト・ライブラリをプログラムに設定するようにライブラリ・リストを変更してください。

実動ライブラリが異なる場合、以下のいずれかの変更を行います。
- UPDPROD パラメーターを *NO (デフォルト) に設定して、デバッグ開始 (STRDBG) を使用する。
- UPDPROD パラメーターを *NO (デフォルト) に設定して、デバッグ変更 (CHGDBG) を使用する。
- 「モジュール・ソースの表示」画面で SET UPDPROD NO を指定する。

注: UPDPROD パラメーターを (*YES) に設定してデバッグ・セッションを開始した場合、デバッグ・セッションで、実動ライブラリのオブジェクトにアクセスできるようになります。

デバッグのためのリスト・ビューの作成
リスト・ビュー には、コンバイラーによって生成されたコンバイラー・リストのテキストと類似のテキストが含まれています。デバッグ用のリスト・ビューを作成するには、モジュールまたはプログラムの作成時に DBGVIEW(*LIST) オプションを使用します。

注:
1. ILE C コンパイラは、リストのセクション内の行をモジュールにコピーすることで、リスト・ビューを作成します。

2. ILE C++ コンパイラは、該当するソース・ファイルのテキストをモジュールにコピーすることで、リスト・ビューを作成します。

リスト・ビューは、その基盤のソース・ファイルにリンクされていません。

例えば、ソース・ファイル myfile.cpp から作成されたプログラムをデバッグするためにリスト・ビューを作成するには、以下を入力します。

CRTBNDCPP MYFILE SRCSTMF('/home/myfile.cpp') DBGVIEW(*LIST)

注: リスト・ビューの行の最大長は、255 文字です。

ソース・デバッグ・セッションの処理

ILE ソース・デバッガーを使用して、ILE C/C++ プログラムおよびサービス・プログラムのプログラミング・エラーを見つけます。

ILE ソース・デバッガーを使用する前に、ソース・ファイルのコンパイル時にいずれかの非デフォルト・デバッグ・オプション (DBGVIEW) を使用しておく必要があります。次に、デバッグ・セッションを開始できます。停止点または他の ILE ソース・デバッガー・オプションを設定した後に、プログラムを呼び出すことができます。

このトピックでは、以下を行うための方法について説明します。

・ デバッグ・セッションを開始する
・ デバッグ・セッションでプログラムを追加および除去する
・ デバッグ・オプションを設定および変更する
・ デバッグ・セッションからプログラム・ソースを表示する

ソース・デバッグ・セッションの開始

ILE ソース・デバッガーを開始するには、デバッグ開始 (STRDBG) コマンドを使用します。デバッガーを一度開始すると、デバッガーの終了 (ENDDBG) コマンドを入力するまで活動化されたままです。

注: STRDBG コマンドを使用するための *USE オブジェクト権限と、デバッグするオブジェクトの *CHANGE 権限が必要です。

最初に、STRDBG コマンドのプログラム (PGM) バラメーターを使用して最大 20 個のプログラムをデバッグ・セッションに追加することができます。 OPM プログラムまたは ILE プログラムを任意に組み合わせて追加できます。 ILE ソース・デバッガーを使用してそれらのすべてをデバッグできるかどうかは、以下によって決まります。

・ OPM プログラムがコンパイルされた方法
・ デバッグ環境設定

また、サービス・プログラム (SRVPGM) バラメーターを指定した STRDBG コマンドを使用することにより、サービス・プログラムを 20 個までデバッグ・セッションに追加することもできます。

ILE ソース・デバッガーを使用して ILE C/C++ プログラムまたはサービス・プログラムをデバッグするには、モジュールまたはプログラムが作成されたときに有効なデバッグ・ビューが指定されていなければなりません。有効なデバッグ・ビューには、*SOURCE、*LIST、*STMT、または *ALL が含まれます。
デバッグするモジュールごとに、複数のビューを作成することができます。それらは次のとおりです。

- ルート・ソース・ビュー
  ルート・ソース・ビューは、ルート・ソース・ファイルからのテキストを含みます。このビューに、インクルード・ファイルの展開は含まれません。

モジュールまたはプログラムの作成時に、それぞれ CRTCMOD/CRTCPPMOD または CRTBNDC/CRTBNDCPP コマンドの DBGVIEW パラメーターに *SOURCE または *ALL オプションを使用することによってルート・ソース・ビューを作成できます。

ILE C/C++ コンパイラは、モジュール・オブジェクト (*MODULE) のコンパイル中に、ルート・ソース・ビューを作成します。ルート・ソース・ビューは、ファイルのテキストをビューにコピーするのでなく、ルート・ソース・ファイルのテキストの位置の参照を使用して作成されます。このため、これら のファイルのモジュール作成とこれらのファイルから作成されたモジュールのデバッグの間で、ルート・ソース・ファイルの変更、名前変更、あるいは移動をしてはなりません。

- ステートメント・ビュー
  これは、C または C++ のコマンドで DBGVIEW パラメーターで *STMF オプションを指定してソースをコンパイルした場合に得られるビューです。これにより、リストで参照可能なステートメント番号とシンボリック ID を使用してプログラムをデバッグすることができます。

モジュールまたはプログラムの作成時に、それぞれ CRTCMOD/CRTCPPMOD または CRTBNDC/CRTBNDCPP コマンドの DBGVIEW パラメーターに *STMF オプションを使用することによってステートメント・ビューを作成できます。

- インクルード・ソース・ビュー
  インクルード・ソース・ビューには、ルート・ソース・ファイルからのテキストのほか、ソースのテキストに展開されるすべてのインクルード・ファイルのテキストも含まれています。このビューに、マクロの展開は含まれません。このビューを使用すると、ルート・ソース・ファイルとすべてのインクルード・ファイルをデバッグすることができます。

DBGVIEW パラメーターの *SOURCE または *ALL オプション、および OPTION パラメターの *SHOWINC を使用して、インクルード・ソース・ビューを作成してモジュールをデバッグできます。

- リスト・ビュー
  リスト・ビュー には、OUTPUT() コマンド・パラメターで指定され、ILE C/C++ コンパイラによって生成されたコンパイラ・リストと類似のテキストが含まれます。

モジュールのコンパイル時に *LIST または *ALL オプションを使用することで、リスト・ビューを作成してモジュールをデバッグできます。また、表示するリスト・ビューに応じて、OPTION パラメターに *SHOWINC、*SHOWUSR、*SHOWSYS、および *SHOWSKP の少なくとも 1 つを指定することもできます。

注: *SHOWSKP は ILE C のみに有効です。

STRDDBG コマンドで指定した最初のプログラムが表示されるのは、そのプログラムにデバッグ・データがある場合です。ILE プログラムの場合、デバッグ・データがあれば呼び出されるモジュールが表示され、そうでない場合は、デバッグ・データを持つ ILE プログラムにバイナリで作成された最初のモジュールが表示されます。ILE ソース・デッガーを使用して OPM プログラムをデバッグするには、次の条件が満たされているければなりません。

- プログラムが OPM RPG または COBOL プログラムの場合、それが OPTION(*LSTDBG) でコンパイルされた。
プログラマが OPM 制御言語プログラムの場合、それが OPTION(*SRCDBG) でコンパイルされた。
ILE デバッグ環境が、OPM プログラムを受け入れるように設定されている。STRDBG コマンドに OPM SRC(*YES) を指定すればこれを設定することができます（システムのデフォルト値は OPM SRC(*NO) です）。

これら 2 つの条件が満たされないときは、OPM システム・デバッグ器を使用して OPM プログラムをデ

バッグしてください。

OPM プログラム RPGPGM を呼び出すサンプル・デバッグ・プログラム DEBUGEX のデバッグ・セッショ

ンを開始するには、次のように入力します。

STRDBG PGM(MYLIB/DEBUGEX MYLIB/RPGPGM) OPM SRC(*YES)

DBGVIEW(*NONE) がデフォルトの DBGVIEW オプションです。モジュールの作成時にデバッグ・データ

は作成されません。

デバッグ・データまたはデバッグ・ビューを持つモジュールを作成し、それをプログラム・オブジェクト

(*PGM) にバインドした後で、プログラムのデバッグを開始することができます。

例:

この例では、デバッグ・ビューを持つ 3 つのモジュールを作成して、デバッグ・セッションを開始する方

法を示します。

1. ルート・ソース・ビューを持つモジュール T1520IC1 を作成するために、次のように入力します。

   CRTMOD MODULE(MYLIB/T1520IC1) SRCFILE(QCPPPLE/QACSRC) DBGVIEW(*SOURCE)

   モジュール T1520IC1 をデバッグするために、ルート・ソース・ビューとデバッグ・データが作成され

   ます。

2. 3 つすべてのビューを持つモジュール T1520IC2 を作成するために、次のように入力します。

   CRTMOD MODULE(MYLIB/T1520IC2) SRCFILE(QCPPPLE/QACSRC) DBGVIEW(*ALL)

   OPTION(*SHOWINC)

   モジュール T1520IC2 をデバッグするために、すべてのビューとデバッグ・データが作成されます。

3. ルート・ソース・ビューとインクルード・ビューの両方を持つモジュール T1520IC3 を作成するため

   に、次のように入力します。

   CRTMOD MODULE(MYLIB/T1520IC3) SRCFILE(QCPPPLE/QACSRC) DBGVIEW(*SOURCE)

   OPTION(*SHOWINC)

   モジュール T1520IC3 をデバッグするために、ルート・ソース・ファイルとユーザーエ・インクルード・

   ファイルを含むインクルード・ビューと、デバッグ・データが作成されます。

4. プログラム T1520PG1 を作成するために、次のように入力します。

   CRTPGM PGM(MYLIB/T1520PG1) MODULE(MYLIB/T1520IC1 MYLIB/T1520IC2) ENTMOD(*ONLY)

   BDNIOR(MYLIB/T1520BD1) DETAIL(*FULL)

   注：このプログラムの作成には、モジュール、サービス・プログラム、およびバインド・ディレクトリ

   ーが必要です。4-7 ページの『2 ステップでのプログラムの作成』を参照してください。

5. プログラム T1520PG1 のデバッグ・セッションを開始するために、次のように入力します。

   STRDBG PGM(MYLIB/T1520PG1)

   「モジュール・ソースの表示」画面が次のように表示されます。
モジュール T1520IC1 が表示されます。これは、main() 関数を含むモジュールです。

次のように入力して、OPM プログラム、または ILE プログラムと OPM プログラムの組み合わせに対して、デバッグ・セッションを開始できます。

STRDBG PGM(MYLIB/T1520RP1 MYLIB/T1520CB1 MYLIB/T1520IC5) DSPMODSRC(*YES)

パラメーター DSPMODSRC(*YES) は、デバッグ開始時に ILE ソース・デバッグ・プログラム表示パネルを表示することを指定します。DSPMODSRC パラメーターは、デフォルトとして値 *PGMDEP を受け入れます。この値は、STRDBG PGM リストのプログラムが ILE プログラムである場合、ソース表示パネルが表示されることが示す。

**デバッグ・セッションでのプログラムの追加および除去**

*CHANGE の権限を持っているければ、プログラムおよびサービス・プログラムをデバッグ・セッションに追加したり、現行のデバッグ・セッションから除去したりすることができます。

**ILE プログラムについては**、DSPMODSRC コマンドの「モジュールの処理リスト」画面 (F14) のオプション 1 (プログラムの追加) を使用します。ILE プログラムまたはサービス・プログラムを除去するには、同じ画面のオプション 4 を使用します。ILE プログラムまたはサービス・プログラムを除去すると、そのプログラムのすべてのブレーキポイントは除去されます。デバッグ・セッションに一度入れることができるので ILE プログラムまたはサービス・プログラムの数を制限はありません。

**OPM プログラムの場合には**、OPMSRC に指定された値に応じて 2 つの選択があります。SET デバッグ・コマンドの STRDBG、あるいはデバッグ変更 (CHGDBG) オプションのいずれかを使用して OPMSRC(*YES) を指定している場合には、「モジュールの処理」画面を使用して OPM プログラムの追加もしくは除去を行います。(モジュール名は OPM プログラムリストされていないことに注意してください)。OPMSRC(*YES) が指定されているときにデバッグ・セッションに入れることができる OPM プログラムの数に制限はありません。

OPMSRC(*NO) を指定している場合には、プログラム追加 (ADDPGM) コマンドまたはプログラム除去 (RMVPGM) コマンドを使用する必要があります。OPMSRC(*NO) が指定されているときに、一度にデバッグ・セッションに入れることができる OPM プログラムは 20 個だけです。
注: デバッグ・データを含む OPM プログラムを、ILE デバッグ・セッションと OPM デバッグ・セッションの両方からデバッグすることはできません。OPM プログラムが既に OPM デバッグ・セッションに入っている場合には、まず最初にプログラムをこのセッションから除去してから、ILE デバッグ・セッションに追加したり、呼び出しステートメントからステップインしなければなりません。同様に、プログラムを OPM デバッグ・セッションからデバッグしたい場合には、まず最初にプログラムを ILE デバッグ・セッションから除去する必要があります。

例:

この例は、ILE C サービス・プログラムをデバッグ・セッションに追加する方法、および ILE C プログラムをデバッグ・セッションから除去する方法を示しています。

注: ILE C プログラム T1520ALP がこのデバッグ・セッションに含まれており、このプログラムのデバッグが完了したものとします。このプログラムは、このデバッグ・セッションから除去できます。

1. デバッグ・セッションへのプログラムの追加またはデバッグ・セッションからのプログラムの除去を行うには、次のように入力します。

DSPMODSRC

そして、実行キーを押します。「モジュール・ソースの表示」画面が表示されます。

2. F14 (モジュール・リストの処理) を押して、モジュール・リストの処理を表示します。

3. この画面では、リストの最初の行に 1 (プログラム追加) を入力して、プログラムおよびサービス・プログラムをデバッグ・セッションに追加します。サービス・プログラム T1520SPI を追加するには、「プログラム/モジュール」フィールドに T1520SPI と入力し、「ライブラリー」フィールドに MYLIB と入力して、デフォルト・プログラム・タイプを *PGM から *SRVPGM に変更し、Enter キーを押します。

4. この画面では、デバッグ・セッションから削除する各プログラムまたはサービス・プログラムの横にある行に 4 (プログラム除去) と入力します。

5. プログラム T1520ALP を除去するには、T1520ALP の横に 4 と入力し、Enter キーを押します。

6. F12 (取消し) を押して「モジュール・ソースの表示」画面に戻ります。

デバッグ・データを含む ILE C/C++ プログラムがデバッグ・セッションにある場合、main() 関数を使用したモジュールが表示されます (デバッグ・ビューがある場合)。そうでない場合には、デバッグ・データを持つ ILE C/C++ プログラムにバインドされた、最初のモジュールが表示されます。

セッション中のデバッグ・オプションの設定または変更

デバッグ・セッションを開始した後で、以下のデバッグ・オプションを設定あるいは変更することができます。

- プログラムのデバッグ中にデータベース・ファイルを更新できるかどうか (このオプションは、STRDBG コマンドの UPDPROD バラメーターと対応しています)。
- FIND を使ったテキスト検索は大文字小文字の区別をするかどうか。
- ILE ソース・デバッガを使用して OPM プログラムをデバッグするかどうか。 (これは OPMSRC バラメーターと対応しています)。

デバッグ・コマンド SET を使用してデバッグ・オプションを変更すると、対応するバラメーターが STRDBG コマンドに指定されていれば、このバラメーターの値に影響を与えます。また、デバッグの変更 (CHGDBG) コマンドを用いてデバッグ・オプションを設定することもできます。
例: ILE デバッグ・セッションへの OPM プログラムの追加
以下の例では、ILE ソース・デバッガーが ILE デバッグ・セッションに OPM プログラムを追加できるようにする方法を示しています。

ILE プログラムを処理するデバッグ・セッション中であり、しかも、デバッグ・データが利用できる OPM プログラムもデバッグする必要があるとします。その場合、ILE ソース・デバッガーが OPM プログラムを受け入れられるようにするには、次のようにします。
1. STRDBG の入力後に現行画面が「モジュール・ソースの表示」画面になっていない場合には、以下を入力します。
   
   DSPMODSRC

   «モジュール・ソースの表示」画面が表示されます。
2. SET
   と入力します。
3. 「デバッグ・オポションの設定 (Set Debug Options)」画面が表示させられ、「OPM ソース・デバッ グ・サポート (OPM source debug support)」フィールドに Y (Yes) を入力して、実行キーを押します。
   これで「モジュール・ソースの表示」画面に戻ります。

これで、モジュール処理画面を使用するか、あるいはそのプログラムへの呼び出しステートメソッドを処理することにより、OPM プログラムを追加できるようになりました。

例: デバッグ・セッション時のデバッグ・オポションの設定
以下の例では、デバッグ・セッション時に Update production files デバッガー・オプションを設定する方法を示しています。
1. デバッグ・セッションからデバッグ・オプションを設定するには、
   
   DSPMODSRC

   と入力します。
2. 「モジュール・ソースの表示」画面が表示されたら、
   
   SET

   と入力します。
3. 「デバッグ・オプションの設定 (Set Debug Options)」画面が表示させられ、「実動ファイルの更新」
   フィールドに Y (Yes) を入力して実行キーを押します。これで「モジュール・ソースの表示」画面に戻ります。ジョブがデバッグ・モードになっているときには、実動ライブラリー内のデータベース・ファイルが更新されます。

プログラム・ソースの表示
「モジュール・ソースの表示」画面には、一度に 1 モジュールずつ ILE プログラム・オブジェクトのソ ースが表示されます。 ILE モジュール・オブジェクトが次のデバッグ・ビュー・オプションのいずれかを 使用してコンパイルされている場合には、そのモジュール・オブジェクトのソースを表示することができま

・ DBGVIEW(*SOURCE)
・ DBGVIEW(*COPY) - ILE RPG のみ
・ DBGVIEW(*LIST)
・ DBGVIEW(*ALL)

IBM i: ILE C/C++ プログラマーの手引き
以下の条件が満たされる場合には、OPM プログラムのソースを表示することができます。
1. OPM プログラムが、OPTION(*LSTDBG) を指定してコンパイルされている。
2. ILE デバッグ環境が、OPM プログラムを受け入れるように設定されている。すなわち OPMSRC の値が *YES です (システムのデフォルト値は OPMSRC(*NO) です)。

モジュールのビューを表示した後、別のモジュールを表示したり、異なる複数のビューでモジュールを作成した場合に、同じモジュールの別のビューを表示したりしたい場合があります。ILE ソース・デバッガーやモジュールが表示された最後の位置を覚えており、モジュールを再度表示する時には同じ位置で表示します。ブレーキポイントが設定された行は強調表示されます。ブレーキポイント、ステップ、またはメッセージによりプログラムが停止し、画面が表示されるとブレーキポイントが起こった場所のステートメントが強調表示されます。

プログラムで他のモジュールを表示する

ご使用のプログラムの他のモジュールでデバッグ・オプションを設定することが必要な場合があります。これを行うには、「モジュール・ソースの表示」画面に表示されているモジュールを変更して、優先モジュールを指定してください。

「モジュール・ソースの表示」画面に表示されているモジュールを変更するには、以下のいずれかを使用します。
・ 「モジュール・リストの処理」画面
・ 「モジュールの表示」デバッグ・コマンド

ILE プログラム・オブジェクトで「モジュールの表示」デバッグ・コマンドを使うと、ルート・ソース、コピー、またはリスト・ビューの入力モジュールが (もしあれば) 表示されます。そうでない場合には、デバッグ・データを持つプログラム・オブジェクトにバインドされた、最初のモジュール・オブジェクトが表示されます。このオプションを OPM プログラム・オブジェクトに対して使用した場合、ソース・ビューまたはリスト・ビューが使用可能であれば、それが表示されます。

例: セッションで表示されるモジュールの変更: 以下の例では、「モジュール・ソースの表示」画面に表示されているモジュールを、「モジュールの表示」デバッグ・コマンドを使用して同じプログラム内の別のモジュールに変更する方法を示しています。
1. デバッグ・セッション中に、DSPMODSRC と入力します。「モジュール・ソースの表示」画面が表示されます。
2. デバッグ・コマンド行で display module T1520IC2 と入力します。

モジュール T1520IC2 が表示されます。

モジュールの別のビューの表示

モジュールの作成時に指定した値に応じて、モジュールのいくつかの異なるビューを使用することができ、それらは次のとおりです。
・ ルート・ソース・ビュー
・ インクルード・ソース・ビュー
・ リスト・ビュー

例:

この例は、「モジュール・ソースの表示」画面に表示されるモジュールのビューを変更する方法を示しています。
1. 「モジュール・ソースの表示」画面上のモジュールのビューを変更するには、DSPMODSRC と入力して Enter キーを押してください。「モジュール・ソースの表示」画面が表示されます。

2. F15（ビューの選択）キーを押してください。「ビューの選択」ウィンドウは次のようになります。

<table>
<thead>
<tr>
<th>ビューの選択</th>
</tr>
</thead>
<tbody>
<tr>
<td>現行ビュー . . .</td>
</tr>
<tr>
<td>オプションを入力して、実行キーを押してください。</td>
</tr>
<tr>
<td>OPT ビュー</td>
</tr>
<tr>
<td>ILE C ルート・ソース・ビュー</td>
</tr>
<tr>
<td>ILE C 組み込みビュー</td>
</tr>
<tr>
<td>F12=取消し</td>
</tr>
<tr>
<td>オプションを入力して、実行キーを押してください。</td>
</tr>
</tbody>
</table>

現在のビューがウィンドウの最上部にリストされ、使用可能な他のビューはその下に表示されます。作成に使用したデバッグ・オプションによって、プログラム中の各モジュールがビューの異なるセットをもつことがあります。

3. ILE C Include view の横に 1 と入力します。「モジュール・ソースの表示」画面が表示され、インクルード・ソース・ビューでモジュールが表示されます。インクルード・ビューのソースは、現行ビューのステートメント位置に相当するステートメント位置に表示されます。

### 停止点を使用したデバッグの支援

プログラムの流れは停止点によって制御することができます。

停止点でプログラムが停止されると、「モジュール・ソースの表示」画面が表示されます。この画面を使用して、変数を評価し、追加停止点を設定し、ソース・デバッガー・コマンドを実行します。条件が発生した行にソースが位置付けられ、該当するモジュールが表示されます。最後にソースを表示していたときにカーソルが画面のテキスト域にあった場合、カーソルは、停止点が発生した行に置かれます。そうでない場合には、デバッガー・コマンド行に位置付けられます。

停止点を設定した後にモジュールのビューを変更する場合には、停止点の行番号はソース・デバッガーによって新しいビューにマップされます。

このトピックでは以下について説明します。

- 停止点のタイプ
- 非スレッド・プログラムで条件付きおよび無条件停止点を設定する方法
- 条件付きスレッド停止点を設定する方法
- 停止点をテストする方法
- すべての停止点を除去する方法
ブレークポイントのタイプ
ブレークポイントのタイプによって、制御するフローの有効範囲が決まります。

ジョブおよびスレッドのブレークポイント
ブレークポイントにはジョブとスレッドの2つのタイプがあります。
・一般的に、ブレークポイントは、プログラムまたはジョブの処理を停止するために使用します。
・スレッド化されたアプリケーションの各スレッドは、自身のスレッド・ブレークポイントを持つ場合があります。

ジョブおよびスレッドのブレークポイントは両者とも、無条件または条件付きのいずれかにすることできます。

一般に、デバッガ・コマンドとファンクション・キーのセットが、ジョブ・ブレークポイントとスレッド・ブレークポイントのそれぞれに1つずつあります。

ブレークポイントに関するこのセクションの残りの部分では、用語「ブレークポイント」は、特に断りがない限り、ジョブとスレッドの両方に対して使用します。

条件付きブレークポイントと無条件ブレークポイント
無条件ブレークポイントと条件付きブレークポイントを設定できます。無条件ブレークポイントは、特定のステートメントでプログラムを停止します。条件付きブレークポイントは、特定のステートメントの特定の条件が満たされるとプログラムを停止します。

ブレークポイントの設定
モジュールを処理するには、以下のいずれかを使用します。
・F13 (モジュール・ブレークポイントの処理)
・F6 (ブレークポイントの追加/消去)

BREAK デバッガ・コマンドを使用することで、条件付きまたは無条件のブレークポイントを設定することができます。

注: また、BREAK または TBREAK デバッガ・コマンドでブレークポイントを追加することも可能です。
BREAK コマンドの使用法については、5-22ページの『コマンド行から無条件ブレークポイントを設定する』を参照してください。TBREAK コマンドの使用法については、5-24ページの『条件付きスレッド・ブレークポイントの設定』を参照してください。

CLEAR デバッガ・コマンドを使用することで、条件付きまたは無条件のブレークポイントを除去することもできます。

「モジュール・ソースの表示」画面での無条件ブレークポイントの設定
例:
以下の例では、F6 (ブレークポイントの追加/消去) を使って無条件ブレークポイントを設定する方法を示しています。
1. DSPMODSRC と入力します。「モジュール・ソースの表示」画面が表示されます。
2. 変更したいモジュール・ソースを表示するには、display module name と入力して、実行キーを押します。name は、変更したいモジュールのファイル名です。
3. 設定したい無条件ブレークポイントごとに、以下のようにします。
a. カーソルを新規ブレークポイントに続く行に置きます。
b. F6 (ブレークポイントの追加/消去) キーを押してください。

注: 指定した行にブレークポイントがない場合は、その行に無条件ブレークポイントが設定されています。指定した行にブレークポイントがある場合は、たとえ条件付きブレークポイントであっても除
去されます。
以下の例は、モジュール T1520PG1 の行 50 に設定されている無条件ブレークポイントを示しています。

<table>
<thead>
<tr>
<th>プログラム</th>
<th>T1520PG1</th>
<th>タイプ</th>
<th>MLIB</th>
<th>モジュール</th>
<th>T1520C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>if (j&lt;0) return(0);</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>if (hold_formatted_cost[i] == '$')</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>formatted_cost[j] = hold_formatted_cost[i];</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>break;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>if (i&lt;16 &amp; &amp; (i-2)%3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>formatted_cost[j] = ',';</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>--j;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>formatted_cost[j] = hold_formatted_cost[i];</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>--j;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

注: ブレークポイントを除去するには、CLEAR コマンドを使用します。例えば、clear 50 では行
50 のブレークポイントが除去されます。

4. すべてのブレークポイントが設定された後では、以下のようにします。
  a. 「モジュール・ブレークポイントの処理」画面を終了するには、F12 (取消し) を押します。
  b. ILE ソース・デバッガーを終了するには、F3 (プログラム終了) を押します。ブレークポイントは除
去されません。

コマンド行から無条件ブレークポイントを設定する

BREAK デバッグ・コマンドを使用して無条件ブレークポイントを設定するには、デバッグ・コマンド行に
BREAK line-number と入力してください。変数の line-number は、ブレークポイントを設定したいモジュールの現在表示されているビューの行番号です。

注: CLEAR デバッグ・コマンドを使用して無条件ブレークポイントを除去するには、デバッグ・コマンド行に CLEAR line-number と入力してください。

マクロの条件付きブレークポイントの設定

マクロ・ステートメント・マクロの最初のステートメントでブレークポイントを設定するには、マクロ展開
ではなくマクロ呼び出しを含む行にカーソルを置きます。

例:

以下の例では、F13 (モジュール・ブレークポイントの処理) を使って条件付きブレークポイントを設定す
る方法を示しています。
1. モジュールを処理するには、DSPMODSRC と入力します。「モジュール・ソースの表示」画面が表示されます。
2. 変更したいモジュール・ソースを表示するには、display module name と入力して、実行キーを押します。name は、変更したいモジュールのファイル名です。
3. 設定したい条件付きブレーキポイントごとに、以下のようにします。
   a. カーソルを新規ブレーキポイントに続く行に置きます。
   b. F13 (モジュール・ブレーキポイントの処理) を押します。「モジュール・ブレーキポイントの処理」画面が表示されます。
   c. リストの最初の行にカーソルを置き、1 (追加) を入力してから、実行キーを押します。例えば、i が21である場合に、行35で条件付きブレーキポイントを設定するとします。
      1) その行で35と入力します。
      2) 「条件」フィールドにi==21と入力します。
      3) 実行キーを押します。

注：パネルを切り替えたくない場合、「モジュール・ソースの表示」画面のコマンド行で以下を入力して同じブレーキポイントを設定できます。
break 35 when i==21

条件付きブレーキポイントが行35に設定されます。

<table>
<thead>
<tr>
<th>モジュール停止点の処理</th>
<th>システム：TORASD80</th>
</tr>
</thead>
<tbody>
<tr>
<td>プログラム .: T1520PG1</td>
<td>ライブラリー .: MYLIB</td>
</tr>
<tr>
<td>モジュール .: T1520IC2</td>
<td>タイプ .:.: *PGM</td>
</tr>
<tr>
<td>オプションを入力して、実行キーを押してください。</td>
<td></td>
</tr>
<tr>
<td>1=追加 4=消去</td>
<td></td>
</tr>
<tr>
<td>OPT 行 条件</td>
<td></td>
</tr>
<tr>
<td>1 35 i==21</td>
<td></td>
</tr>
<tr>
<td>_ 50</td>
<td></td>
</tr>
</tbody>
</table>

注：既存のブレーキポイントは常に、同じ位置に入力された新しいブレーキポイントによって置き換えられます。

4. すべてのブレーキポイントが設定された後では、以下のようにします。
   a. 「モジュール・ブレーキポイントの処理」画面を終了するには、F12 (取消し) を押します。
   b. ILE ソース・デバッガーを終了するには、F3 (プログラム終了) を押します。ブレーキポイントは除去されません。

ステートメントの条件付きブレーキポイントの設定
条件付きブレーキポイントをステートメントに設定できます。例えば、行番号およびステートメント番号を含むコンパイラ・リストがある場合、1つの行に複数のステートメントがあるときに、ステートメント構文を使用して特定のステートメントにブレーキポイントを設定できます。

<table>
<thead>
<tr>
<th>Line</th>
<th>Stmt</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>24</td>
<td>i=j; j=0;</td>
</tr>
<tr>
<td>34</td>
<td>26</td>
<td>array[i] = cost;</td>
</tr>
</tbody>
</table>

Break myfunction/25は、ステートメントi=0がmyfunctionにあるものとみなしてこのステートメントにブレーキポイントを設定します。ここでBreak33と入力すると、ステートメント24(i=j)でブレーキポイントが設定されます。
マルチ・ステートメント・マクロの最初のステートメントでブレークポイントを設定するには、マクロ展開ではなくマクロ呼び出しを含む行にカーソルを置きます。

### 条件付きスレッド・ブレークポイントの設定
条件付きスレッド停止点の設定または除去を行うには、次のものを使用します。
- 「モジュール・ブレークポイントの処理」画面
- 現行スレッドの条件付きスレッド・ブレークポイントを設定する TBREAK デバッグ・コマンド
- 条件付きスレッド・ブレークポイントを除去する CLEAR デバッグ・コマンド

#### 「モジュール・ブレークポイントの処理」画面での条件付きスレッド・ブレークポイントの設定
「モジュール・ブレークポイントの処理」画面を使用して条件付きスレッド・ブレークポイントを設定するには、以下のように行います。
1. F13 を押して「モジュール・ブレークポイントの処理」を表示し、実行キーを押します。
2. Opt フィールドに 1 (追加) と入力して、実行キーを押します。
3. 残りのフィールドを、条件付きジョブ・ブレークポイントと同様に埋めます。

「モジュール・ブレークポイントの処理」画面を使用して条件付きスレッド・ブレークポイントを除去するには、除去したいブレークポイントの横の OPT に 4 (消去) と入力して、実行キーを押します。

### コマンド行から条件付きスレッド・ブレークポイントを設定する
TBREAK デバッグ・コマンドの構文は BREAK デバッグ・コマンドの場合と同じ構文を使います。この 2 つのコマンドの相違点は、BREAK デバッグ・コマンドが条件付きジョブ停止点をすべてのスレッドの同じ位置に設定するのに対して、TBREAK デバッグ・コマンドは現行スレッドに条件付きスレッド停止点を設定するところです。

注: 条件付きスレッド・ブレークポイントを除去するには、CLEAR デバッグ・コマンドを使います。条件付きスレッド・ブレークポイントが消去される時は、現行スレッドのみが対象です。

### 停止点のテスト
1. プログラムを呼び出してください。
2. 無条件停止点に達すると、プログラムが停止して「モジュール・ソースの表示」画面が再表示されます。
3. 条件付き停止点に達すると、ステートメントが実行される前に式が評価されます。
   - 結果が真の場合 (例えば、i が 21 に等しい場合)、プログラムは停止し、「モジュール・ソースの表示」画面が表示されます。
   - 結果が偽の場合には、プログラムは実行し続けます。

### すべての停止点の除去
CLEAR PGM デバッグ・コマンドを使用すると、「モジュール・ソースの表示」画面に表示されているモジュールを含むプログラムから、すべてのブレークポイント (条件付きおよび無条件) を除去できます。デバッギング・コマンドを使用するためには、デバッギング・コマンド行に CLEAR PGM と入力して実行キーを押してください。プログラムにバインドされたすべてのモジュールからブレークポイントが除去されます。

5-24  IBM i: ILE C/C++ プログラマーの手引き
ウォッチを使用したデバッグの支援
ウォッチ条件を使用して、記憶域の位置のアドレスを決定する変数または式の現行値の変更をモニターします。ウォッチ条件の設定は、条件付きブレークポイントの設定とほぼ同じですが、次のような重要な相違点が1つあります。

- デバッグ状態では、変数の値が現行値から変更されると、ただちにプログラムが停止します。
- 条件付きブレークポイントでは、ステートメントの実行時に、関連付けられた式に記述された条件が満たされた場合にのみ、プログラムが停止します。

デバッガーは、ウォッチ条件の設定時に計算される記憶域アドレスの内容によって、変数を監視します。記憶域アドレスの内容が、ウォッチ条件の設定時の値、または最終のウォッチ条件発生時の値から変更されると、ブレークポイントが設定され、プログラムは停止します。

注: デバッグ状態で変数が変更された後、監視されている記憶域の位置にある新しい内容が、対応する変数の新しい値として保存されます。監視されている変数の最新の値がそれ以降に変更されると、次のウォッチ条件が登録されます。

このトピックでは以下について説明します。
- デバッグの特性
- デバッグ条件の設定および除去方法
- 变数へのウォッチ条件の設定例
- 活動状態のウォッチの表示方法

ウォッチの特性と制約
ウォッチを使用する場合、以下のウォッチの特性に留意してください。

- サーバーはシステム全体でモニターされ、同時に活動状態にできるウォッチの最大数は256です。
- サーバーはシステムが設定したウォッチの数が含まれています。

システム全体の使用状況に応じて、所定の時間に設定できるウォッチ条件の数が制限されることがあります。システム内で活動状態のウォッチが最大数を超えている場合にウォッチ条件を設定しようとするとき、エラー・メッセージが表示されます。

注: 1つの変数がページ境界にまたがる場合は、ストレージ・ロケーションのモニターのために2つのウォッチが同時に使用されます。この場合、システム全体を通じて同時にウォッチ可能な変数の最大数は、128から256の範囲になります。

- ウォッチ条件を設定できるのは、デバッグによりプログラムが停止されている場合に、ウォッチされる変数が有効範囲内にある場合です。これにあてはまらない場合にウォッチが要求されると、対応する呼び出しスタック項目が存在しないことを示すエラー・メッセージが表示されます。
- いったんウォッチ条件が設定されると、ウォッチされるストレージ・ロケーションのアドレスは変わりません。したがって、一時ロケーションにウォッチを設定すると、実体の値を更新するウォッチ条件通知が生じる可能性があります。

この例はILE C/Cブリッジャーの自動ストレージ（ブリッジャーが終了した場合には、再利用することができる）です。

ウォッチされる変数が有効範囲内にある場合でも、ウォッチ条件がトリガーされることがあります。そのため、ウォッチ条件が報告されたら、ウォッチの変数が有効範囲内にあると見なすことはできません。
ウォッチ条件の設定および除去
ウォッチ条件の設定は、プログラムをデバッグの制御下で停止する必要があり、またウォッチしたい変数が有効範囲内になければなりません。

- グローバル変数をウォッチするには、ウォッチ条件の設定前に、変数が定義されているプログラムが活動状態であることを確認する必要があります。
- ローカル変数をウォッチするには、ウォッチ条件を設定する前に、変数が定義されている関数にステップインツグする必要があります。

ウォッチ条件の設定
以下を使用して、ウォッチ条件を設定することができます。

- カーソルの下の変数にウォッチ条件を設定する F17 (ウォッチ変数)。
- パラメーターの有無に関係なく、WATCH デバッグ・コマンド。

WATCH デバッグ・コマンドの使用
WATCH コマンドを使用する場合には、単一のコマンドとして入力しなければなりません。同一のコマンド入力行で他のデバッグ・コマンドを使用することはできません。
- 以下に示した「ウォッチの処理」画面にアクセスするには、パラメーターなしで WATCH と入力します。
監視の処理

オプションを入力して、実行キーを押してください。

4=消去  5=表示

<table>
<thead>
<tr>
<th>OPT</th>
<th>NUM</th>
<th>変数</th>
<th>アドレス</th>
<th>長さ</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>1</td>
<td>salary</td>
<td>080090506F027004</td>
<td>4</td>
</tr>
</tbody>
</table>


コマンド

---

F3=終了  F4=プロンプト  F5=最新表示  F9= コマンドの複写  F12= 取り消し

「監視の処理」画面には、デバッグ・セッションで現在活動中の全監視が表示されます。この画面からウォッチを消去し表示することができます。5=表示を選択すると、以下に示した「ウォッチ表示」ウィンドウに現在アクティブのウォッチに関する情報が表示されます。

<table>
<thead>
<tr>
<th>監視の処理</th>
</tr>
</thead>
<tbody>
<tr>
<td>監視の表示</td>
</tr>
<tr>
<td>監視番号 . . . . : 1</td>
</tr>
<tr>
<td>アドレス . . . . : 080090506F027004</td>
</tr>
<tr>
<td>長さ . . . . . . : 4</td>
</tr>
<tr>
<td>ヒットの数 . . . : 0</td>
</tr>
<tr>
<td>監視設定時の有効範囲 :</td>
</tr>
<tr>
<td>アドレス/タブ/タイプ : PAYROLL  ABC  +PGM</td>
</tr>
<tr>
<td>モジュール : PAYROLL</td>
</tr>
<tr>
<td>プロシージャー : main</td>
</tr>
<tr>
<td>変数 . . . . : salary</td>
</tr>
<tr>
<td>F12= 取り消し</td>
</tr>
</tbody>
</table>

コマンド

---

F3=終了  F4=プロンプト  F5=最新表示  F9= コマンドの複写  F12= 取り消し

- ウォッチされる変数を指定するには、以下のデバッグ・コマンドを入力します。
  
  **WATCH variable**
  
  このコマンドは、variable の値が現行値から変更される場合に、ブレークポイントを設定するよう要求します。例えば、watch V。V は変数です。

- ウォッチされる式を指定するには、以下のデバッグ・コマンドを入力します。
  
  **WATCH expression**
  
  このコマンドは、expression の値が現行値から変更される場合に、ブレークポイントを設定するよう要求します。

注: expression は、ウォッチされるストレージ・ロケーションのアドレスを決定するために使用されるもので、割り当て可能なロケーションに解決される必要があります。例えば、watch (p+2)。p はポインターです。

ウォッチにおける式の変数の有効範囲は、直前に出した QUAL コマンドによって定義されます。
ウォッチ条件を設定し、併せてウォッチの長さを指定するには、WATCH expression : watch-length と入力します。
ウォッチごとに、最大 128 バイトの連続記憶域をモニタし、比較することができます。128 バイトの最大長を超えると、監視条件は設定されず、デバッガーがエラー・メッセージを出します。
デフォルトでは、式タイプの長さは、ウォッチ比較命令の長さでもあります。watch-length パラメーターはこのデフォルトをオーバーライドします。このパラメーターは式のバイト数を判別するので、このバイト数を比較して、値が変更されたかどうかを判別します。
たとえば、4 バイトの 2 進数が変数として指定されていて、監視長パラメーターが指定されていないければ、比較長は 4 バイトということになります。ただし、ウォッチの長さパラメーターを指定すると、ウォッチの長さを判別する際の式の長さを一時変更します。
ウォッチ条件の除去
以下的方法でウォッチを除去できます。
• WATCH キーワードを指定した CLEAR コマンドは、1 つのウォッチを選択して終了したり、すべてのウォッチを終了します。例えば、watch-number によって識別されるウォッチを消去するには、以下を入力します。
  CLEAR WATCH watch-number
ウォッチ番号は、「ウォッチの処理」画面から取得できます。
セッションのすべてのウォッチを消去するには、以下を入力します。
  CLEAR WATCH ALL
注: CLEAR PGM コマンドは、表示中のモジュールが含まれるプログラム内のすべてのブレークポイントを消去しますが、ウォッチには何の影響もありません。ウォッチ条件を除去するには、CLEAR コマンドで WATCH キーワードを明示的に使用する必要があります。
ウォッチ条件の自動除去
ウォッチは、以下的方法でも除去されます。
• CL デバッグ終了 (ENDDBG) コマンドは、ローカル・ジョブまたはサービス・ジョブ内に設定されたウォッチを除去します。
注: 異常時には ENDDBG が自動的に呼び出され、影響を受けたウォッチがすべて除去されるようにします。
• IBM i プラットフォームの初期プログラム・ロード (IPL) では、システム全体ですべてのウォッチ条件が除去されます。
ウォッチ条件の設定例
この例では、プログラム MYLIB/PAYROLL の変数 salary をウォッチします。ウォッチ条件を設定するには、デバッグ行に WATCH salary と入力し、ウォッチの長さとしてデフォルト値を受け入れます。
後で変数 salary の値が変わると、アプリケーションが停止し、「モジュール・ソースの表示」画面は以下のようになります。
プログラム：PAYROL
イブレ：MYLIB
モジュール：PAYROLL

52 for (cnt=0;
53 cnt<EMP.MAX &&
54 scanf("%s%shd%d", payptr->first, payptr->last,
55 &(payptr->wage), &eflag, &(payptr->hrs))!=EOF;
56 cnt++, payptr++)
57 {
58 payptr->exempt=eflag;
59 }
60 empsort(payfile, cnt);
61 for (index=1, payptr=payfile; index<=cnt; index++,payptr++)
62 if (payptr->exempt==1) {
63 salary = 40*(payptr->wage);
64 numexempt++; 
65 }
66 else
67 salary = (payptr->hours)*(payptr->wage);

モジュール・ソースの表示
(ソースを使用することができない。)

次のプログラムは、ILE デバッグ環境に追加することはできません。
- デバッグ・データのない ILE プログラム
- 非ソース・デバッグ・データだけがある OPM プログラム
- デバッグ・データのない OPM プログラム

最初の 2 つの場合、停止したステートメント番号が渡されます。3 番目の場合、停止した MI 命令が渡されます。上に示すブランクの「モジュール・ソースの表示」画面の下に情報が表示されます。行番号ではなく、ステートメントまたは命令番号が表示されます。

活動状態のウォッチの表示
システム全体における活動状態のウォッチのリストを表示し、どのジョブがそれらを設定したかを表示するには、コマンド行に DSPDBGWCH と入力します。このコマンドは、以下に示した「デバッグ・ウォッチの表示」画面を表示します。
注: この画面には、システムによって設定されたウォッチ条件は表示されません。

プログラムのステップスルー

ILE ソース・デバッガーの STEP コマンドを使用して、プログラムの指定した数のステートメントを実行し、実行する次のステートメントの位置で「モジュール・ソースの表示」画面に戻ることができます。カーソルは、最後にソースが表示されたときに画面のテキスト域にあった場合は、このステートメントに位置付けられます。そうでない場合には、デバッガー・コマンド行に位置付けられます。プログラムは、停止したステートメントで開始します。ブレークポイントを設定すると、そのステートメントが実行される前にプログラムは停止します。実行するステートメントのデフォルト数は 1 です。

このトピックでは、以下を行うための方法について説明します。
- プログラムのステップオーバー
- プログラムへのステップイン
- プロシージャーのステップオーバー
- プロシージャーへのステップイン

プログラムのステップオーバー

プログラムをステップオーバーするには、以下を使用できます。
- 「モジュール・ソースの表示」画面の F10 (ステップ)
- STEP OVER デバッガー・コマンド

F10 でのプログラムのステップオーバー

「モジュール・ソースの表示」画面で F10 (ステップ) を使用すると、デバッガー・セッションの呼び出されたプログラムをステップオーバーすることができます。次に実行されるステートメントが別のプログラムへの CALL ステートメントである場合、F10 (ステップ) を押すと、呼び出し側プログラムがもう一度停止される前に、呼び出されたプログラムが完了まで実行されます。

STEP OVER デバッガー・コマンドの使用

ステップオーバー・デバッガー・コマンドを使用すると、デバッガー・セッションの呼び出されたプログラムをステップオーバーすることができます。ステップオーバー・デバッガー・コマンドを使用するには、STEP number-of-statements OVER と入力します。変数 number-of-statements は、プログラムがもう一度停止する前に次のステップで実行したいプログラムのステートメントの数です。

この変数が除外された場合、デフォルトは 1 です。実行されるステートメントの 1 つに別のプログラムの呼び出しが含まれている場合、ILE ソース・デバッガーは呼び出されたプログラムをステップオーバーします。
プログラムへのステップイン

プログラムにステップインするには、以下を使用します。
• 「モジュール・ソースの表示」画面の F22 (ステップイン)
• STEP INTO デバッグ・コマンド

F22 を使用したプログラムへのステップイン

「モジュール・ソースの表示」画面の F22 (ステップイン) を使用して、デバッグ・セッションで呼び出し先プログラムにステップインします。次に実行するステートメントが他のプログラムへの呼び出し (CALL) ステートメントである場合に F22 を押すと、呼び出し先プログラムの最初の実行可能ステートメントが実行されます。その後、呼び出し先プログラムが「モジュール・ソースの表示」画面に表示されます。

注: 呼び出し先プログラムは、「モジュール・ソースの表示」画面に表示されるために、デバッグ・データが関連付けられていなければならない。

STEP INTO デバッグ・コマンドの使用

STEP INTO デバッグ・コマンドを使用して、デバッグ・セッションで呼び出し先プログラムにステップインします。 STEP INTO デバッグ・コマンドを使用するには、次のように入力します。

STEP number-of-statements INTO

変数 number-of-statements は、プログラムがもう一度停止する前に次のステップで実行したいプログラムのステートメントの数です。この変数を省略した場合、デフォルトの値は 1 です。

呼び出されたプログラムのステップイントゥ

実行するステートメントの 1 つに別のプログラムの呼び出し (CALL) ステートメントが含まれる場合、ソース・デバッガーは、呼び出されたプログラムにステップイントゥします。呼び出されたプログラム中の各ステートメントがステップとして数えられます。呼び出されたプログラムがストップになると、その呼び出されたプログラムが「モジュール・ソースの表示」画面に表示されます。例えば、STEP 5 INTO と入力すると、プログラムの次の 5 つのステートメントが実行されます。3 番目のステートメントが別のプログラムの呼び出し (CALL) ステートメントである場合、呼び出し側プログラムの 2 つのステートメントが実行され、呼び出されたプログラムの最初の 2 つのステートメントが実行されます。

注: このステップはステートメントとしてカウントされます。

STEP INTO コマンドは呼び出し (CALL) コマンドも処理します。呼び出しの後で、これを利用してプログラムをステップスルーすることができます。ソース・デバッガーを開始した後で、初期「モジュール・ソースの表示」画面で STEP 1 INTO と入力して、実行キーを押します。これにより、ステップ・カウントは 1 に設定されます。

F22 を使用したプログラムへのステップイントゥの例

F22 (ステップイントゥ) を使用して、プログラム DEBUGEX からプログラム CPGM にステップイントゥします。
1. 「モジュール・ソースの表示」画面が DEBUGEX のソースを表示しているものとします。
2. プログラム CPGM の関数 CalcTax() の呼び出し前にある最後の実行可能ステートメントである行 92 に無条件停止点を設定するには、Break 92 と入力し、Enter を押します。
3. F3 (プログラム終了) を押して、「モジュール・ソースの表示」画面を終了します。
4. プログラムを呼び出してください。プログラムは図5-9に示すように、停止点92で停止します。

5. F22 (ステップイン)を押してください。プログラムの1つのステートメントが実行されてから、CPGM
の「モジュール・ソースの表示」画面が表示されます。

プログラムは、CPGMの最初の実行可能なステートメント（行13）で停止します。

注：F22を使用する時にはステップスルーするステートメントの数を指定することはできません。F22
キーを押すと、1つのステップが実行されます。
デバッグ・データが利用できない場合には、ソースが利用不能であることを知らせるメッセージを持つ、ブランドの「モジュール・ソースの表示」画面が表示されます。

**OPM プログラムへのステップイン**

他の関数への呼び出しが発生したとき、OPM プログラムに利用可能なデバッグ・データがあり、デバッグ・セッションがデバッグのために OPM プログラムを受け入れる場合には、OPM プログラムにステップインすることができます。

ILE ソース・デバッガーが OPM プログラムを受け入れるように設定されていない場合、あるいはデバッグ・データが利用できない場合には、ソースが利用不能であることを知らせるメッセージが示された、ブランドの「モジュール・ソースの表示」画面が表示されます。OPM プログラムが OPTION(*LSTDBG) を使用してコンパイルされた場合、このプログラムにはデバッグ・データがあります。

デフォルトのステップ・モードはステップオーバーです。

**プロシージャーのステップオーバー**

STEP デバッグ・コマンドに over を指定すると、プロシージャーまたは関数の呼び出しが 1 つのステートメントとしてカウントされます。これは、STEP のデフォルト・モードです。プログラムの 4 つのステートメントをステップスルーしたときに、その 1 つが 16 個のステートメントを含むプロシージャーの呼び出しであった場合、20 個のステートメントを実行することになります。ステップオーバー機能を開始するには、以下を使用できます。

・STEP OVER デバッグ・コマンド
・F10 (ステップ)

例:

この例では、F10 (ステップ) を使用して、プログラムで一度に 1 つのステートメントをステップオーバーする方法を示します。

1. モジュールを処理するためには、DSPMODSRC を入力して、実行キーを押してください。「モジュール・ソースの表示」画面が表示されます。
2. display module T1520IC2 と入力します。
3. 行 50 に無条件ブレーキポイントを設定するために、デバッグ・コマンド行に Break 50 と入力します。
4. 行 35 に条件付きブレーキポイントを設定するために、デバッグ・コマンド行に Break 35 when i==21 と入力します。
5. F12 (再開) を押して、「モジュール・ソースの表示」画面を終了します。
6. プログラムを呼び出してください。プログラムは、i が 21 である場合にブレーキポイント 35 で停止するか、あるいは行 50 で停止するか、どちらか先に発生したときに停止します。
7. ステートメントをステップオーバーするために、F10 (ステップ) を押します。プログラムの 1 つのステートメントが実行されると、「モジュール・ソースの表示」画面が表示されます。ステートメントが関数呼び出しである場合、その関数は完了まで実行されます。ただし、呼び出し関数にブレーキポイントが設定されている場合は、ブレーキポイントがヒットします。この時点では関数の中に位置し、次のステップで関数内の次のステートメントに進みます。

注: F10 を使用する時にはステップスルーするステートメントの数を指定することはできません。F10 を押すと、1 つのステップが実行されます。
モジュール・ソースの表示
プログラム：T1520PG1  仮名：MYLIB  エージェント：T1520IC2
47     if (j<0) return(0);
48     if (hold_formatted_cost[i] == '$')
        {
30     formatted_cost[j] = hold_formatted_cost[i];
31     break;
32     }
43     if (i<16 && (!(i-2)%3))
58     formatted_cost[j] = ',';
60     --j;
61     formatted_cost[j] = hold_formatted_cost[i];
62     --j;
63     }
64
F3=プログラム終了  F6=停止点の追加 / 消去  F10=ステップ
F11=変数の表示  F12=再開  F17=変数監視  F24=キーの続き
行 50 が停止点。

8. 5 個のステートメントをステップオーバーするには、デバッグ・コマンド行に step 5 over と入力します。プログラムの次の 5 つのステートメントが実行され、「モジュール・ソースの表示」画面が表示されます。

3 番目のステートメントが関数の呼び出しである場合には、最初の 2 つのステートメントが実行され、関数が呼び出され、残りの 2 つのステートメントが実行されます。

9. 11 個のステートメントをステップオーバーするには、デバッグ・コマンド行に step 11 over と入力します。プログラムの次の 11 個のステートメントが実行されます。「モジュール・ソースの表示」画面が表示されます。

プロシージャへのステップイン
自動のステップ実行機能があります。この機能では、サービス・プログラムが自動的にデバッグに入ります。これにより、デバッグ中の他のプログラムからステップインしたサービス・プログラムが、以下である場合に行われます。
- デバッグ・データを持つ
- デバッグ中ではない
- プロシージャを含む

ユーザーに対してこのサービス・プログラムがデバッグのために追加され、DSPMODSRC パネルにサービス・プログラムのプロシージャが表示されます。この時点から、ユーザーがデバッグのために追加したプログラム内のモジュールと同じように、「モジュールの処理」画面を使用してサービス・プログラム内のモジュールにアクセスできるようになります。

STEP デバッグ・コマンドで INTO を指定すると、呼び出されたプロシージャーまたは関数の各ステートメントが、1 個のステートメントとしてカウントされます。関数へのステップインを開始するには、以下を使用できます。
- STEP INTO デバッグ・コマンド
- F22 (ステップイン)

例：
この例では、F22（ステップイン）を使用して、1つ目のブローシャーにステップインする方法を示します。
1. DSPMODSRCと入力します。「モジュール・ソースの表示」画面が表示されます。
2. 行50に無条件ブレーカポイントを設定するために、デバッガ・コマンド行にBreak50と入力します。
3. 行35に条件付きブレーカポイントを設定するために、デバッガ・コマンド行にBreak35wheni==21と入力します。
4. F12（再開）を押して、「モジュール・ソースの表示」画面を終了します。
5. プログラムを呼び出してください。プログラムは、iが21である場合にブレーカポイント35で停止するか、あるいは行50で停止するか、どちらか先に発生したときに停止します。

モジュール・ソースの表示
プログラム：T1520PG1 インクルード：MYLIB エージェント：T1520IC2
47 if (j<0) return(0);
48 if (hold_formatted_cost[i] == $')
49 {
50 formatted_cost[j] = hold_formatted_cost[i];
51 break;
52 }
53 if ((i<16 && !(i-2)%3))
54 {
55 formatted_cost[j] = ,;  //ヌル値を変更
56 --j;
57 }
58 formatted_cost[j] = hold_formatted_cost[i];
59 --j;
60 }

デバッグ...

6. F22（ステップイン）を押してください。プログラムの1つのステートメントが実行されると、「モジュール・ソースの表示」画面が表示されます。ステートメントがプロシージャーまたは関数呼び出しの場合、プログラムはそのプロシージャーまたは関数の最初のステートメントで停止します。

注：F22を使用する時にはステップスルーするステートメントの数を指定することはできません。F22キーを押すと、1つのステップが実行されます。

7. 5つのステートメントにステップインするには、デバッガ・コマンド行にstep5int入力します。

プログラムの次の5つのステートメントが実行され、「モジュール・ソースの表示」画面が表示されます。3番目のステートメントが関数の呼び出しである場合には、呼び出し側プロシージャーの最初の2つのステートメントが実行され、関数の最初の2つのステートメントが実行されます。

注：このステップはステートメントとしてカウントされます。

8. 11個のステートメントにステップインするには、デバッガ・コマンド行にstep11int入力します。プログラムの次の11個のステートメントが実行されます。「モジュール・ソースの表示」画面が表示されます。
変数のデバッガ

スカラー変数、式、構造体、配列、または errno の値を表示し、EVAL デバッガ・コマンドを使用してスカラー変数または errno の値を変更できます。

このトピックでは、以下を行うための方法について説明します。

- デバッガ・セッション時にスカラー変数、式、構造体、配列、または errno の値を表示する
- EVAL デバッガ・コマンドを使用して、変数の値を変更する
- デバッガ・セッション時にスカラー変数または errno の値を変更する
- デバッガ・セッション時に変数、式、またはコマンドを指定して短縮名を等値化する

EVAL デバッガ・コマンドの使用法を示すサンプル・ソースが用意されています。

変数の値の表示

変数の値を表示または変更するには、以下のように行います。

- 「モジュール・ソースの表示」画面に表示されるモジュールは、デバッガ・セッション内のプログラムにバインドする必要があります。
- プログラムは、停止点またはステップ位置で呼び出しなお停止する必要があります。
EVAL デバッガ・コマンドで使用される変数の有効範囲は、QUAL デバッガ・コマンドを使用して定義されます。
- EVAL デバッガ・コマンド
- F11 (変数の表示)

画面間のトグル・スイッチとして Enter キーを使用できます。

注：代入を指定した EVAL デバッガ・コマンドを使用することによって、変数を変更することができま

F11 を使用した変数の表示

データまたは式を表示する最も簡単な方法は、「モジュール・ソースの表示」画面で F11 (変数の表示) を使用することです。表示する変数の上でカーソルを置いて、F11 を押します。変数の現行値が、「モジュール・ソースの表示」画面の最下部にあるメッセージ行に表示されます。

構造体、レコード、クラス、配列、ポインター、列挙、ビット・フィールド、共用体、または関数を評価している場合には、F11 (変数の表示) を押した時に返されるメッセージが数行にわたることがあります。複数行にわたるメッセージは「評価式」画面で表示され、メッセージのテキスト全体が示されます。「評価式」画面でメッセージの表示を止めたら、実行キーを押して「モジュール・ソースの表示」画面に戻ります。

「評価式」画面には、入力した過去のすべてのデバッガ・コマンドおよびそのコマンドの結果も表示されま
す。「評価式」画面で指定ストリングまたはテキストを前方または後方に検索したり、デバッガ・コマンドを再取得または再発行したりすることができます。

例:

この例では、F11 (変数の表示) を使用して変数を表示する方法を示します。
1. DSPMDSRC と入力します。「モジュール・ソースの表示」画面が表示されます。
2. display module T1520IC2 と入力します。

5-36 IBM i: ILE C/C++ プログラマーの手引き
3. 行 50 の変数 hold_formatted_cost の上にカーソルを置き、F11 (変数の表示) を押します。以下のメッセージ行に配列へのポインタが表示されます。

プログラム： T1520PG1 ライブラリ： MYLIB エージェント： T1520IC2
47  if (j<0) return(0);
48  if (hold_formatted_cost[i] == '$')
49  {
50      formatted_cost[j] = hold_formatted_cost[i];
51      break;
52  }
53  if (i<16 && !((i-2)%3))
54  {
55      formatted_cost[j] = ',';
56      --j;
57  }
58  formatted_cost[j] = hold_formatted_cost[i];
59  --j;
60  }

デバッグ... 続く ...

複数行の応答があるメッセージでは、「評価式」画面が表示されます。この画面では、すべての応答行が表示されます。また、入力したデバッグ・コマンドの履歴およびそのコマンドの結果も表示されます。「モジュール・ソースの表示」画面に戻るには、Enter キーを押します。画面間のトグル・スイッチとして Enter キーを使用できます。単一行の応答は、「モジュール・ソースの表示」のメッセージ行に表示されます。

EVAL デバッグ・コマンドを使用して、式の値を判別することもできます。例えば、j に値 1024 が含まれている場合に、デバッグ・コマンド行で eval (j * j)/512 と入力できます。QUAL デバッグ・コマンドを使用して、EVAL デバッグ・コマンドで変数のスコープにする関数内の行またはステートメント番号を判別します。「評価式」画面に、(j * j)/512 = 2048 と表示されます。

変数の値の変更
代入を指定した EVAL デバッグ・コマンドを使用することによって、変数を変更することができます。
EVAL コマンドのスコープを指定するには、QUAL コマンドを使用します。

例:
この例では、EVAL デバッグ・コマンドを使用して式を変数に代入する方法を示します。
1. DSPMODSRC と入力します。「モジュール・ソースの表示」画面が表示されます。
2. display module T1520IC2 と入力します。
3. EVAL コマンドのスコープを指定するために、QUAL コマンドを使用できます。例えば、QUAL 48 を使用すると、行 48 があるスコープに EVAL コマンドが限定されます。行 48 は、以下の EVAL デバッグ・コマンドで変数のスコープにする関数内の数字です。

注: EVAL デバッグ・コマンドの前に QUAL デバッグ・コマンドを必ず使用する必要があるわけではないはずです。ブレークポイントが発生するか、もしくはステップが実行されたときに、自動 QUAL が行われます。これにより、有効範囲規則のデフォルトが現在の停止位置に設定されます。
4. 示されているモジュールの式を変更するには、EVAL x=<expr> と入力します。ここで、x は変数名で、<expr> は変数 x に代入する式です。
スカラー変数の値の変更

割り当て演算子 (=) を指定した EVAL デバッグ・コマンドを使用してスカラー変数の値を変更します。この値を変更するには、ブレークポイントまたはステップ位置でプログラムを呼び出し、停止する必要があります。変数の値を変更するには、次のように入力します。

EVAL variable-name = value

ここで、variable-name は変更したい変数の名前であり、value は変数 variable-name に代入する ID、リテラル、または定数値です。

例:
EVAL COUNTER=3

によって、COUNTER の値は 3 に変わり、また「モジュール・ソースの表示」画面のメッセージ行に
COUNTER=3 =

と表示されます。

文字変数に値を割り当てる時には、次の規則が適用されます。
• 割り当て元の文字列が割り当て先の文字列より短い場合には、割り当て先の式でデータが左寄せされ、残りの桁にブランクが埋め込まれます。
• 割り当て元の文字列が割り当て先の文字列より長い場合には、割り当て先の式でデータが左寄せされ、割り当て先の式の長さまでに切り捨てられます。

EVAL デバッグ・コマンドで使用される変数の有効範囲は、QUAL デバッグ・コマンドを使用して定義されます。行 48 の変数を変更するには、QUAL 48 と入力します。行 48 は、EVAL デバッグ・コマンド用の変数の有効範囲の対象となる関数内の値です。
注: EVAL デバッグ・コマンドの前に QUAL デバッグ・コマンドを必ず使用する必要があるわけではありません。ブレークポイントが発生するか、もしくはステップが実行されたときに、自動 QUAL が行われます。これにより、有効範囲規則のデフォルトが現在の停止位置に設定されます。

以下の例は、1 の配列エレメントを $ から # に変更した結果を示しています。

```c
EVAL hold_formatted_cost [1] = '#'
    hold_formatted_cost[i] = '#':

//Code evaluated before statement 51 where a breakpoint is set
47    if (j<0) return(0);
48    if (hold_formatted_cost[i] == '$')
49        {
50            formatted_cost[j] = hold_formatted_cost[i];
51            break;
52        }
53    if (i<16 && (!((i-2)%3))
54        {
55            formatted_cost[j] = ',';
56            --j;
57        }
58    formatted_cost[j] = hold_formatted_cost[i];
59    --j;
60 }
```

図 5-11. EVAL を使用した変数の変更

変数、式、またはデバッグ・コマンドを指定した名前の等価化

簡略化した使用のために、変数、式、またはデバッグ・コマンドを指定して名前の等価化できます。その後、単独で、または別の式内でその名前を使用できます。別の式で使用した場合には、名前の値は式が評価される前に判別されます。等価化は、デバッグ・セッションが終了するか、名前が除去されるまで活動状態のままになります。

例:

この例では、変数名を指定して Equate デバッグ・コマンドを使用する方法を示します。
1. DSCMODSRC と入力します。「モジュール・ソースの表示」画面が表示されます。
2. 式を等価化するために、equate <name> <definition> と入力します。ここで、<name> はプランクが含まれていない文字ストリングであり、<definition> は少なくとも 1 つのプランクで <name> から区切られた文字ストリングです。文字ストリングは、大文字、小文字、または大/小文字混合にすることができます。合計した文字ストリングの長さは、コマンド行の長さの 144 文字に制限されています。等価化が展開された後に、長さは、最大コマンド長の 150 文字に制限されます。例えば、equate dv display variable と入力します。

定義が指定されておらず、以前の等価化デバッグ・コマンドがその名前を定義していた場合、前の定義が除去されます。名前に前に定義されていなかった場合には、エラー・メッセージが表示されます。

現行デバッグ・セッションで定義されている等価化を表示するには、display equate と入力します。「評価式」画面に活動状態の等価化のリストが表示されます。
構造体の表示
以下の例では、2つのエレメントが表示されている構造体を示します。構造体の各エレメントは、タイプに従ってフォーマット設定されて表示されています。

1. DSPMODSRC と入力します。「モジュール・ソースの表示」画面が表示されます。
2. 行 9 に停止点を設定します。
3. F12 (再開) を押して、「モジュール・ソースの表示」画面を終了します。
4. プログラムを呼び出してください。プログラムが行 9 の停止点で停止します。
5. 以下に示すように、デバッグ・コマンド行で eval test と入力します。

```
<table>
<thead>
<tr>
<th>モジュール・ソースの表示</th>
</tr>
</thead>
</table>
| struct {
  2  char charValue;
  3  unsigned long intValue;
  4  } test;
  5 
  6  int main(){
    7    test.intValue = 10;
    8    test.charValue = 'c';
    9    test.intValue = 11;
  10  return 0;
  11 } |
```

6. Enter を押して、次の画面に進みます。以下に示すように、「評価式」画面に構造体全体が示されます。

```
評価式
> BREAK 9
> EVAL test
test.charValue = 'c'
test.intValue = 10
```

7. 「評価式」画面で Enter を押して、「モジュール・ソースの表示」画面に戻ります。

16 進値としての変数の表示
以下の例では、16 進変数のダンプに使用されるステップおよび構文を示します。

1. DSPMODSRC と入力します。「モジュール・ソースの表示」画面が、以下に示すように表示されます。
2. 行 9 に停止点を設定します。
3. F12 (再開) を押して、「モジュール・ソースの表示」画面を終了します。
4. プログラムを呼び出してください。プログラムが行 9 の停止点で停止します。
5. 以下に示すように、デバッグ・コマンド行で eval test: x 32 と入力します。
プログラム : TEST1 ドイド : DEBUG モジュール : MAIN

1 struct {
2   char charValue;
3   unsigned long intValue;
4 } test;

5 int main(){
6   test.intValue = 10;
7   test.charValue = 'c';
8   test.charValue = 11;
9   return 0;
10 }

6. 「評価式」画面が表示されます。要求したように、32 バイトが表示されていますが、最初の 8 バイトのみが意味のあるものになっています。左のカラムは、変数の先頭からの 16 進オフセットです。右のカラムは、データの EBCDIC 文字表現です。「x」の後に長さを指定しない場合、変数の長さが長さとして使用されます。最小で 16 バイトが表示されます。Enter キーを押すと、「モジュール・ソースの表示」画面に戻ります。

評価式
> BREAK 9
> EVAL test: x 32
00000 8300000 00000000 00000000 - c............
00010 0000000 00000000 00000000 - ............

ヌル終了文字配列の表示
以下の例では、文字ストリングの表示を示します。配列は、「*」演算子によって間接参照する必要があります。* 演算子が入力されていない場合は、配列は、スペース・ポインターとして表示されます。間接参照演算子が使用されているが、「:s」が式に追加されていない場合は、最初の配列要素のみが表示されます。

1. デバッグ・セッション中に、DSPMODSRC と入力します。「モジュール・ソースの表示」画面が表示されます。
2. 行 6 に停止点を設定します。
3. F12 (再開) を押して、「モジュール・ソースの表示」画面を終了します。
4. プログラムを呼び出してください。プログラムが行 6 の停止点で停止します。
5. 以下に示すように、デバッグ・コマンド行で eval *array1: s と入力します。
モジュール・ソースの表示
プログラム： TEST3 ドライバー： DEBUG モジュール： MAIN
1 #include <string.h>
2 char array1 [11];
3 int i;
4 int main()
5 { 
6  strcpy(array1,"0123456789"); 
7  i = 0;
8  return 0;
9 }
デバッグ  .. eval *array1: s = "0123456789" ...
F3= プログラム終了  F6= 停止点の追加 / 消去  F10= ステップ  
F11= 変数の表示  F12= 再開  F17= 変数監視  F24= キーの続き

以下の例では、ストリング出力の表示時に改行文字 (x'15') をスキャンするように指定する :f 構文の使用法を示します。表示行の末尾になると、出力は次の表示行に折り返されます。

:f フォーマット設定コードを使用すると、テキスト・ストリングは、改行が検出されるまで現在行に表示されます。表示画面行の末尾に達するまでに改行文字が検出されなかった場合、出力は改行が検出された時点で折り返されます。必要に応じて DBCS SO/SI 文字を追加して、確実に一致するようにすることができます。

:f フォーマット・コード使用法の例を以下に示します。

```
int main()
{
  char testc[]="This is the first line.
This is the second line.
This is the third line."
  int i;
  i = 1;
}
```

このプログラムの結果として、以下の画面出力になります。
文字配列の表示
以下の例では、文字配列としてフォーマット設定するための「:c」構文の使用法を示します。配列は、
「*」演算子によって間接参照する必要があります。「*」演算子が入力されていない場合は、配列は、スペース・ポインタとして表されます。間接参照演算子が使用されているが、「:c」が式に追加されていない
場合は、最初の配列エレメントのみが表示されます。デフォルトの表示長は 1 です。
1. デバッグ・セッション中に、DSPMODSRC と入力します。「モジュール・ソースの表示」画面が表示され
ます。
2. 行 6 に停止点を設定します。
3. F12 (再開) を押して、「モジュール・ソースの表示」画面を終了します。
4. プログラムを呼び出してください。プログラムが行 6 の停止点で停止します。
5. 以下に示すように、デバッグ・コマンド行で eval *array1: c 11 と入力します。

プログラム： TEST3 デバッグ： DEBUG モジュール： MAIN
1 #include <string.h>
2 char array1 [11];
3 int i;
4 int main(){
5 strcpy(array1,"0123456789");
6 i = 0;
7 return 0;
8 }

デバッグ . . eval *array1: c 11 終わり
F3= プログラム終了 F6= 停止点の追加 / 消去 F10= ステップ
F11= 変数の表示 F12= 再開 F17= 変数監視 F24= キーの続き

以下に、11 文字 (ヌル文字を含む) の表示を示します。ヌル文字は、プランクとして表示されます。

プログラム： TEST3 デバッグ： DEBUG モジュール： MAIN
1 #include <string.h>
2 char array1 [11];
3 int i;
4 int main(){
5 strcpy(array1,"0123456789");
6 i = 0;
7 return 0;
8 }

デバッグ. . .
F3= プログラム終了 F6= 停止点の追加 / 消去 F10= ステップ
F11= 変数の表示 F12= 再開 F17= 変数監視 F24= キーの続き
*array1: c 11 = '0123456789' ...
Pointers
// Display a pointer
>eval pc1
   pc1 = SPP:0000C0260900107C

// Assign a value to a pointer
>eval pc2=pc1
   pc2=pc1 = SPP:0000C0260900107C

// Dereference a pointer
>eval *pc1
   *pc1 = 'C'

// Take the address of a pointer
>eval &pc1
   &pc1 = SPP:0000C02609001040

// Build an expression with normal C precedence
>eval *(short *)pc1
   *(short *)pc1 = -15616

// Casting a pointer
>eval *(short *)pc1
   *(short *)pc1 = -15616

// Treat an unqualified array as a pointer
>eval arr1
   arr1 = SPP:0000C02609001070

// Apply the array type through dereferencing
// (character in this example)
>eval *arr1
   *arr1 = 'A'

// Override the formatting of an expression that is an lvalue
>eval *arr1:s
   *arr1:s = "ABC"

// Set a pointer to null by assigning 0
>eval pc1=0
   pc1=0 = SYP:*NULL

// Evaluate a function pointer
>eval fncptr
   fncptr = PRP:0000A0CD0004F010

// Use the arrow operator
>eval *pY->x.p
   *pY->x.p = ''

Simple Variables
// Perform logical operations
>eval i1==u1 || i1<u1
   i1==u1 || i1<u1 = 0 // Unary operators occur in proper order
>eval i1++
   i1++ = 100
// i1 is incremented after being used
>eval i1
i1 = 101

// i1 is incremented before being used
>eval ++i1
++i1 = 102

// Implicit conversion
>eval u1 = -10
u1 = -10 = 4294967286

// Implicit conversion
>eval (int)u1
(int)u1 = -10

Bit Fields
// Display an entire structure
>eval bits
bits.b1 = 1
bits.b4 = 2

// Work with a single member of a structure
>eval bits.b4 = bits.b1
bits.b4 = bits.b1 = 1

// Bit fields are fully supported
>eval bits.b1 << 2
bits.b1 << 2 = 4

// You can overflow bit fields, but no warning is generated
>eval bits.b1 = bits.b1 << 2
bits.b1 = bits.b1 << 2 = 4
>eval bits.b1
bits.b1 = 0

以下の例は、構造体、共用体、および列挙を使用した EVAL コマンドの使用を示しています。構造体、共用体、および列挙は、5-48 ページの『サンプル EVAL コマンドのソース』のソースに基づいています。

注: C++ では、構造体はクラスではなく、単純構造です。

C 構造体、共用体、および列挙のサンプル EVAL コマンド

Structures and Unions
// Cast with typedefs
>eval (struct z *)&zz
(struct z *)&zz = SPP:0000C005AA0010D0

// Cast with tags
>eval *(c *)&zz
(*(c *)&zz).a = 1
(*(c *)&zz).b = SYP:*NULL

// Assign union members
>eval u.x = -10
u.x = -10 = -10

// Display a union. The union is formatted for each definition
>eval u
u.y = 4294967286
u.x = -10

Enumerations
// Display both the enumeration and its value
>eval Color
Color = blue (2)
>eval Number
Number = three (2)
// Cast to a different enumeration
>eval (enum color)Number
(enum color)Number = blue (2)

// Assign by number
>eval Number = 1
Number = 1 = two (1)

// Assign by enumeration
>eval Number = three
Number = three = three (2)

// Use enums in an expression
>eval arr1[one]
arr1[one] = 'A'

System and Space Pointers
// System pointers are formatted
// :1934:QTEMP :11111110
>eval pSYSptr
pSYSptr =
  SYP:QTEUSERSPC
   0011100
// Space pointers return 8 bytes that can be used in
// System Service Tools
>eval pBuffer
pBuffer = SPP:0000071ECD000200

System and Space Pointers
// System pointers are formatted
// :1934:QTEMP :11111110
>eval pSYSptr
pSYSptr =
  SYP:QTEUSERSPC
   0011100
// Space pointers return 8 bytes that can be used in
// System Service Tools
>eval pBuffer
pBuffer = SPP:0000071ECD000200

図5-12. システム・ポインターおよびスペース・ポインター用のサンプル EVAL コマンド

C および C++ 言語のフィーチャーおよび構造で EVAL コマンドを使用できます。ILE ソース・デバッガーは完全なクラスまたは構造体を表示できますが、派生クラスで定義されているフィールドのみが含まれます。派生クラスを特定の基底クラスにキャストすることで、基底クラスを完全に表示できます。

以下の例で、C++ 言語構造を使用した EVAL コマンドの使用を示します。C++ 言語構造は、5-52 ページの「C++ 構造を表示するためのサンプル EVAL コマンドのソース」のソースに基づいています。ソース・デバッガーのオンライン・ヘルプに追加の C++ の例が示されています。
// Follow the class hierarchy (specifying class D is optional)
> EVAL *(class D *)this
  *(class D *)this).__vbp1B = SPP:C40F5E3D7F000490
  *(class D *)this).__vbp1D = SPP:C40F5E3D7F000440
  *(class D *)this).d = 4

// Follow the class hierarchy (without specifying class D)
> EVAL *(D *)this
  *(D *)this).__vbp1B = SPP:C40F5E3D7F000490
  *(D *)this).__vbp1D = SPP:C40F5E3D7F000440
  *(D *)this).d = 4

// Look at a local variable
> EVAL VAR
  VAR = 1

// Look at a global variable
> EVAL ::VAR
  ::VAR = 2

// Look at a class member (specifying this-> is optional)
> EVAL this->f
  this->f = 6

// Look at a class member (without specifying this->)
> EVAL f
  f = 6

// Disambiguate variable ac
> EVAL A::ac
  A::ac = 12

// Scope operator with template
> EVAL E<int>::ac
  E<int>::ac = 12

// Cast with template:
> EVAL *(E<int> *)this
  *(E<int> *)this).__vbp1B = SPP:C40F5E3D7F000490
  *(E<int> *)this).__vbp1EXTi_ = SPP:C40F5E3D7F000400
  *(E<int> *)this).e = 5

// Assign a value to a variable
> EVAL f=23
  f=23 = 23

// See all local variables in a single EVAL statement
> EVAL %LOCALVARS
  local = 828
  this = SPP:C40F5E3D7F000400
  VAR = 1

図5-13. C++ 式用のサンプル EVAL コマンド

クラス・テンプレートおよび関数テンプレートの表示: クラス・テンプレートまたは関数テンプレートを表示するには、デバッグ・コマンド行に EVAL template-name と入力します。変数 template-name は、表示したいクラス・テンプレートまたは関数テンプレートの名前です。

下の例は、クラス・テンプレートの評価の結果を示します。デマングルされたテンプレート名と一致するテンプレート名を入力する必要があります。型定義名は、テンプレート名がマングルされるときに typedef 情報が除去されるため無効です。
サンプル EVAL コマンドのソース

5-44 ベージの「ポインター、変数、およびビット・フィールド用のサンプル EVAL コマンド」および
5-45 ベージの「C 構造体、共用体、および列挙型のサンプル EVAL コマンド」に示されているサンプル
EVAL コマンドは、以下の図に示されているソースに基づいています。

```c
#include <iostream.h>
#include <pointer.h>

/** POINTERS **/
_SYSPTR pSys; //System pointer
_SPCPTR pSpace; //Space pointer
int (*fncptr)(void); //Function pointer
char *pc1; //Character pointer
char *pc2; //Character pointer
int *pi1; //Integer pointer
char arr1[] = "ABC"; //Array

/** SIMPLE VARIABLES **/
int i1; //Integer
unsigned ul; //Unsigned Integer
char c1; //Character
float f1; //Float

/** STRUCTURES **/
struct { //Bit fields
```
typedef struct z { // Structure typedef
    int z;
    char *p;
} z ;
z zz; // Structure using typedef
z *pZZ; // Same

typedef struct c { // Structure typedef
    unsigned a;
    char *b;
} c ;
c d; // Structure using typedef
/** UNIONS **/
union u{ // Union
    int x;
    unsigned y;
};
union u u; // Variable using union
union u *pu; // Same

/** ENUMERATIONS **/
enum number {one, two, three};
enum color {red,yellow,blue};
enum number Number = one;
enum color Color = blue;

/** FUNCTION **/
int ret100(void) { return 100;}
int main()
{
    float dec1;
    struct y y, *pY;
    bits.b1 = 1;
    bits.b4 = 2;
    i1 = ret100();
    cl = 'C';
    f1 = 100e2;
    decl = 12.3;
    pcl = &cl;
    p1 = &i1;
    d.a = 1;
pZZ = &zz;
pZZ->z=1;
pY = &y;
pY->x.p=(char*)&y;
p0=&u;
pU->x=255;
Number=(number)Color;
fncptr = &ret100;
pY->x.x=1; // Set breakpoint here
return 0;
}
システム・ポインターおよびスペース・ポインターを表示するためのサンプル EVAL コマンドのソース

5-46 ページの図 5-12 に示されているシステム・ポインターおよびスペース・ポインターを表示するためのサンプル EVAL コマンドは、以下の図に示されているソースに基づいています。
```c
#include <stdio.h>
#include <mispace.h>
#include <pointer.h>
#include <mispcobj.h>
#include <except.h>
#include <lecond.h>
#include <leenv.h>
#include <qtedbgs.h> /* From qsysinc */
/* Link up the Create User Space API */
#pragma linkage(CreateUserSpace,OS)
#pragma map(CreateUserSpace,"QUSCRTUS")
void CreateUserSpace(char[20],
    char[10],
    long int,
    char,
    char[10],
    char[50],
    char[10],
    _TE_ERROR_CODE_T *
    ) ;
/* Link up the Delete User Space API */
#pragma linkage(DeleteUserSpace,OS)
#pragma map(DeleteUserSpace,"QUSDLTUS")
void DeleteUserSpace(char[20],
    _TE_ERROR_CODE_T *
    ) ;
/* Link up the Retrieve Pointer to User Space API */
#pragma linkage(RetrievePointerToUserSpace,OS)
#pragma map(RetrievePointerToUserSpace,"QUSPTRUS")
void RetrievePointerToUserSpace(char[20],
    char **,
    _TE_ERROR_CODE_T *
    ) ;
int main (int argc, char *argv[])
{
    char *pBuffer;
    _SYSPTR pSYSptr;
    _TE_ERROR_CODE_T errorCode;
    errorCode.BytesProvided = 0;
    CreateUserSpace("QTEUSERSPCQTEMP ",
        "QTESSPC ",
        10,
        "*ALL ",
        "*YES ",
        &errorCode
        ) ;
    /* call RetrievePointerToUserSpace - Retrieve Pointer to User Space */
    /*!! (pass: Name and library of user space, pointer variable */
    /*!! return: nothing (pointer variable is left pointing to start*/
    /*!! of user space) */
    RetrievePointerToUserSpace("QTEUSERSPCQTEMP ",
        pBuffer,
        &errorCode)
    ;
    /* convert the space pointer to a system pointer */
    pSYSptr = _SETSPFP(pBuffer);
    printf("Space pointer: %p\n",pBuffer);
    printf("System pointer: %p\n",pSYSptr);
    return 0;
}
```
C++ 構造を表示するためのサンプル EVAL コマンドのソース

C++

5-47ページの図 5-13 に示されている C++ 構造を表示するためのサンプル EVAL コマンドは、以下の図に示されているソースに基づいています。

// Program demonstrates the EVAL debug command
class A {
    public:
        union {
            int a;
            int ua;
        };
        int ac;
        int amb;
        int not_amb;
    };

    class B {
        public:
            int b;
    };

    class C {
        public:
            int ac;
            static int c;
            int amb;
            int not_amb;
    };

    int C::c = 45;
    template <class T> class E : public A, public virtual B {
        public:
            T e;
    };

    class D : public C, public virtual B {
        public:
            int d;
    };

    class outter {
        public:
            static int static_i;
        class F : public E<int>, public D {
            public:
                int f;
                int not_amb;
                void funct();
        } inobj;
    };

    int outter :: static_i = 45;

    int VAR = 2;

    void outter::F::funct()
    {
        int local;
        a=1; //EVAL VAR : Is VAR in global scope
    }
b=2;
c=3;
d=4;
e=5;
f=6;

local = 828;
int VAR;

VAR=1;
static_i=10;
A::ac=12;
C::ac=13;
not_amb=32;

not_amb=13;
// Stop here and show:
// EVAL VAR : is VAR in local scope
// EVAL ::VAR : is VAR in global scope
// EVAL %LOCALVARS : see all local vars
// EVAL *this : fields of derived class
// EVAL this->f : show member f
// EVAL f : in derived class
// EVAL a : in base class
// EVAL b : in Virtual Base class
// EVAL c : static member
// EVAL static_i : static var made visible
// EVAL au : anonymous union members
// EVAL a=49 : anonymous union members
// EVAL au
// EVAL ac : show ambiguous var
// EVAL A::ac : disambig with scope op
// EVAL B::ac : Scope op
// EVAL E<int>::ac : Scope op
// EVAL this : notice pointer values
// EVAL (E<int>*)this : change
// EVAL (class D *)this : class is optional
// EVAL *(E<int> *)this : show fields
// EVAL *(D *) this : show fields
}

int main()
{
  outer obj;
  int outer::F::*mptr = &outer::F::b;
  int i;
  int& r = i;
  obj.inobj.funct();
  i = 777;

  obj.static_i = 2;
  // Stop here
  // EVAL obj.inobj.*mptr : member ptr
  // EVAL obj.inobj.b
  // EVAL i
  // EVAL r
  // EVAL r=1
  // EVAL i
  // EVAL (A &) (obj.inobj) : reference cast
  // EVAL
}
モジュール最適化およびプログラム識別情報の変更

プログラムの作成後に、問題や修正されたユーザー要件に対応するために、その後のプログラム変更が必要である場合があります。

プログラムが実動の場合、パフォーマンス用に最適化され、最小サイズに削減されます。

プログラムのデバッグ時に、以下が適用されます。
- データを処理する際のプログラムの動作を観察する。
- 高いレベルの最適化では表示されないことがある変数値を表示する。

リスト・ビューを作成する際に、プログラムの動作を監視するために必要なデータを追加します。5-12ページの『デバッグのためのリスト・ビューの作成』を参照してください。

デバッグ・セッション時に、プログラムをデバッグするときには変数を正しく表示するためにモジュールの最適化レベルを低くし、その後プログラムを実動用に使用可能にするときに、プログラムの効率を高めるためにレベルを再び上げることができます。

デバッグ・セッション後に、モジュールのプログラム識別情報を削除して、モジュールのサイズを削減できます。

このトピックでは、以下を行うための方法について説明します。
- デバッグ・セッション次に最適化レベルを変更する。
- モジュールのプログラム識別情報を削除する。

最適化レベルの変更

オブジェクトの最適化とは、コンパイルされたコードを見て、実行時のパフォーマンスをできるだけ高速にするには何ができるかを判断し、必要な変更を行うことを意味します。 一般に、最適化の要求が高くなるにつれて、オブジェクトの作成にかかる時間が長くなります。 実行時には、高度に最適化されたプログラムまたはサービス・プログラムは、対応する最適化されていないプログラムまたはサービス・プログラムより高速で実行されることになります。

例:

この例は、プログラムがデバッグ・モードのときに変数を表示したり変更したりできるようにするために、モジュール T1520IC4 の最適化レベルを *FULL から *NONE に変更する方法を示しています。デバッグが完了したら、実行時のパフォーマンスを向上するために最適化レベルを *FULL に戻すことができます。

1. WRKMOD MODULE(T1520IC1) と入力します。「モジュールの処理」画面が表示されます。
2. オプション 5 (表示) を選択し、変更の必要がある属性値を参照します。「モジュール情報の表示」画面が表示されます。
モジュール情報の表示

<table>
<thead>
<tr>
<th>モジュール</th>
<th>T1520IC1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ライブラリー</td>
<td>MYLIB</td>
</tr>
<tr>
<td>詳細</td>
<td>*BASIC</td>
</tr>
<tr>
<td>モジュール属性</td>
<td>CLE</td>
</tr>
<tr>
<td>モジュール情報：</td>
<td></td>
</tr>
<tr>
<td>モジュール作成日/時刻</td>
<td>93/09/93 12:00:00</td>
</tr>
<tr>
<td>ソース・ファイル</td>
<td>QACSRC</td>
</tr>
<tr>
<td>ライブラリー</td>
<td>MYLIB</td>
</tr>
<tr>
<td>ソース・メンバー</td>
<td>T1520IC1</td>
</tr>
<tr>
<td>ソース・ファイル変更日/時刻</td>
<td>93/08/18 13:31:40</td>
</tr>
<tr>
<td>所有者</td>
<td>SMITH</td>
</tr>
<tr>
<td>コード化文字セット ID</td>
<td>65353</td>
</tr>
<tr>
<td>テキスト記述</td>
<td></td>
</tr>
<tr>
<td>作成データ</td>
<td>*YES</td>
</tr>
<tr>
<td>中間言語データ</td>
<td>*NO</td>
</tr>
</tbody>
</table>

続行するには、実行キーを押してください。
F3= 終了 F12= 取り消し

注: 上記の画面では、作成データの値は *YES です。これは、最適化レベル値が変更されても、モジュールを再度変換できることを意味しています。値が *NO の場合、最適化レベルを変更するためには、モジュールをコンパイルし直す必要があります。

3. 次ページ・キーを押すと、以下のようにモジュールの情報がさらに表示されます。

<table>
<thead>
<tr>
<th>モジュール情報の表示</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>モジュール</td>
<td>T1520IC4</td>
</tr>
<tr>
<td>ライブラリー</td>
<td>MYLIB</td>
</tr>
<tr>
<td>詳細</td>
<td>*BASIC</td>
</tr>
<tr>
<td>モジュール属性</td>
<td>CLE</td>
</tr>
<tr>
<td>ソート順序テーブル</td>
<td>*HEX</td>
</tr>
<tr>
<td>言語 ID</td>
<td>*JOBRUN</td>
</tr>
<tr>
<td>最適化レベル</td>
<td>*NONE</td>
</tr>
<tr>
<td>最大最適化レベル</td>
<td>*FULL</td>
</tr>
<tr>
<td>デバッグ・データ</td>
<td>*YES</td>
</tr>
<tr>
<td>壓縮</td>
<td>*NO</td>
</tr>
<tr>
<td>プログラム入リ口プロシージャー名</td>
<td>_C_pep</td>
</tr>
<tr>
<td>パラメーターの数</td>
<td>0 255</td>
</tr>
<tr>
<td>モジュールの状態</td>
<td>*USER</td>
</tr>
<tr>
<td>モジュール定義域</td>
<td>*SYSTEM</td>
</tr>
<tr>
<td>エクスポートされた定義済み記号の数</td>
<td>1</td>
</tr>
<tr>
<td>インポートされた（未解決）記号の数</td>
<td>10</td>
</tr>
<tr>
<td>続行するには、Enter キーを押してください。</td>
<td></td>
</tr>
</tbody>
</table>

続行 ... F3= 終了 F12= 取り消し

4. 最大最適化レベル値を調べてください。最適化レベル値がすでに希望するレベルになっている場合もあります。モジュールに作成データがあり、最適化レベルを変更したい場合には、F12 (取り消し) を押します。「モジュールの処理」画面が表示されます。

5. 最適化レベルを変更するモジュールの場合には、オプション 2 (変更) を選択します。CHGMOD コマンド・プロンプトが表示されます。

6. 「モジュールの最適化」フィールドに指定されている値を上書き入力してください。モジュールを低いレベルの最適化に変更すると、デバッグ中に変数の値を表示して、場合によっては変更することができるようにになります。コマンド・プロンプトで以下のコマンドを使用して最適化レベルを *NONE に下げることがでできますが、それはジョブ・ログには書き込まれません。

\[
\text{CHGMOD MODULE(MYLIB/T1520IC4) OPTIMIZE(*NONE)}
\]

7. 追加で変更したいモジュールがあれば、ステップ 2 (5-54 ページ) から 4 までを繰り返してください。同じ ILE プログラム内で変更するモジュールが 1 つであっても複数であっても、システムがそれらのモジュールを検出するとすべてのインポートが解決されるので、プログラム作成時間は同じです。
モジュール識別情報の除去

モジュールを識別可能にするには、2 つのタイプのデータがモジュールと一緒に保管されていなければなりません。

この 2 つのタイプのデータとは、以下のものでです。

作成データ

*CRTDTA 値で表されます。このデータはコードを機械語命令に変換するのに必要です。モジュールの最適化レベルを変更するには、モジュールにこのデータがなければなりません。

デバッグ・データ

*DBGDTA 値で表されます。

モジュールをデバッグするには、*CRTDTA と *DBGDTA の両方が必要です。

これらの 2 つのデータ・タイプがモジュールと一緒に保管されている場合にのみ、モジュールを再コンパイルせずに変更することが可能になります。モジュールを再コンパイルした後に、このデータのみを除去することができます。このデータを除去すると、その識別情報も除去され、データを置換するにはモジュールの再コンパイルが必要になります。

すべての識別情報を除去すると、モジュールはその最小サイズに圧縮されます。この場合、モジュールを再コンパイルしない限り、モジュールを変更することはできません。これを再コンパイルするには、ソース・コードへのアクセス権限をもっていなければなりません。

どちらの種類のデータでも、モジュールからの除去には、CHGMOD コマンドを使用できます。

例:

T1520IC4 のプログラムからプログラム識別情報を除去するには、以下の手順に従います。

1. WRKMOD と入力します。「モジュールの処理」画面が表示されます。
2. オプション 5（表示）を選択し、変更の必要がある属性値を参照します。「モジュール情報の表示」画面が表示されます。

「作成データ」フィールドの値を調べてください。値が *YES の場合、作成データが存在し、除去不可能です。この値が *NO の場合、除去する作成データがありません。モジュールは、再作成しない限り、再び変換することはできません。
3. 次ページ・キーを押すと、モジュールの情報がさらに表示されます。「デバッグ・データ」フィールドの値を調べてください。値が *YES の場合、モジュールはデバッグ可能です。値が *NO の場合、モジュールを再コンパイルしてデバッグ・データを組み込まない限り、そのモジュールをデバッグすることはできません。F3 を押して「モジュールの処理」画面に戻ります。
4. 識別情報を変更するモジュールについて、オプション 2（変更）を選択してください。CHGMOD コマンド・プロンプトが表示されます。
5. 識別情報除去 プロンプトに指定されている値を上書き入力してください。Enter キーを押すと、モジュールの変更コマンドについて、ジョブ・ログに以下のコマンドが示されます。

CHGMOD MODULE(MYLIB/T1520IC4) RMVOBS(*ALL)
6. 「モジュールの強制再作成」 パラメーターの値を *YES に変更することで、モジュールが必ず再作成されるようにすることができます。

注: 単に最適化レベルが変更されるだけでは、このパラメーターは必要ありません。作成データが除去されていない限り、最適化レベルの変更は通常、結果的にモジュールの再作成になります。デバッグ・データを除去した後、最適化レベルは変更せずにプログラムを再変換する場合には、モジュールの強制再作成 パラメーターを使用しなければなりません。

7. 変更する追加モジュールについて、ステップ 2 (5-56 ページ) から 5 (5-56 ページ) を再び行います。同じ ILE プログラム内で変更するモジュールが１つであっても複数であっても、システムがそれらのモジュールを検出するとすべてのインポートが解決されるので、プログラム作成時間は同じです。

注: CRTPGM コマンドの *UNRSLVREF パラメーターを使用して、インポートを未解決のままにすることができます。プログラム作成時間は同じです。

8. CRTPGM コマンドを使用して ILE プログラムを再作成します。
入出力操作の実行

このトピックでは、以下を行うための方法について説明します。

- ファイルで ILE C/C++ のストリーム入出力関数およびレコード入出力関数を使用する
- 統合ファイル・システム (IFS) で ILE C/C++ ストリーム入出力関数を使用する

IBM i ファイルでの ILE C/C++ ストリームおよびレコード入出力関数の使用

ILE C/C++ コンパイラでは、プログラムでストリーム・ファイルをテキスト・ストリーム・ファイルまたはバイナリー・ストリーム・ファイルとして処理できます。6-6 ページの『テキスト・ストリームおよびバイナリー・ストリームのファイル制御構造』を参照してください。

このトピックでは以下について説明します。

- ILE C レコード入出力関数
- IBM i ファイル
  - テキスト・ストリームおよびバイナリー・ストリームのファイル制御構造
  - テキスト・ストリーム・ファイルの入出力プロセス
  - バイナリー・ストリーム・ファイルの入出力プロセス
  - オープン・フィードバック域
  - 入出力フィードバック域
- セッション管理機能の使用方法

ILE C レコード入出力関数

ILE C ライブラリーでは、入出力の ISO C 定義に対する拡張セットが提供されています。この拡張セットは、レコード入出力と呼ばれるもので、プログラムがレコード単位で入出力操作を実行できるようにします。

ILE C のレコード入出力関数は、IBM i プラットフォームでサポートされるすべてのファイル・タイプで動作します。

_Ropen() でオープンした各ファイルには、タイプ _RFILE の構造が関連付けられています。<recio.h> ヘッダー・ファイルで、この構造が定義されます。

重要：この構造を変更しようと思うと、予期しない結果が発生する可能性があります。

IBM i データ管理ファイル・タイプにはそれぞれ異なるオープン・モードとキーワード・パラメーターが適用されます。各ファイル・タイプと、_Ropen() でレコード・ファイルをオープンする方法については、以下を参照してください。

- 7-1 ページの『プログラムでの外部記述ファイルの使用』
- 7-22 ページの『プログラムでのデータベース・ファイルおよび分散ファイルの使用』
ストリーム・バッファリング

ISO 標準 C ストリームには、3 つのバッファリング方式が定義されています。それらは次のとおりです。

・バッファリングなし - 文字は、ソースから、または先に可能な限り速やかに表示されるようになって
   います。ILE C コンパイラーは、バッファリングなしストリームをサポートしません。

・完全バッファリング - バッファーがいっぱいになった後で、文字が一度に 1 ブロック単位でファイルと
   の間で送られます。ILE C コンパイラーは、ブロックをシステム・ファイルのレコード・サイズとして
   抜います。

・行バッファリング - 改行制御文字 (Yn) が見つかると、ファイルとの間で文字がブロックとして送られ
   ます。

ブロックと行は開かれたファイルのレコード長と等しいため、ILE C コンパイラーは、完全バッファリン
グと行バッファリングのストリームを同様にサポートします。

注: setbuf() および setvbuf() 関数では、システム使用時のバッファリングおよびバッファー・サイズの
制御は許可されません。

動的ストリーム・ファイルの作成
テキスト・ストリーム・ファイルの動的ファイルの作成は、以下を指定するのと同じです。

CRTSRCPF FILE(filename) RCDLEN(recln)

バイナリー・ストリーム・ファイルの動的ファイルの作成は、以下を指定するのと同じです。

CRTPF FILE(filename) RCDLEN(recln)

fopen() の lrecl パラメーターで指定された長さは、以下の例外を除き、作成されるファイルのレコード
長に使用されます。

・テキスト・ファイルを開く際にレコード長を指定しなかった場合、レコード長が 266 のソース物理ファ
イルが作成されます。

・バイナリー・ファイルまたはレコード・ファイルを開く際にレコード長を指定しなかった場合、レコー
ド長が 80 の物理ファイルが作成されます。

・テキスト・ファイルを開く際にレコード長としてゼロを指定した場合 (lrecl=0)、レコード長が 266 の
ソース物理ファイルが作成されます。

・バイナリー・ファイルを開く際にレコード長としてゼロを指定した場合 (lrecl=0)、レコード長が 80 の
物理ファイルが作成されます。

・プログラム記述ファイルで lrecl パラメーターが指定されていない場合、CRTPRTG または CRTPRTF
で指定されているレコード長が使用されます。この長さのデフォルト値は 132 であり、指定する場合は
1 以上でなければなりません。

注: 原始ステートメント入力ユーティリティー (SEU) を使用してソース・ファイルを編集するには、
fopen() で lrecl の値として 240 文字以下を指定してください。

6-2 IBM i: ILE C/C++ プログラマーの手引き
動的に作成されたストリーム・ファイルのオープン・モード
ファイルを開く際にモードを指定し、指定したファイルが存在しない場合、IBM i プラットフォームはそのファイルを自動的に作成します。
- バイナリー・モードを使用した場合、物理データベース・ファイルが作成されます。
- テキスト・モードを使用した場合、ソース物理ファイルが作成されます。
ファイルが存在するが、メンバーが存在しない場合は、IBM i プラットフォームはメンバーをファイルに追加します。
ファイルを開く際にライブラリーファイル名を指定しなかった場合、ライブラリー QTEMP にデータベース・ファイルが動的に作成されます。メンバー名を指定しなかった場合、ファイルと同じ名前でメンバーが作成されます。

標準入出力テキスト・ストリーム・ファイル (<stdio.h>)
<stdio.h> ファイルをインクルードするプログラムが開始するとき、次の 3 つのテキスト・ストリームが定義されます。
- 標準入力 (stdin) による端末からの入力の読み取り。
- 標準出力 (stdout) による端末への出力の書き込み。
- 標準エラー出力 (stderr) による端末への診断出力の書き込み。

stdin、stdout、および stderr のストリームは、それらが最初に使用されるときに暗黙的にオープンされます。
- ストリーム stdin は fopen("stdin", "r") でオープンされます。
- ストリーム stdout は fopen("stdout", "w") でオープンされます。
- ストリーム stderr は fopen("stderr", "w") でオープンされます。

注: これらのストリームは、本当の IBM i ファイルではありませんが、ILE C ライブラリーオペレーション数ではファイルとしてシミュレートされます。デフォルトで、これらの端末セッションに送信されます。

端末への標準出力のオーバーライド: stdin、stdout、および stderr ストリームは、それぞれファイル
stdin、stdout、および stderr に対して IBM i オーバーライド・コマンドを使用して他の装置に関連付けることができます。stdin、stdout、および stderr が使用され、これらのストリームを開く前にそのいずれかにファイル・オーバーライドが存在する場合、そのオーバーライドが効力を持ち、入出力操作が端末に送られないことがあります。

stdout または stderr が非対話式ジョブで使用され、そのストリームにファイル・オーバーライドがない場合、ILE C コンパイラは印刷出力ファイル QPRINT へストリームをオーバーライドします。出力は、ワークステーションに表示されるのではなく、印刷されるか、印刷用にスプールされます。

プログラムによる QINLINE を指定した入力ファイルの再読み取りを可能にする: 対話式ジョブの一部でない入力ファイルに stdin を指定した場合（またはデフォルトを受け入れた）場合、QINLINE ファイルが使用されます。QINLINE を指定したファイルを再読み取りすることはできません。データベース読み取りプログラムはこれをファイル名のないファイルとして扱うからです。したがってこのファイルを 2 度読み取ることはできません。オーバーライドを発行すると、これを避けすることができます。stdin から文字を読み取っている場合、F4 を押すと、ランタイムは保留中の入力をすべて終了し、EOF 標識をオフに設定することになります。F3 を押すことは、ILE C/C++ プログラムから exit() を呼び出すことと同じです。

stdin をパッチで指定し、それに関連したオーバーライドがない場合、QINLINE が使用されます。stdin に関連のオーバーライドがある場合、QINLINE の代わりにそのオーバーライドが使用されます。
注: `freopen()` を使用して、テキスト・ストリームを再度開くこともできます。 `stdout` および `stderr` ストリームは、印刷出力ファイルおよびデータベース・ファイル用に再度開くことができます。 `stdin` ストリームは、データベース・ファイルでしかオーバーライドすることはできません。 `freopen()` を使用した `stdin/stdout/stderr` と `IFS` ストリーム・ファイルとの間のリダイレクトはサポートされていません。

**IBM i ファイル**

**ILE C** ストリーム・ファイルまたはレコード・ファイルは、IBM i ファイルと同じです。システム・ファイルは、ファイル・オブジェクトとも呼ばれます。各 IBM i ファイルまたはファイル・オブジェクトは、その中に保管された情報によって区別および分類されます。各ファイルには、それぞれに固有の一連の特性があり、これによってファイルの使用方法およびファイルの提供機能が決まります。この情報は、ファイル記述と呼ばれます。

また、ファイル記述には、ファイルの特性のほか、ファイルに関連付けられたデータのレコードへの編成方法やこれらのレコード内でのフィールドの編成方法に関する詳細も含まれます。オペレーティング・システムは、常にこのファイル記述を使用してファイルを処理します。データは、ファイル・オブジェクトを通じてシステムで作成され、アクセスされます。

**IBM i ファイル・タイプ**

IBM i ファイルを以下にリストしています。

- データベース・ファイル。IBM i プラットフォーム上でデータを保管します。
- 装置ファイル。ディスプレイ装置、プリンター、テープ装置、ディスケット装置などの外部接続装置にアクセスできるようにします。
- システム間通信機能 (ICF) ファイル。異なるシステム上の 2 つのアプリケーション・プログラムの間で送受信されるデータのレイアウトを定義します。このファイルは、リモート・システムとの通信に使用される構成オブジェクトをリンクします。
- 保管ファイル。データをバックアップおよびリカバリーの目的で使用される形式で保存します。
- 分散 (DDM) ファイル。リモート・システム上のデータにアクセスします。

**ストリーム・ファイルおよび ILE C 入出力操作**

**C** 国際標準では、C 言語のストリーム・ファイルを、1 度に 1 文字ずつ読み書きされるデータのシークエンスとして定義しています。ISO C でのすべての入出力操作は、ストリーム操作です。

IBM i プラットフォームでは、以下のとおりです。

- ストリームは、連続した文字ストリングです。
- すべてのファイルは、レコードで構成されます。
- オペレーティング・システム・レベルでのすべての入出力操作は、操作を使用して 1 度に 1 レコードずつ実行されます。

**ILE C/C++** ランタイム・ライブラリーでは、プログラムでストリーム・ファイルをテキスト・ストリーム・ファイルまたはバイナリ・ストリーム・ファイルとして処理できます。テキスト・ストリーム・ファイルは、1 度に 1 文字ずつ処理します。バイナリ・ストリーム・ファイルは、1 度に 1 文字ずつ、または 1 度に 1 レコードずつ処理します。

IBM i プラットフォームが 1 度に 1 レコードずつ入出力操作を実行するため、ILE C/C++ ライブラリーは、IBM i のレコードを使用して、ストリーム・ファイル処理をシミュレートします。 ILE C/C++ ライブラリーは 1 度に 1 文字ずつ入出力を論理的に処理しますが、オペレーティング・システムで実行される実際の入出力は、1 度に 1 レコードずつ行われます。
ファイルでの位置決めの問題の回避: オペレーティング・システムは一度に1レコードずつ入出力操作を実行するため、同じファイルに対してストリーム入出力操作とともに OPNORBYF などのシステム・コマンドを使用すると、プログラムが処理しているファイルで位置決めの問題が生じることがあります。

注意:
- 予測不能な結果が発生する可能性があるため、同じファイルに対してレコード入出力の ILE C/C++ 拡張とストリーム・ファイル関数を混用しないでください。
- ストリーム入出力操作が含まれたプログラムでは、文字の代わりにレコードを論理的に処理するシステム・コマンドを使用しないでください。

fopen() 関数の使用: `fopen()` の形式は次のとおりです。

```c
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

`mode` 変数は、キーワード・パラメーターを後に続けることができるオープン・モードで構成される文字ストリングです。オープン・モードとキーワード・パラメーターは、1つのコンマまたは1つ以上のプランク文字で区切る必要があります。

注: `recln` パラメーターについて詳しくは、6-2 ページの『動的ストリーム・ファイルの作成』を参照してください。

open() メンバー関数の使用:

### C++

入力、出力、または入出力のファイル・ストリームを作成し、ファイルにリンクします。ファイル・ストリーム・クラスの open() メンバー関数を使用して、ファイル・ストリームをファイルにリンクします。

open() メンバー関数の形式は以下のとおりです。

```c
#include <iostream>
#include <fstream>

void ifstream::open(const char *filename, openmode mode = ios::in);
void ofstream::open(const char *filename, openmode mode = ios::out | ios::trunc);
void fstream::open(const char *filename, openmode mode);
```

### IBM i のファイル命名規則

IBM i ファイルを参照する _Ropen() および fopen() 関数には、ファイル名が必要です。このファイル名は、ヌル終了ストリングでなければなりません。

ファイル名の構文は、以下のとおりです。

```
library-name/file-name (member-name)
```

- **library-name**: ファイルが含まれているライブラリーの名前を入力します。ライブラリーを指定しなかった場合は、システムはジョブのライブラリー・リストでファイルを検索します。
- **filename**: ファイルの名前を入力します。これは必須パラメーターです。
- **member-name**: ファイル・メンバーの名前を入力します。メンバー名を指定しなかった場合、最初のメンバー (*FIRST) が使用されます。
注: fopen() および _Ropen() の使用時に member-name に *ALL を指定した場合、複数メンバー処理が行われます。

円記号と引用符 (" ) の制御シーケンスでストリーミングを囲まない限り、library-name、filename、または member-name に指定する文字はすべて、大文字に変換されます。これにより、IBM i の引用符付き名前を指定できます。以下に例を示します。
""tstlib"/tstfile(tstmbbr)"

Library is:  "tstlib"
File is:  TSTFILE
Member is:  TSTMBR

ファイル名、ライブラリー名、またはメンバー名を二重引用符で囲むと、名前が通常名であった場合でも、二重引用符は ILE C/C++ コンパイラによって破棄されます。通常名とは、以下の文字で構成された任意のファイル、ライブラリー、またはメンバー名です。
- 大文字
- 数値
- $ (16 進数値 0x5B)
- @ (16 進数値 0x7C)
- # (16 進数値 0x7B)
- _ (16 進数値 0x6D)
- . (16 進数値 0x4B)

ファイル名、ライブラリー名、またはメンバー名のどの位置に以下文字は使用できません。

正しくない文字
16 進数表現
( 0x4D
* 0x5C
) 0x5D
/ 0x61
? 0x62
0x7D
" 0x7F
(プランク) 0x40

注: "( )/" は、引用符で囲まれたファイル名で使用できます。

テキスト・ストリームおよびバイナリー・ストリームのファイル制御構造
テキスト・ストリームとバイナリー・ストリームはどちらも、IBM i ファイル内のレコードにマップされます。これらは、文字単位またはレコード単位で処理できます。
図6-1. IBM i レコードの ILE C ストリーム・ファイルへのマッピング

"table"}

<table>
<thead>
<tr>
<th>定義</th>
<th>テキスト・ストリーム</th>
<th>バイナリー・ストリーム</th>
</tr>
</thead>
<tbody>
<tr>
<td>行で構成される、配列された文字シーケンス。各行は、ゼロ個以上の文字で構成され、改行文字で終了します。</td>
<td>関連付けられた IBM i ファイルに格納された文字と 1 対 1 の対応を持つ文字シーケンス。IBM i プラットフォームで、バイナリー・ストリーム・ファイルの長さはレコード長の倍数になります。</td>
<td></td>
</tr>
<tr>
<td>IBM i は、入出力中にいくつかの特殊文字を追加、変更、または削除する可能性があります。そのため、テキスト・ストリームに書き込まれた文字と、同じテキスト・ストリームから読み取った文字が、1 対 1 で対応しない場合があります。以下のようなすべてを満たす場合、テキスト・ストリームから読み取ったデータは、テキスト・ストリームに書き込まれたデータと同じになります。</td>
<td>バイナリー・ストリームに対して文字変換は行われません。バイナリー・ストリームに書き込まれたデータは、後で読み取ったときに同じものになります。注: バイナリー・ストリームでは、改行文字は特別な意味を持ちません。</td>
<td></td>
</tr>
<tr>
<td>データが、印刷可能文字、水平タブ、垂直タブ、改行文字、または用紙送り制御文字から成る。</td>
<td>バイナリー・ストリームの最終の文字が改行文字である。</td>
<td></td>
</tr>
<tr>
<td>改行文字の直前にスペース（ブランク）文字がない。</td>
<td>ファイルに書き込まれた行が、ファイルのレコード長を超えない。</td>
<td></td>
</tr>
<tr>
<td>ストリームの最後の文字が改行文字である。</td>
<td>ファイルのクローズ時、ファイルの末尾に既に改行文字が指定されていなければ、暗黙的な改行文字が付加されます。</td>
<td></td>
</tr>
</tbody>
</table>

ファイルの終了処理

ファイルのクローズ時、ファイルの末尾に既に改行文字が指定されていなければ、暗黙的な改行文字が付加されます。|

ファイルのクローズ時、ファイルの最終レコードで、レコードの末尾にヌル（16進数値 0x00）が埋め込まれます。

テキスト・ストリーム・ファイルの入出力プロセス

この節では、以下的方法について説明します。

- テキスト・ストリーム・ファイルを開く

入出力操作の実行 6-7
テキスト・ストリーム・ファイルを開く操作

IBM i ファイルをテキスト・ストリーム・ファイルとして開くには、以下のいずれかのモードで fopen() を使用します。

- r
- r+
- w
- w+
- a
- a+

注:
1. fopen() で同時に開くことができるファイル数は、使用可能なシステム・ストレージ量によって異なります。
2. fopen() 関数のオープン・モードは、freopen() 関数にも適用されます。
3. テキストストリーム・ファイルに削除されたレコードが含まれている場合、削除されたレコードは、テキストストリーム入出力関数によってスキップされます。

有効なキーワード・パラメーターは次のとおりです。

- lrecl
- ccsid
- recfm (F、FA、および FB のみ)

fopen() で無効なモードまたはキーワード・パラメーターを指定した場合、errno が EBADMODE に設定され、NULL が返されます。

例:

以下の例では、テキスト・ストリーム・ファイルを開く方法を示します。ライブラリー MYLIB が存在している必要があります。ファイル TEST が存在しない場合、そのファイルが自動的に作成されます。モード w+ は、MBR が存在しない場合に、更新用に作成されることを指示します。存在している場合は、消去されます。
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    /* Open a text stream file. */
    /* Check to see if it opened successfully */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "w+" ) ) == NULL )
    {
        printf ( "Cannot open MYLIB/TEST(MBR)\n" );
        exit ( 1 );
    }

    printf ( "Opened the file successfully\n" );
    /* Perform some I/O operations. */
    fclose ( fp );
    return 0;
}

図6-2. ILE C テキスト・ストリーム・ファイルを開くための ILE C ソース

注: 処理用に開かれている任意のテキスト・ストリーム・ファイルに対して読み取り、書き込み、または更新を実行できます。

> C++

IBM i ファイルを開くには以下のモードで open() メンバー関数を使用します。
- ios::app
- ios::ate
- ios::in
- ios::out
- ios::trunc

### テキスト・ストリーム・ファイルの書き込み

書き込み操作中に、バッファ内の改行文字があると、テキスト・ストリーム・ファイルに書き込まれるレコードの残りに、プランク文字 (16 進数値 0x40) が埋め込まれます。改行文字自体は破棄されます。

![キャラクターデータシーケンス](characters.png)

図6-3. テキスト・ストリーム・ファイルへの書き込み

バッファ内に書き込まれた文字数がファイルのレコード長を超えた場合、ファイルに書き込まれるデータは切り捨てられ、errno が ETRUNC に設定されます。

例:
以下の例では、テキスト・ストリーム・ファイルに書き込む方法を示します。

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buf[12] = "abcd\nefghi\n";
    FILE *fp;
    /* Open a text file for writing. */
    if ((fp = fopen("MYLIB/TEST(MBR)", "w")) == NULL)
    {
        printf("Cannot open file\n");
        exit(1);
    }
    /* Write characters to the file. */
    fputs(buf, fp);
    /* Close the text file. */
    fclose(fp);
    return 0;
}
```

図6-4. テキスト・ストリーム・ファイルに文字を書き込むための ILE C ソース

### テキスト・ストリーム・ファイルの読み取り

テキスト・ストリーム・ファイルからの読み取り操作中、ファイルからパッファーに読み取られるレコード内のすべての末尾ブランク文字 (16 進数値 0x40) は無視されます。最後の非ブランクの後に改行文字が挿入されます。

<table>
<thead>
<tr>
<th>RRN</th>
<th>ファイル</th>
<th>バッファー</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>characters</td>
<td>a b c d \n e f g h i \n</td>
</tr>
<tr>
<td>1</td>
<td>abcd</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>efghi</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

図6-5. テキスト・ストリーム・ファイルからの読み取り

例:

以下の例では、テキスト・ストリーム・ファイルから読み取る方法を示します。
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  char buf[12];
  char *result;
  FILE *fp;
  /* Open an existing text file for reading. */
  if (! fp = fopen ( "MYLIB/TEST(MBR)", "r" ) ) == NULL )
  {
    printf ( "Cannot open file
" );
    exit ( 1 );
  }
  /* Read characters into the buffer. */
  result = fgets ( buf, sizeof(buf), fp );
  printf("%10s", result);
  result = fgets ( buf+5, sizeof(buf), fp );
  printf("%10s", result);
  fclose ( fp );
  return 0;
}

図6-6. テキスト・ストリーム・ファイルから文字を読み取るための ILE C ソース

テキスト・ストリーム・ファイルの更新
テキスト・ストリーム・ファイルへの更新操作時に、ファイルに書き込まれた文字数がファイルのレコード長を超えた場合、レコード内の末尾の文字は切り捨てられ、errno が ETRUNC に設定されます。

テキスト・ストリーム・ファイルに書き込まれるデータが、更新されるレコード長より短く、書き込まれるデータの最後の文字が改行文字である場合は、レコードが更新され、レコードの残りがブランク文字で埋められます。書き込まれるデータの最後の文字が改行文字でない場合は、レコードは更新され、レコードの残りは変更されません。

バイナリー・ストリーム・ファイルの入出力プロセス
この部では、以下の方法について説明します。
1. 度に 1 文字ずつバイナリー・ストリーム・ファイルを開く
2. 度に 1 文字ずつバイナリー・ストリーム・ファイルに書き込む
3. 度に 1 文字ずつバイナリー・ストリーム・ファイルを読み取る
4. 度に 1 文字ずつバイナリー・ストリーム・ファイルを更新する
5. 度に 1 レコードずつバイナリー・ストリーム・ファイルを開く
6. 度に 1 レコードずつバイナリー・ストリーム・ファイルに書き込む
7. 度に 1 レコードずつバイナリー・ストリーム・ファイルを読み取る

バイナリー・ストリーム・ファイルのオープン (文字単位)
文字単位処理のバイナリー・ストリーム・ファイルとして IBM i ファイルをオープンするには、以下のいずれかのモードで fopen() を使用します。
1. rb
2. r+b
3. rb+
注:
1. fopen() で同時にオープンできるファイルの数は、使用可能なシステム・ストレージのサイズによって異なります。
2. fopen() 関数のオープン・モードは、freopen() 関数にも適用されます。
3. バイナリー・ストリーム・ファイルに削除済みレコードが含まれる場合、バイナリー・ストリーム出力関数はその削除済みレコードをスキップします。

有効なキーワード・パラメーターは次のとおりです。
- blksize
- lrecl
- recfm
- type
- commit
- ccsid
- arrseq
- indicators

type パラメーターを指定する場合、値は、バイナリー・ストリームの文字単位処理のために memory でなければなりません。

注: memory パラメーターは、C プログラムからのみアクセスできるメモリー・ファイルとして、このファイルを識別します。このパラメーターはデフォルトであり、無視されます。

fopen() で無効なモードまたはキーワード・パラメーターを指定した場合、errno に EBADMODE が設定されます。

例:

以下の例では、バイナリー・ストリーム・ファイルをオープンする方法を示しています。
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    /* Open an existing binary file. */
    if ((( fp = fopen ( "MYLIB/TEST(MBR)", "wb+" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    printf ("Opened the file successfully\n");
    /* Perform some I/O operations. */
    fprintf (fp, "Hello, world");
    fclose ( fp );
    return 0;
}

図6-7. バイナリー・ストリーム・ファイルをオープンするための ILE C のソース

注: 文字単位処理用にオープンしたバイナリー・ストリーム・ファイルの読み取り、書き込み、または更新を行えます。

### C++

文字単位処理のバイナリー・ストリーム・ファイルとして IBM i ファイルをオープンするには、ios::binary のほか、以下のいずれかのモードで open() メンバー関数を使用します。

- ios::app
- ios::ate
- ios::in
- ios::out
- ios::trunc

バイナリー・ストリーム・ファイルの書き込み (文字単位)

文字単位で処理されるバイナリー・ストリームにデータを書き込むときに、データのサイズが現行レコード長より大きい場合、その過剰なデータは現行レコードにそのレコード・サイズまで書き込まれ、残りのデータはファイル内の次のレコードに書き込まれます。

---

図6-8. バイナリー・ストリーム・ファイルへの文字単位の書き込み

例:
以下の例では、バイナリー・ストリームに文字単位で書き込む方法の例を示しています。

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[5] = {'a', 'b', 'c', 'd', 'e'};
    /* Open an existing binary file for writing. */
    if ((fp = fopen("MYLIB/TEST(MBR)", "wb")) == NULL)
    {
        printf("Cannot open file\n");
        exit(1);
    }
    /* Write 5 characters from the buffer to the file. */
    fwrite(buf, 1, sizeof(buf), fp);
    fclose(fp);
    return 0;
}
```

図6-9. バイナリー・ストリーム・ファイルに文字を書き込むための ILE C のソース

**バイナリー・ストリーム・ファイルの読み取り (文字単位)**

文字単位で処理されるバイナリー・ストリームからの読み取り操作中に、読み取るデータの長さがファイルのレコード長より大きい場合、ファイル内の次のレコードからデータが読み取られます。

![バッファー](chart.png)

<table>
<thead>
<tr>
<th>RRN</th>
<th>ファイル</th>
<th>レコード</th>
<th>レコード長=3</th>
<th>RRN=相対レコード番号</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>abc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>def</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

図6-10. バイナリー・ストリーム・ファイルからの文字単位の読み取り

例:

以下は、バイナリー・ストリーム・ファイルから文字単位で読み取る方法の例を示しています。
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[6];
    /* Open an existing binary file for reading. */
    if ((fp = fopen("MYLIB/TEST(MBR)", "rb")) == NULL)
    {
        printf("Cannot open file
");
        exit(1);
    }
    /* Read characters from the file to the buffer. */
    fread(buf, 1, sizeof(buf), fp);
    printf("%6s
", buf);
    fclose(fp);
    return 0;
}

図6-11. バイナリー・ストリーム・ファイルから文字を読み取るための ILE C のソース

バイナリー・ストリーム・ファイルの更新 (文字単位)

更新されるデータ量が現行レコード長を超える場合、超過したデータは次のレコードを更新します。現行レコードがファイルの最終レコードである場合は、新規レコードが作成されます。

![レコード長より長いデータを使用したファイル構造](image)

図6-12. レコード長より長いデータを使用したバイナリー・ストリーム・ファイルの更新

例:

以下の例は、レコード長より長いデータを使用したバイナリー・ストリーム・ファイルの更新を示しています。
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[5] = "12345";
    /* Open an existing binary file for updating. */
    if ((fp = fopen("QTEMP/TEST(MBR)", "rb+")) == NULL)
    {
        printf("Cannot open file\n");
        exit(1);
    }
    /* Write 5 characters from the buffer to the file. */
    fwrite(buf, 1, sizeof(buf), fp);
    fclose(fp);
    return 0;
}

図6-13 レコード長より長いデータを使用してバイナリー・ストリーム・ファイルを更新するための ILE C ソース

更新されるデータ量が現行レコード長より短い場合、レコードは部分的に更新され、残りは変更されません。

図6-14 レコード長より短いデータを使用したバイナリー・ストリーム・ファイルの更新

例:

以下の例は、レコード長より短いデータを使用したバイナリー・ストリーム・ファイルの更新を示しています。
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[2] = "12";
    /* Open an existing binary file for updating. */
    if (( fp = fopen ( "QTEMP/TEST(MBR)\", "rb\" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Write 2 characters from the buffer to the file. */
    fwrite ( buf, 1, sizeof(buf), fp );
    fclose ( fp );
}

図6-15. レコード長よりも短いデータを使用してバイナリー・ストリーム・ファイルを更新するための ILE C ソース

バイナリー・ストリーム・ファイルのオープン (レコード単位)

レコード単位の処理のバイナリー・ストリーム・ファイルとして IBM i ファイルをオープンするには、以下のいずれかのモードで fopen() を使用します。

- rb
- r+b
- rb+
- wb
- w+b
- wb+
- ab
- a+b
- ab+

注:
1. fopen() で同時にオープンできるファイルの数は、使用可能なシステム・ストレージのサイズによって異なります。
2. fopen() のオープン・モードは、freopen() にも適用されます。
3. バイナリー・ストリーム・ファイルに削除済みレコードが含まれる場合、バイナリー・ストリーム入出力関数はその削除済みレコードをスキップします。
4. ファイルは、type を record に設定してオープンする必要があります。

有効なキーワード・パラメーターは次のとおりです。

- blksize
- recfm
- commit
- arrseq
- lrecl
・type
・ccsid
・indicator

fopen() 関数で無効なモードまたはキーワード・パラメーターを指定した場合、errno に EBADMODE が設定されます。

レコード単位の処理用にオープンしたバイナリー・ストリーム・ファイルに使用できるのは、fread() と fwrite() のみです。

### C++

レコード単位の処理のバイナリー・ストリーム・ファイルとして IBM i ファイルをオープンするには、ios::binary のほか、以下のいずれかのモードで open() 関数を使用します。

・ios::app
・ios::ate
・ios::in
・ios::out
・ios::trunc

### バイナリー・ストリーム・ファイルの書き込み (レコード単位)

レコード単位で処理されるバイナリー・ストリームにデータを書き込むときに、サイズとカウンター (fwrite のパラメーター) の積がレコード長より大きい場合は、現行レコードに収まるデータだけが書き込まれ、errno が ETRUNC に設定されます。

サイズとカウンターの積が実際のレコード長より小さい場合は、現行レコードにプランク文字が埋め込まれ、errno が EPAD に設定されます。

ファイル |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>abc</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>レコード長=3</td>
</tr>
</tbody>
</table>

パッファー

![図6-16 バイナリー・ストリーム・ファイルへのレコード単位の書き込み](image)

レコード単位の処理用にオープンしたバイナリー・ストリーム・ファイルへの書き込みに有効なのは fwrite() 関数のみです。その他の出力関数および位置決め関数はすべて失敗し、errno が ERECIO に設定されます。

例:

以下の例では、バイナリー・ストリーム・ファイルにレコード単位で書き込む方法の例を示しています。
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[5] = {'a', 'b', 'c', 'd', 'e'};
    /* Open an existing binary file for writing. */
    if ((fp = fopen("MYLIB/TEST(MBR)", "wb,type=record,lrecl=3")) == NULL)
    {
        printf("Cannot open file\n");
        exit(1);
    }
    /* Write 3 characters from the buffer to the file. */
    fwrite(buf, 1, sizeof(buf), fp);
    fclose(fp);
    return 0;
}

図6-17 バイナリー・ストリーム・ファイルにレコード単位で書き込むための ILE C のソース

バイナリー・ストリーム・ファイルの読み取り (レコード単位)

レコード単位で処理されるバイナリー・ストリームからデータを読み取るときに、サイズとカウント (fread() のパラメーター) の積がレコード長より大きい場合は、現行レコード内のデータだけが、バッファーに読み取られます。 fread() 関数は、バッファー内のデータが指定より小さいことを示す値を戻します。

サイズとカウントの積が実際のレコード長より小さい場合は、errno が ETRUNC に設定され、バッファーにコピーされなかったデータがレコードにあることを示します。

次の図は、サイズとカウントの積がレコード長より大きいときに、現行レコードのみをバッファーに読み取る方法の例を示しています。

```
RRN         ファイル          バッファー
            0   1   2
1         abc   a   b   c
2         def   
3         
レコード長=3           RRN=相対レコード番号
```

図6-18 バイナリー・ストリーム・ファイルからのレコード単位の読み取り

レコード単位の処理用にオープンしたバイナリー・ストリーム・ファイルの読み取りに有効なのは fread() 関数のみです。その他の入力関数および位置決め関数はすべて失敗し、errno が ERECIO に設定されます。

例:
以下の例では、バイナリー・ストリームをレコード単位で読み取る方法を示しています。

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[6];
    /* Open an existing binary file for reading a record at a time. */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "rb, type=record" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Read characters from the file to the buffer. */
    fread ( buf, sizeof(buf), 1, fp );
    printf ( "%6s\n", buf );
    fclose ( fp );
    return 0;
}
```

図6-19. バイナリー・ストリーム・ファイルからレコード単位で読み取るための ILE C のソース

オープン・フィードバック域
オープン・フィードバック域は、オープン・データ・バスに関連付けられたオープン・ファイルに関する情報が含まれている、オープン・データ・バスの部分です。_Ropnfbk() 関数を使用して、この情報にポインタを割り当てることができます。オープン・フィードバック域にマップされる構造体は、<xxfdbk.h> ヘッダー・ファイルにあるものにすることが可能です。

入出力フィードバック域
入出力フィードバック域は、成功した各非ブロック入出力操作後に更新されるファイルのオープン・データ・パスの一部です。レコード・ブロッキングが行われている場合、入出力フィードバックは、各レコード・ブロックがプログラムとシステム間で転送された後に更新されます。

入出力フィードバックは、2 つの部分 (すべてのファイル・タイプに共通の一部とファイルのタイプに固有の部分) で構成されています。

入出力フィードバック域の共通部分へのポインタを割り当てるには、_Riofbk() 関数を使用します。ファイルのタイプに固有な入出力フィードバック域の部分へのポインタを割り当てるには、共通部分へのポインタに、共通部分の file_dep_fb_offset フィールドに含まれているオフセットを追加します。

注: オフセットはバイト単位であるため、ポインタ演算の実行時に、共通部分へのポインタ (char *) を、文字へのポインタにキャストする必要があります。入出力フィードバック域にマップされる構造体は、<xxfdbk.h> ヘッダー・ファイルに含まれている構造体です。

セッション管理機能の使用
情報を画面に出力する ILE C ストリーム入出力関数は、Dynamic Screen Manager (DSM) のセッション管理機能 API を使用して定義されます。
セッション・ハンドルの入手

C/C++ セッション用のセッション・ハンドルを入手し、DSM API を使用してそのセッションを処理することができます。セッション・ハンドルは、<stdio.h> 内の _C_Get_Ssn_Handle() を通じて提供されます。

以下の例に示すように、DSM QsnClrScl API を使用して、C セッションを消去する単純な C プログラムを作成できます。

```c
#include <stdio.h>
#include "qsnapi.h"
void main (void)
{
  QsnClrScl(_C_Get_Ssn_Handle(), '0', NULL);
}
```

図6-20. C セッションを消去する単純な C プログラム

セッション・マネージャー API の使用

セッション・ハンドルで有効な操作を実行するには、DSM API を使用できます。その中には、ウィンドウ・サービス API や多くの下位レベル・サービスが含まれます。

以下に例を示します。

• QsnStrWin、QsnDspSsnBot、および QsnReadSsnDta の API の組み合わせを使用してセッションを表示することができますが、この場合には getc() を含むプログラムを作成するだけのほうが簡単です。

• QsnRtvWinD と QsnChgWin の API を使用すると、C/C++ セッションをデフォルトのフルスクリーン・ウィンドウからより小さいウィンドウに変更することができます。

例：ILE バインド可能 API を使用した DSM セッションの表示

以下の例は、Dynamic Screen Manager (DSM) ILE バインド可能 API を呼び出して DSM セッションを表示する方法を示しています。この DSM セッションでは、DSM セッション中に入力したデータがエコー・バックされます。

手順:
1. 6-22 ページの『コード・サンプル』に示したソースを使用してモジュール T1520API を作成するには、以下を入力します。
   ```c
   CRTCMOD MODULE(MYLIB/T1520API) SRCFILE(QCPPLE/QACSRC) OUTPUT(*PRINT)
   ```
2. プログラム T1520API を作成するには、以下を入力します。
   ```c
   CRTPGM PGM(MYLIB/T1520API) MODULE(MYLIB/T1520API) BNDDIR(QSNAPI)
   ```
   CRTPGM コマンドは、ライブラリー MYLIB 内にプログラム T1520API を作成します。
3. プログラム T1520API を実行するには、以下を入力します。
   ```c
   CALL PGM(MYLIB/T1520API)
   ```
   出力は以下のようになります。

入出力操作の実行 6-21
/* This program uses Dynamic Screen Manager API calls to */
/* create a window and echo whatever is entered. This is an */
/* example of bound API calls. Note the use of #pragma argument */
/* in the <qsnsess.h> header file. OS, nowiden ensures that a pointer */
/* to an unwidened copy of the argument is passed to the API. */
/* */
/* Use BNDDIR(QSNAPI) on the CRTPGM command to build this */
/* example. */
#define BOTLINE " Echo lines until: PF3 - exit"

#include <stddef.h>
#include <string.h>
#include <stdio.h>
#include "QSYSINC/H/QSNAPI"

/* QSNSESS nests QSNWIN and QSNLL include files. To get these 3 */
/* include files, do the following: */
/* 1) If you do not have a SRCPF called H in your Library (MYLIB), */
/* create one. */
/* 2) Copy QUSRTOOL/QATTSYSC/QPSN3API to MYLIB/H/QSNSESS */
/* 3) Copy QUSRTOOL/QATTSYSC/QPSN2API to MYLIB/H/QSNWIN */
/* 4) Copy QUSRTOOL/QATTSYSC/QPSN1API to MYLIB/H/QSNLL */

#define BOTLINE " Echo lines until: PF3 - exit"

/* DSM Session Descriptor Structure. */

typedef struct{
    Qsn_Ssn_Desc_T sess_desc;
    char buffer[300];
}storage_t;

void F3Exit(const Qsn_Ssn_T *Ssn, const Qsn_Inp_Buf_T *Buf, char *action)
{
    *action = '1';
}

int main(void)
{
    int i;
    storage_t storage;

    /* Declarators for declaring windows. Types are from the <qsnsess.h> */
    /* header file. */
```c
Qsn_Inp_Buf_T    input_buffer = 0;
Q Bin4           input_buffer_size = 50;
char             char_buffer[100];
Q Bin4           char_buffer_size;

Qsn_Ssn_T        session1;
Qsn_Ssn_Desc_T   *sess_desc = (Qsn_Ssn_Desc_T *) &storage;
Qsn_Win_Desc_T   win_desc;
Q Bin4           win_desc_length = sizeof(Qsn_Win_Desc_T);
char             *botline = BOTLINE;
Q Bin4           botline_len = sizeof(BOTLINE) - 1;
Q Bin4           sess_desc_length = sizeof(Qsn_Ssn_Desc_T) +
                     botline_len;
Q Bin4           bytes_read;

/* Initialize Session Descriptor API. */
QsnInzSsnD( sess_desc, sess_desc_length, NULL);

/* Initialize Window Descriptor API. */
QsnInzWinD( &win_desc, win_desc_length, NULL);

sess_desc->cmd_key_desc_line_1_offset = sizeof(Qsn_Ssn_Desc_T);
sess_desc->cmd_key_desc_line_1_len = botline_len;
memcpy( storage.buffer, botline, botline_len );
(sess_desc->cmd_key_desc_line_2_offset = sizeof(Qsn_Ssn_Desc_T) +
botline_len);

/* Set up the session type. */
sess_desc->EBCDIC_dsp_cc = '1';
sess_desc->scl_line_dsp = '1';
sess_desc->num_input_line_rows = 1;
sess_desc->wrap = '1';

/* Set up the window size. */
win_desc.top_row    = 3;
win_desc.left_col   = 3;
win_desc.num_rows   = 13;
win_desc.num_cols   = 45;

/* Create a window session. */
sess_desc->cmd_key_action[2] = F3Exit;
session1 = QsnCrtSsn( sess_desc, sess_desc_length,
                       NULL, 0,
                       '1',
                       &win_desc, win_desc_length,
                       NULL, 0,
                       NULL, NULL);

if(input_buffer == 0)
{
    input_buffer = QsnCrtInpBuf(100, 50, 0, NULL, NULL);
}
for (;;)
{
    /* Echo lines until F3 is pressed. */

    QsnReadSsnDta(session1, input_buffer, NULL, NULL);
    if (QsnRtvReadAID(input_buffer, NULL, NULL) == QSN_F3) {
    
    }

```

注:
1. DSM API のプロトタイプは <qnsess.h> ヘッダー・ファイルにあります。
2. 各 API に #pragma argument (API, OS, nowiden) ディレクティブが指定されています。これにより、
値引数が値によって間接的に渡されるようになります。

IBM i 統合ファイル・システムでの ILE C/C++ ストリーム関数の使用
このセクションでは、IBM i 統合ファイル・システム (IFS) を通じてテキスト・ストリーム・ファイルおよ
びバイナリー・ストリーム・ファイルのオープン、書き込み、読み取り、更新を行う方法について説明し
ます。
IFS では、ストリーム・ファイルの情報を保管および操作するための共通インターフェースが提供されてい
ます。ストリーム・ファイルの例として、PC ファイル・システム、UNIX のファイル・システム、LAN
サーバー・ファイル・システム、およびフォルダがあります。
注: ILE C/C++ の IFS 対応のストリーム入出力関数は、統合ファイル・システムを通じて定義されます。
ILE C/C++ ストリーム入出力関数を使用するには、統合ファイル・システムを熟知する必要があります。
統合ファイル・システムは、7 つのファイル・システムによって構成されます。アプリケーションと環境に
応じて、このファイル・システムのいくつかを使用できます。IBM i ファイルを使用する既存アプリケー
ションがある場合は、QSYS.LIB ファイル・システムの制限を把握する必要があります。新しいアプリケー
ションがある場合は、QSYS.LIB ファイル処理の制約事項がない、その他のファイル・システムを使用で
きます。各ファイル・システムについては、「統合ファイル・システム (IFS)」のセクションを参照して
ください。

このトピックでは以下について説明します。
• 統合ファイル・システム (IFS) のコンポーネント
• IFS ストリーム入出力を使用可能にする方法
  • ストリーム・ファイル、テキスト・ストリーム、およびバイナリー・ストリーム
  • テキスト・ストリーム・ファイルおよびバイナリー・ストリーム・ファイルをオープンする方法
  • テキスト・ストリームまたはバイナリー・ストリームとしてデータを保管する方法
  • IFS ファイルの使用に関する便利な情報

統合ファイル・システム (IFS)
ファイル・システム は、論理単位として編成されているストレージの特定のセグメントにアプリケーショ
ンがアクセスできるようにするサポートを提供します。論理単位には、ファイル、ディレクトリー、ライブラリー、およびオブジェクトがあります。

統合ファイル・システムには、以下の 7 つのファイル・システムがあります。
• ルート (/)
• オープン・システム (QOpenSys)
• ライブラリー (QSYS.LIB)
• 文書ライブラリー・サービス (QDLS)
・LAN Server/400 (QLANsvr)
・光ディスク・サポート (QOPT)
・ファイル・サーバー (QFileSvr.400)

図 6-21 に、これらのファイル・システムを示します。

ユーザーおよびアプリケーション・プログラムは、共通統合ファイル・システム (IFS) インターフェースを使用して、任意のファイル・システムと対話できます。このインターフェースは、各種インターフェースを使用して提供されるレコード入出力とは異なり、ストリーム・データの入出力用に最適化されています。共通統合ファイル・システム・インターフェースには、一連のユーザー・インターフェース (コマンド、メニュー、および画面) およびアプリケーション・プログラム・インターフェース (API) が含まれています。

図 6-21. 統合ファイル・システム・インターフェース

ルート (/) ファイル・システム
ルート (/) ファイル・システムは、統合ファイル・システムのストリーム・ファイル・サポートおよび階層ディレクトリ構造をフル活用するように設計されています。これには、DOS および OS/2 のファイル・システムの特性があります。

ユーザー・アクセス: ルート (/) ファイル・システムは、統合ファイル・システム・インターフェースを介してのみアクセスできます。ルート (/) ファイル・システムを処理するには、統合ファイル・システムのコマンド、ユーザー画面、または ISO ストリーム入出力関数とシステム API を使用します。

パス名: このファイル・システムは、オブジェクト名が入力された時の大文字/小文字形式を保ちますが、システムが名前を検索する際には、大文字/小文字の区別はありません。
・パス名の形式は、以下のとおりです。
Directory/Directory/.../Object
• 부스명의 각 컴포넌트는, 최대 255 문자의 길이가 하이 로그 기록이 가능합니다. 부스는 최대 16 메가바이트에 달할 수 있습니다.
• 프로그램 및 스피어스의 제한은, 디렉토리 레벨의 깊이는 제한이 없습니다.
• 명령이 보유할 수 있는 하이 로그에서는, 명령의 하이 로그 세트 2 (UCS2) 레벨 1 형식으로 변환됩니다.

**오픈-시스템 (QOpenSys) 파일 시스템**
오픈-시스템 (QOpenSys) 파일 시스템은, POSIX 및 XPG 등의 UNIX 베이스의 오픈-시스템 표준과 호환을 위한 구현이 있습니다. 각 파일은 고유한 이름을 가지며, 시스템은 각 파일에 대해 독립적으로 관리합니다. 파일들은 시스템의 메모리 영역에 저장되며, 파일은 이메시지 기반의 방법으로 접근할 수 있습니다.

유저 액세스: QOpenSys는, 통합 파일 시스템과 인터페이스를 통해 유저 액세스가 가능합니다. QOpenSys는, 파일 시스템의 사용자 명령어, 유저 상호 작용, 또는 ISO 스트림 I/O를 이용한 시스템 API를 사용합니다.

육수: QSYS.LIB, QDLS, QLANSrv, 이에 해당하는 루트 (t) 파일 시스템과 다른 루트 (t) 파일 시스템은, 같은 이름의 파일을 차지할 수 없습니다. QOpenSys은, 오브젝트의 이름을 검색하여 대소문자 사양을 보유할 수 있습니다.

육수, 링크, 보조, 스크립트, 그리고 ISO 스트림 I/O의 파일 시스템은, 파일의 이름에 따라, 시스템 API를 사용하여 각 루트 (t) 파일 시스템에서 사용할 수 있습니다.

**라이브러리 (QSYS.LIB) 파일 시스템**
라이브러리 (QSYS.LIB) 파일 시스템은, IBM i 라이브러리의 기능을 지원합니다. 데이터베이스의 파일, 라이브러리의 보조, 관리, 그리고 기타 IBM i 오브젝트의 타입에 따른 액세스가 가능합니다.

QSYS.LIB 파일 시스템은, IBM i 파일 시스템에 맵핑됩니다. 예를 들어, /qsys.lib/qsysinc.lib/h.file/stdio.mbr는, 루트 라이브러리 QSYS 내의 라이브러리 QSYSINC의 파일 H에 있는 파일 및 멤버 STUDIO를 참조합니다.

필수 처리의 제약 사항: 통합 파일 시스템의 사용 시에는, 몇 가지 제약 사항이 있습니다.
• 두드러지 파일은 지원되지 않습니다.
• 사용 가능한 묘지 파일의 타입은, 단일 파일을 저장하거나 분산 파일을 저장한 파일 및 단일 파일을 저장한 파일입니다.
• 파일 범위는 지원되지 않습니다.
• 데이터베이스의 파일 및 멤버가 포함된 시스템의 경우, 흔히 사용하는 시스템의 경우, 파일의 수의 갱신을 수행해야 합니다. 파일의 갱신을 수행할 수 있는 파일의 수는 1 이지만, 다른 파일은 파일과 파일의 갱신을 수행할 수 없습니다. 이상의 파일은 각 파일을 갱신할 수 있습니다.

육수: QSYS.LIB 파일 시스템에서는, 오브젝트의 이름 대문자와 소문자를 사용하는 제한이 없습니다. 이름의 대문자와 소문자를 사용할 수 있습니다. 결과는 동일한 경우입니다.

단, 명령이 이용 가능한 경우, 명령의 이름과 대문자와 소문자로 유지가 가능합니다. 명령이 이용 가능한 경우, 명령의 이름과 대문자와 소문자로 유지가 가능합니다. 명령이 이용 가능한 경우, 명령의 이름과 대문자와 소문자로 유지가 가능합니다.

육수의 모든 컴포넌트와 오브젝트의 이름과 그 후에 오브젝트의 타입이 유지되지 않는 경우, 다음과 같은 예시를 보여줍니다.
/QSYS.LIB/QGPL.LIB/PRT1.OUTQ
/QSYS.LIB/PAYROLL.LIB/PAY.FILE/TAX.MBR
オブジェクト名とオブジェクト・タイプは、ビリオド (．) で区切ります。オブジェクト・タイプが異なっていれば、1つのライブラリーに同じ名前の複数のオブジェクトを入れることができます。したがって、固有のオブジェクトを識別するためには、必ずオブジェクト・タイプを指定する必要があります。

各コンポーネントのオブジェクト名は 10 文字まで、オブジェクト・タイプは 6 文字までの長さにすることができます。

QSYS.LIB 内のディレクトリ階層は、アクセスされるオブジェクトのタイプによって、2 レベルまたは 3 レベルの深さ（パス名のコンポーネントが 2 つまたは 3 つ）のいずれかになります。オブジェクトがデータベース・ファイルの場合、階層は 3 レベル（ライブラリー、ファイル、メンバー）になります。それ以外の場合には、2 レベル（ライブラリー、オブジェクト）のみになります。各コンポーネント名の長さとディレクトリのレベル数の組み合わせによって、パス名の最大長が決まります。

ルート (/) および QSYS.LIB が最初の 2 レベルとして含まれている場合、QSYS.LIB のディレクトリ階層は、4 または 5 レベルの深さにすることができます。

名前が保管されるときには、名前の文字がコード・ページ 37 に変換されます。引用符付き名前は、ジョブのコード・ページを使用して保管されます。

文書ライブラリー・サービス (QDLS) ファイル・システム
文書ライブラリー・サービス (QDLS) ファイル・システムでは、フォルダー・オブジェクトがサポートされます。文書とフォルダーへのアクセスを提供します。

ユーザー・アクセス: 統合ファイル・システム・インターフェースを通じて QDLS ファイル・システムを処理するには、統合ファイル・システムのコマンド、ユーザー画面、または ISO ストリーム出入力関数とシステム API を使用します。

QDLS 内のオブジェクトを処理するすべてのユーザーは、システム配布ディレクトリに登録されている必要があります。

パス名: QDLS は、英字 a から z のみを含める名前の大文字と小文字を区別しません。大文字と小文字のどちらでオブジェクト名を検索しても、結果は同じです。

その他の文字は、大/小文字が区別され、そのまま使用されます。

パス名の各コンポーネントは、以下のように、1つの名前だけで構成されます。
/QDLS/FLR1/DOC1
あるいは以下のように名前に拡張子がつっています。
/QDLS/FLR1/DOC1.TXT

各コンポーネントの名前は 8 文字まで、拡張子は 3 文字までの長さにすることができます。パス名の最大長は 82 文字です。

/QDLS/ の下のディレクトリ階層は 32 レベルの深さにすることができます。

名前が保管されるときには、名前の文字がコード・ページ 500 に変換されます。コード・ページ 500 に変換できない名前は拒否されます。
LAN Server/400 (QLANSrv) ファイル・システム

LAN Server/400 (QLANSrv) ファイル・システムは、LAN Server/400 ライセンス・プログラムを使用してアクセスされるのと同じディレクトリおよびファイルへのアクセスを提供します。これにより、IBM i ファイル・サーバーおよび IBM i アプリケーションのユーザーが、LAN Server/400 クライアントと同じデータを使用できるようになります。

QLANSrv ファイル・システム内のファイルおよびディレクトリは、OS/2 LAN サーバーに基づいた LAN サーバーによって保管および管理されます。この LAN サーバーでは、ファイルまたはディレクトリの所有者および所有グループの概念はサポートされません。ファイル所有権は、コマンドを使用しても、ISO ストリーム入出力関数とシステム API を使用しても変更することはできません。アクセスは、アクセス制御リストを使用して制御されます。このリストは、WRKAUT および CHGAUT コマンドを使用して変更できます。

ユーザー・アクセス: 統合ファイル・システム・インターフェースを通じて QLANSrv ファイル・システムを処理するには、統合ファイル・システムのコマンド、ユーザー画面、または ISO ストリーム入出力関数とシステム API を使用します。

パス名: ファイル・システムでは、オブジェクト名が入力されたときの大文字と小文字の形式を保持します。システムが名前を検索する際には、大文字/小文字の区別はありません。大文字と小文字のどちらでオブジェクト名を検索しても、結果は同じです。

パス名の形式は、以下のとおりです。

Directory/Directory/.../Object

パス名の各コンポーネントは、最大 255 文字の長さにすることができます。

QLANSrv 内のディレクトリ階層は、127 レベルの深さにすることができます。パスのすべてのコンポーネントが階層レベルとして含まれている場合、ディレクトリ階層は、132 レベルの深さにすることもできます。

名前前は、ファイル・サーバーに定義されているコード・ページで保管されます。

光ディスク・サポート (QOPT) ファイル・システム

光ディスク・サポート (QOPT) ファイル・システムには、統合ファイル・システム・インターフェースを使用してアクセスできます。これを行うには、IBM i ファイル・サーバーを使用するか、統合ファイル・システムのコマンド、ユーザー画面、および ISO ストリーム入出力関数とシステム API を使用します。

パス名: QOPT は、オブジェクト名で使用されるときに小文字の英字 a から z を大文字に変換します。

したがって、こうした文字のみを使用しているオブジェクト名の検索では、大/小文字の区別はありません。

パス名は、スラッシュ (/) で始まり、294 文字以内でなければなりません。パスは、ファイル・システム名、ボリューム名、ディレクトリ名とサブディレクトリ名、およびファイル名で構成されます。以下に例を示します。

/QOPT/VOLUMENAME/DIRECTORYNAME/SUBDIRECTORYNAME/Filename

ファイル・システム名の QOPT は必須です。

ボリューム名は必須であり、32 文字までの長さにすることが可能です。

パス名には 1 つ以上のディレクトリまたはサブディレクトリを含めることができますが、いずれも必須ではありません。すべてのディレクトリ名およびサブディレクトリ名の合計文字数は、先行スラッシュを含め、63 文字を超えることはできません。ディレクトリ名とファイル名には、0x00 から 0x3F、0xFF、0x80、小文字の英字、および以下の文字を除くすべての文字を使用できます。

- アスタリスク (*)
ファイル・サーバー (QFileSvr.400) ファイル・システム

ファイル・サーバー (QFileSvr.400) ファイル・システムには、統合ファイル・システム (IFS) インターフェースを使用してアクセスできます。これを行うには、IBM i ファイル・サーバーを使用するか、統合ファイル・システム・コマンド、ユーザー画面、および ISO ファイルストリーム入出力関数とシステム API を使用します。

注: QFileSvr.400 ファイル・システムの特性は、ターゲット・システム上のアクセス対象のファイル・システムの特性によって決まります。

パス名: 実際にはターゲット・システムのルート (/) ディレクトリを表す第 1 レベル・ディレクトリーの場合、QFileSvr.400 ファイル・システムでは、オブジェクト名が入力されたときの大文字と小文字の形式を保持します。ただし、QFileSvr.400 が名前を検索するときには、大文字と小文字を区別しません。

その他のすべてのディレクトリーの場合、大文字と小文字の区別はアクセス対象の特定のファイル・システムによって異なります。QFileSvr.400 では、ファイル要求が IBM i ファイル・サーバーに送られるときに、オブジェクト名が入力された時点の大文字小文字を保持します。

パス名の形式は、以下のとおりです。

/QFileSvr.400/RemoteLocationName/Directory/Directory.../Object

第 1 レベル・ディレクトリー (すなわち上記の例では RemoteLocationName) は、以下の両方を表します。

- 通信接続を確立するために使用されるターゲット・システムの名前。ターゲット・システム名は、以下のいずれかになることができます。
  - TCP/IP ホスト名 (例えば、beowulf.newyork.corp.com)
  - SNA LU 6.2 名 (例えば、appn.newyork)
- ターゲット・システムのルート (/) ディレクトリー

このため、統合ファイル・システム・インターフェースを使用して第 1 レベル・ディレクトリーが作成されるときには、指定された属性はすべて無視されます。

注: 第 1 レベル・ディレクトリーは、初期プログラム・ロード (IPL) 間で持続しません。つまり、IPL を実行するたびに第 1 レベル・ディレクトリーを作成し直さなければなりません。

パス名の各コンポーネントは、最大 255 文字の長さにすることができます。絶対パス名は、最大 16 メガバイトの長さにすることができます。
注: オブジェクトが常駐するファイル・システムによっては、コンポーネントの長さとバス名の長さが、QFileSvr.400 で認められる最大長より短く制限されることがあります。

ディレクトリ階層の深さには、プログラムの制限、システムの制限、およびアクセス対象のファイル・システムの制限以外に制限はありません。

名前に使用される文字は、名前が保管されるときに UCS2 レベル 1 形式に変換されます。

統合ファイル・システム・ストリーム入出力の有効化
モジュール作成コマンドまたはバイナリプログラムの作成コマンドのブロンドでシステムインターフェース・キーワード (SYSIFCOPT) に *IFSIO オプションを指定することで、サイズが最大で 2 ギガバイトのファイル用の ILE C/C++ ストリーム入出力を有効にすることができます。以下に例を示します。

CRTCMD MODULE(QTEMP/IFSIO) SRCFILE(QCPPLE/QACSRC) SYSIFCOPT(*IFSIO)
CRTCMD MODULE(QTEMP/IFSIO) SRCFILE(QCPPLE/QACSRC) SYSIFCOPT(*IFSIO)

ラージ・ファイルでのストリーム入出力の使用
統合ファイル・システム・インターフェースの 64 ビット・バージョンでは、2 ギガバイトを超えるサイズのファイルで ILE C/C++ ストリーム入出力を使用することができます。以下にリストした方法のいずれかを使用すると、このインターフェースを有効にすることができます。

モジュール作成またはバイナリプログラム作成のコマンド・ブロンドで SYSIFCOPT キーワードに *IFS64IO オプションを指定すること。コンバイラは _IFS64_IO__ マクロを定義します。このマクロにより、IBM 提供のヘッダー・ファイル内に _LARGE_FILES マクロおよび _LARGE_FILE_API マクロが定義されます。以下に例を示します。

CRTCMD MODULE(QTEMP/IFSIO) SRCFILE(QCPPLE/QACSRC) SYSIFCOPT(*IFS64IO)

プログラム・ソースに _LARGE_FILES マクロを定義する。あるいはモジュール作成またはバイナリプログラム作成のコマンド行に DEFINE(_LARGE_FILES) を指定します。統合ファイル・システム API および関連のデータ・タイプが 64 ビットの統合ファイル・システムの対応する部分に自動的にマップまたは再定義されます。以下に例を示します。

CRTCMD MODULE(QTEMP/IFSIO) SRCFILE(QCPPLE/QACSRC)
SYSIFCOPT(*IFSIO) DEFINE('_LARGE_FILES')

プログラム・ソースに _LARGE_FILE_API マクロを定義する。あるいはモジュール作成またはバイナリプログラム作成のコマンド行に DEFINE(_LARGE_FILE_API) を定義します。これにより、64 ビット統合ファイル・システムの API およびその対応するデータ・タイプが可視になりますが、使用する統合ファイル・システム API (通常または 64 ビット) をアプリケーションが明示的に指定する必要があります。以下に例を示します。

CRTCMD MODULE(QTEMP/IFSIO) SRCFILE(QCPPLE/QACSRC)
SYSIFCOPT(*IFSIO) DEFINE('_LARGE_FILE_API')

注: __IFS64_IO__、_LARGE_FILES、および _LARGE_FILE_API の各マクロは、相互に排他的ではありません。例えば、コマンド行に SYSIFCOPT(*IFS64IO) を指定し、プログラム・ソースに _LARGE_FILES と _LARGE_FILE_API のいずれかまたは両方のマクロを定義できます。

ストリーム・ファイル
ILE C/C++ コンバイラーでは、真のテキストまたはバイナリ・ストリーム・ファイル (統合ファイル・システム対応ストリーム入出力を使用) として、あるいはシミュレートされたテキストおよびバイナリ・ストリーム・ファイル (デフォルト・ストリーム入出力を使用) としてプログラムがストリーム・ファイルを処理できます。
ストリーム・ファイルを使用するアプリケーションを書き込む場合、パフォーマンスを改善するために、レコード入出力に基づいてマップされるデフォルトCストリーム入出力の代わりに、統合ファイル・システムを使用することをお勧めします。

**ストリーム・ファイルとデータベース・ファイル**

ストリーム・ファイルとIBM iデータベース・ファイルを比較すると、ストリーム・ファイルをよりよく理解できます。

統合ファイル・システムでは、ストリームは単なる連続した文字ストリーミングです。データベース・ファイルはレコード単位で編成されていて、長さやデータ・タイプなどの特定の性質を持つ1つ以上のフィールドに事前に分割されています。

![ストリーム・ファイルとデータベース・ファイルの図](image)

図6-22. ストリーム・ファイルとレコード単位ファイルとの比較

IBM iでのデフォルトのC/C++ストリーム入出力は、IBM iデータベース・ファイルに重ねてシミュレートされます。[図6-23]は、IBM iレコードがどのようにC/C++ストリームにマップされるかを示しています。これは、IBM iレコードで処理するシミュレートされたストリーム・ファイルです。

![IBM iレコードのC/C++ストリーム・ファイルへのマッピング](image)

図6-23. IBM iレコードのC/C++ストリーム・ファイルへのマッピング

ストリーム・ファイルおよびレコード単位ファイルの構造における相違点は、その対話用にアプリケーションをどのように書くか、またアプリケーションにどのタイプのファイルが最適であるかに影響します。

・レコード単位ファイルは、名前、住所、勘定残高などの顧客情報を保管するのに適しています。これらフィールドは、IBM iの拡張プログラミング関数を使用すると、個別にアクセスおよび処理することができます。
・ストリーム・ファイル は、さまざまな色を表す一連のビット・ストリングで構成された顧客の描写などの情報を持ちます。ストリーム・ファイルは、文書のテキスト、イメージ、音声、およびビデオなどのデータのストリングをもつのが適しています。

テキスト・ストリーム
テキスト・ストリーム には印刷可能な文字と制御文字が含まれ、これらの文字が用いられる構文そのものが構成されている。各行は、ゼロ個以上の文字で構成され、改行文字（
m）で終了します。改行文字は、ファイルの終わりに自動的に追加されます。

ILE C/C++ 実行環境は、IBM i 環境でテキストを表すための規則に準拠するために、入力時に一部の特殊文字を追加、変更、または無視することがあります。そのため、ファイルに書き込まれた文字と、同じファイルから再読取られた文字との間に、1 対 1 の対応がないことがあります。

統合ファイル・システム・テキスト・ストリームからの読み取られたデータは、テキストが印刷可能文字、水平タブ、改行、垂直タブ、および用紙送り制御文字のみで構成されている場合、書き込まれたデータと同じです。

ほとんどの統合ファイル・システム・ストリーム・ファイルでは、行は、ゼロ個以上の文字で構成され、復帰と改行文字の組み合わせで終了します。ただし、統合ファイル・システムには、単一の改行（LF）を行終了文字として使用することができる別のシステム上のファイルへの論理リンクが含まれていることがあります。これの良い例として、ほとんどの UNIX システムでのファイルが挙げられます。

入力時には、テキスト・モードのデフォルトでは、改行と復帰文字の組み合わせの行区切り文字からすべての復帰がストリップされます。出力時には、各改行文字が、復帰文字とそれに続く改行文字に変換されます。行終了文字シーケンスは、fopen() の CRLN オプションで変更できます。

注: *IFSIO オプションでも、「¥」エスケープ文字値が 0x25 改行文字に変更されます。*NOIFSIO が指定されている場合、「¥」エスケープ文字値は 0x15 となります。

ファイルがテキスト・モードで開かれると、そのファイルとの間で処理されるデータに対してコード・ページ変換が行われることがあります。データは、ファイルか読み取られる際に、ファイルのコード・ページから、データを受け取るアプリケーション、ジョブ、またはシステムのコード・ページに変換されます。

データは、IBM i ファイルに書き込まれる際に、アプリケーション、ジョブ、またはシステムのコード・ページから、ファイルのコード・ページに変換されます。真のストリーム・ファイルの場合には、行フォーマット設定文字（復帰、タブ、ファイルの終わりなど）がすべて、1 つのコード・ページから別のコード・ページに変換されます。

QSYS.LIB ファイルからの読み取り時に、行の終わりの文字（復帰および改行）が、パッファーで返されるデータの末尾に追加されます。

テキスト・ファイルの処理時に実行されるコード・ページ変換は、fopen() でコード・ページまたは CCSID オプションを指定することで変換できます。デフォルトでは、ファイルから読み取られるすべてのデータが、ジョブの CCSID またはコード・ページに変換されます。

バイナリ・ストリーム
バイナリ・ストリーム は、関連する IBM i ファイルで保存された文字との 1 対 1 対応を持つ文字のシーケンスです。データは入出力時に変更されないため、バイナリ・ストリームから読み取られるデータは、書き込まれたデータと同じです。バイナリ・ストリームでは、改行文字は特別な意味を持ちません。

6-32 IBM i: ILE C/C++ プログラマーの手引き
テキスト・ストリーム・ファイルおよびバイナリー・ストリーム・ファイル
のオープン

各テキスト・ストリーム・ファイルおよび各バイナリー・ストリーム・ファイルは、タイプ file のファイル制御構造で表されます。この構造は、ファイルをオープンした際のモードに応じて初期化され、ファイル制御構造を変更しようとするとき、予期しない結果が発生する可能性があります。

fopen() の形式は次のとおりです。
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);

モード 変数は、オープン・モードと、場合によってその後に続くキーワード・パラメーターで構成される文字ストリングです。オープン・モードとキーワード・パラメーターは、1つコンマまたは1つ以上のプランク文字で区切る必要があります。

テキスト・ストリーム・ファイルをオープンするには、以下のいずれかのモードで fopen() を使用します。
- r または r+
- w または w+
- a または a+

バイナリー・ストリーム・ファイルをオープンするには、以下のいずれかのモードで fopen() を使用します。
- rb、
- rb+、または r+b
- wb、
- wb+ または w+b
- ab、
- awb+ または a+b

バイナリー・ストリームファイルをオープンするには、ios::binary または以下のいずれかのモードで open() メンバー関数を使用します。
- ios::app
- ios::ate
- ios::in
- ios::out
- ios::trunc

テキスト・ストリームまたはバイナリー・ストリームとしてのデータの保管

2つのストリームをそれぞれバイナリー・ストリームとテキスト・ストリームとしてオープンして、両方に同じ情報を書き込んだ場合に、ストリームの内容が異なる可能性があります。以下では、異なるタイプの2つのストリームを示しています。結果ファイル（データが実際にどのように保管されたかを示す）の16進
/* Use CRTBNDC SYSIFCOPT(*IFSIO) */
#include <stdio.h>
int main(void)
{
    FILE *fp1, *fp2;
    char lineBin[15], lineTxt[15];
    int x;
    fp1 = fopen("script.bin","wb");
    fprintf(fp1,"hello world\n");
    fp2 = fopen("script.txt","w");
    fprintf(fp2,"hello world\n");
    fclose(fp1);
    fclose(fp2);
    fp1 = fopen("script.bin","rb");
    fp2 = fopen("script.txt","rb");
    /* opening the text file as binary to suppress */
    /* the conversion of internal data */
    fgets(lineBin, 15, fp1);
    fgets(lineTxt, 15, fp2);
    printf("Hex value of binary file = ");
    for (x=0; lineBin[x]; x++)
        printf("%.2x", (int)(lineBin[x]));
    printf("Hex value of text file = ");
    for (x=0; lineTxt[x]; x++)
        printf("%.2x", (int)(lineTxt[x]));
    fclose(fp1);
    fclose(fp2);
    /* The expected output is: */
    /* Hex value of binary file = 888593939640a69699938425 */
    /* Hex value of text file = 888593939640a6969993840d25 */
}

図6-24 テキスト・ストリームとバイナリ・ストリームの比較

ファイル内容の16進数値がバイナリ・ストリーム(script.bin)で示されるとき、改行文字はline-feedの16進数値(0x25)に変換されます。一方、テキスト・ストリーム(script.txt)で、改行はcarriage-return line-feedの16進数値(0x0d25)に変換されます。

統合ファイル・システム (IFS) の使用

ILE C/C++は主として、IBM iルート・ファイル・システムをサポートします。ルート・ファイル・システムは、統合ファイル・システム・インターフェースでアクセス可能な多くのファイル・システムの1つです。これは、UNIXシステム上のファイルおよびディレクトリへのアクセスに使用される表記と類似した表記を使用するため、ユーザーは、複数のプラットフォームで統一された方法で情報にアクセスすることができます。

各種プラットフォームとの間でファイル転送を行う場合には注意してください。FTPのようなダウンロードおよびアップロードユーティリティーを使用すると、ストリームのソースが一時的に他のプラットフォームに保管されていた場合でもIBM iプラットフォームで引き続き有効になるように、文字の正しいマッピングを指定することができます。詳しいヒントについては、6-43ページの『回避すべき危険性』を参照してください。
ソース・ファイルの IFS へのコピー

メインのソース物理ファイルを統合ファイル・システム (IFS) のファイルにコピーすることができます。ソース物理ファイルに標準の名前を使用したと仮定すると、次のコマンドを使用します。
CPYTOSTMF FROMMBR('/QSYS.LIB/MYLIB.LIB/QCSRC.FILE/QCSRC.MBR') TOSTMF('/home/qcsrc.c')

ストリーム・ファイルの編集

ファイル編集 (EDTF) コマンドを使用して、ストリーム・ファイルを直接編集することができます。また、ストリーム・ファイルと共に使用するファイルを編集する方法には、以下の 3 つがあります。
- クライアント・アクセスを使用して、統合ファイル・システム・ディレクトリを PC ネットワーク・ドライブとしてマップし、PC ベースのエディターを使用してそのパスのファイルをローカル PC ファイルであるかのように編集します。
- SEU で編集し、ストリーム・ファイルへのコピー (CPYTOSTMF) コマンドを使用してそのファイルを従来の QSYS ファイル・システムからルート・ファイル・システム・パスへ移動します。
- 統合ファイル・システムのリンクを使用してソース物理ファイル (SRCPF) 内にソースを配置します。
  (実際のソースは QSYS メンバー内にありますが、そのメンバーを指すルート・ファイル・システム・リンクがあります。) リンクの追加 (ADDLNK) コマンドを使用してリンクを作成し、その後 SEU でメンバーを編集しますが、コンパイル時にはルート・ファイル・システムのパス名リンクを使用します。

SRCSTMF パラメーター

SRCSTMF パラメーターはソース・ストリーム・ファイルをパス名として識別します。コンパイルする ILE C ソース・コードが入っているストリーム・ファイルのパス名を指定します。パス名は、絶対修飾名と相対修飾名のどちらを指定することもできます。

大小文字の区別があるファイル・システムの場合、パス名は大/小文字を区別する可能性があります。

絶対修飾名は / または \ で始まります。パス名の先頭にある / または \ 文字は、最上層のディレクトリーであるルート (/) ディレクトリからパスが始まるものを示します。次に例を示します。
/Dir1/Dir2/Dir3/UsrFile

パス名の先頭に / または \ 文字がない場合は、現行ディレクトリーからパスが始まると見なされます。次に例を示します。
MyDir/MyFile

は、以下と同等です。
/CurrentDir/MyDir/MyFile

MyDir は、現行ディレクトリーのサブディレクトリーです。

チルド (*) 文字やワイルドカード ( * または ? ) はサポートされていません。

SRCSTMF は SRCMBR および SRCFILE と同時には使用できません。また、SRCSTMF を指定すると、コンパイラは TEXT (*SRCMBRTXT) を無視します。 TEXT 用の他の値は有効です。

ヘッダー・ファイルの検索

ソース・ストリーム・ファイルのパラメーターを使ってソース・ファイルを入力する際、コンパイラはさまざまな検索手法を使用します。コンパイラは、ライブラリー・リストの検索メソッドを使用しなくなりました。
組み込みファイル・リンク: ILE C/C++ ヘッダーおよびシステム・ヘッダーは、QSYSINC/H 内にあります。リンクはディレクトリー /QIBM/include にあります。

例えば assert.h の場合、リンクは次のようにになります。
- シンボリック・リンク・オブジェクトのリンクの表示: /QIBM/include/assert.h
- リンクのコンテンツ: /qsys.lib/qsysinc.lib/h.file/assert.mbr

ILE C++ 標準ライブラリー・ヘッダーは QSYSINC/STD にあります。リンクはディレクトリー /QIBM/include/std にあります。

例えば <algorithm> の場合、リンクは次のようにになります。
- シンボリック・リンク・オブジェクトのリンクの表示: /QIBM/include/std/algorithm
- リンクのコンテンツ: /QSYS.LIB/QSYSINC.LIB/STD.FILE/ALGORITHM.MBR

注:
QSYS.LIB ファイル・システム内のファイル（およびメンバー）は長さが 10 文字までに制限されているため、QSYSINC 内の C/C++ ヘッダー名は、その標準名が 10 文字より長い場合、切り捨てる必要があります。また、QSYS.LIB ファイル・システムが使用されている場合（すなわち SRCFILE/SRCMBR オプションが使用されている場合）、コンバイラーは意図的に組み込みファイルの名前を 10 文字に切り捨てます。例えば、#include <type_traits> は切り捨てられて include <type_trait> に（11 文字から 10 文字に）なります。これによって、ソース・コードを変更することなく正常にコンパイルすることができます。なお、組み込みファイルが最初の 10 文字において固有である限り、この仕組みは適切に機能します。

C++ TR1 (C++ ライブラリー拡張機能に関するドライフ・テクニカル・レポート ISO/IEC DTR 19768) から、一部の新しい C++ 標準ライブラリー・ヘッダー・ファイル名は、最初の 10 文字以内において固有でありません。特に、<unordered_set> と <unordered_map> という名前の 2 つの新しいヘッダー・ファイルが導入され、これら両方のファイルの切り捨ては "unordered_" にマップされます。

QSYSINC/STD 内での競合を避けるために、標準名 <unordered_map> または <unordered_set> でインクルードが許可されるように、ファイル・メンバー UNORDER_ が作成されました。 <UNORDER_> はラッパー・ヘッダー・ファイルとしての役割を果たし、さらに、QSYS.LIB ファイル・システムに対応するショート・ネームの STD ヘッダー・ファイル (<unord_map> および <unord_set>) を含んでいます。 ただし、<unord_map> および <unord_set> は QSYSINC/STD 内にある実際のヘッダーです。標準名 <unordered_map> または <unordered_set> が含まれている場合、<unord_map> と <unord_set> の両方が含まれます。

ディレクトリー /QIBM/include 内のヘッダー・ファイルおよびリンクには、ファイル名の長さに関する制限がありません。このため、すべての標準ヘッダーが /QIBM/include 内に格納され、標準名で出荷されます。

インクルード・ディレクティブ構文: #include ディレクティブ構文は、ルート・ソース・ファイル用に指定されたファイル・システム（統合ファイル・システム (IFS) または (DM) ファイル・システム）に依存します。

統合ファイル・システム (IFS) のコンパイル: IFS は、AIX® で見られるような階層ファイル・システムです。

ルート・ソース・ファイルに IFS ファイル指定が使用されている場合（すなわち、SRCSTMF オプションが使用されている場合）, そのコンパイル内のすべての #include ディレクティブは同様に IFS ファイル・システムに解決されます。以下の構文バリエーションがあります。
ファイル・システム（DM）のコンパイル： DM は、従来型の IBM i モノリシック（深さ固定の）ファイル・システムです。オブジェクトを含む複数のライブラリで構成されています。固定されたオブジェクト・タイプのセットがあります。ソース・ファイルは、*FILE オブジェクト・タイプ内の、メンバと呼ばれるサブオブジェクトにあります。すべてのネイティブ IBM i プロセスには順序付けられたライブラリー・リスト（*LIBL）があり、一般に、このライブラリー・リストを検索することで IBM i オブジェクトは解決されます。このライブラリー・リストには、次のように順序付けられた 3 つの構成要素があります。

- システム・ライブラリー・リスト（*SYSLIBL）：オペレーティング・システムを構成するライブラリーのセット。
- プロダクト・ライブラリー・リスト（*PRDLIBL）：通常、正式にライセンス交付を受けたプログラムは、実行時にプロダクト・ライブラリー・リストに追加されます。例えば、C および C++ コンパイラは、実行時にそのプロダクト・ライブラリー QCPPLE をライブラリー・リストに追加します。
- ユーザー・ライブラリー・リスト（*USRSLIBL）：構成または順序付けできる一連のライブラリー。

ルート・ソース・ファイルに DM ファイル指定が使用されている場合（すなわち、SRCFILE/SRCMBR オプションが使用されている場合）、そのコンパイル内のすべての #include ディレクティブは同様に DM ファイル・システムに解決されます。以下の構文バリアエーションがあります。

表 6-2. 統合ファイル・システムのコンパイル

<table>
<thead>
<tr>
<th>#include 指定</th>
<th>ライブラリー</th>
<th>ファイル</th>
<th>メンバー</th>
</tr>
</thead>
<tbody>
<tr>
<td>mbr</td>
<td>デフォルト検索</td>
<td>デフォルト・ファイル</td>
<td>mbr</td>
</tr>
<tr>
<td>mbr.file</td>
<td>デフォルト検索</td>
<td>ファイル</td>
<td>mbr</td>
</tr>
<tr>
<td>file/mbr</td>
<td>デフォルト検索</td>
<td>ファイル</td>
<td>mbr</td>
</tr>
<tr>
<td>file(mbr)</td>
<td>デフォルト検索</td>
<td>ファイル</td>
<td>mbr</td>
</tr>
<tr>
<td>file(mbr.ext)</td>
<td>デフォルト検索</td>
<td>ファイル</td>
<td>mbr または mbr.ext</td>
</tr>
<tr>
<td>lib/file/mbr</td>
<td>lib</td>
<td>ファイル</td>
<td>mbr</td>
</tr>
<tr>
<td>lib/file(mbr)</td>
<td>lib</td>
<td>ファイル</td>
<td>mbr</td>
</tr>
</tbody>
</table>

注：
1. デフォルトのライブラリー検索パスの場合:

   注：*SYINCPATH オプションが指定されているとき、コンパイラはユーザー・インクルード（""）をシステム・インクルード（< >）と同様に扱います。

   - ライブラリーが指定されておらず、かつ
     - #include 指定が < > で囲まれている場合：*LIBL を検索します。
     - #include 指定が " " で囲まれている場合：ルート・ソース・ファイルを含むライブラリーチェックします。そこで見つからなければ、*USRLIBL を検索します。それでも見つからなければ、*LIBL を検索します。
   - ライブラリーが指定されていて、かつ
### #include の使用

- **指定が `<>` で囲まれている場合**: `lib/file/mbr` のみを検索します。
- **指定が " " で囲まれている場合**: 指定された `library/file` でメンバーを検索します。

#### デフォルト・ファイルの場合:

- インクルードの形式が `#include <mbr>` の場合:
  - H: デフォルト・ファイルは H です。
  - C++: デフォルト・ファイルは STD です。
- インクルードの形式が `#include "mbr"` の場合: デフォルト・ファイルはルート・ソース・ファイルです。

#### mbr.ext の場合:

- **指定が `<>` で囲まれている場合**: `<file/mbr.h>` はファイル `file` 内のメンバー `mbr` に解決されます。この規則は POSIX のサポートを目的としたものです (例えば、`<sys/limits.h>` のような指定のインクルードを可能にするためです)。POSIX のサポートをアクティブにするメンバー名は、h または H の拡張子のみです。
- それ以外の場合、`<file/mbr.ext>` はファイル `file` およびメンバー `mbr.ext` に解決されます。
- **指定が " " で囲まれている場合**: `"file/mbr.ext"` はファイル `file` およびメンバー `mbr.ext` に解決されます。

---

### インクルード検索パスの規則

このセクションでは、以下に対する検索パスについて規定する規則をリストに示します。

- **INCDIR (インクルード・ディレクトリー)** コマンド・パラメーター
- **INCLUDE 環境変数**
- **STDINC/*NOSTDINC** コマンド・オプション
- **SYSINC/*NOSYSINC** コマンド・オプション

#### INCDIR (インクルード・ディレクトリー) コマンド・パラメーター

インクルード・ディレクトリーのパラメーター (INCDIR) は、モジュールの作成およびバイナードプログラムの作成のコンパイラ・コマンドと一緒に動作し、これを使用して、ソース・ストリーム・ファイルのコンパイル時にのみ、(`#include` ディレクトリで指定された) インクルード・ヘッダー・ファイルを検索するためのパスを再定義することができる。ソース・ファイルのロケーションがソース・ストリーム・ファイル (SRCSTMF) パラメーターで IFS バスとして定義されていない場合、または `#include` ディレクトリで完全 (絶対) パス名が指定されている場合、このパラメーターは無視されます。

このパラメーターは IFS ディレクトリーのリストを受け入れます。これらのディレクトリーは、入力順でインクルード検索パスに挿入されます。

インクルード・ファイルの検索パスは、次のディレクトリー検索順序に従ってファイルを探します。

#### 表 6-4. INCDIR コマンド・パラメーター

<table>
<thead>
<tr>
<th>#include の型</th>
<th>ディレクトリー検索順序</th>
</tr>
</thead>
</table>
| `#include <file_name>` | 1. INCDIR パラメーターでディレクトリーを指定している場合、コンパイラはまず、そのディレクトリー内で `file_name` を検索します。
| | 2. 複数のディレクトリーが指定されている場合、コンパイラはコマンド行での出現順にそれらのディレクトリーを検索します。
<p>| | 3. ディレクトリー <code>/QIBM/include</code> を検索します。|</p>
<table>
<thead>
<tr>
<th>#include の型</th>
<th>ディレクトリー検索順序</th>
</tr>
</thead>
</table>
| #include "file_name" | 1. 現在のソース・ファイルが存在しているディレクトリーを検索します。現在のソース・ファイルは、#include "file_name" ディレクトリールを含むファイルです。
2. INCDIR パラメーターでディレクトリーを指定している場合、コンパイラは、そのディレクトリー内で file_name を検索します。
3. 複数のディレクトリーが指定されている場合、コンパイラはコマンド行での出現順にそれらのディレクトリーを検索します。
4. ディレクトリー /QIBM/include を検索します。 |

例えば、INCDIR パラメーターに以下の値を入力したとします。

```
#include directory ./dir1
+ for more values ./dir2
```

また、ソース・ストリーム・ファイルで以下のヘッダー・ファイルをインクルードしたとします。

```
#include "foo.h"
#include <stdio.h>
```

コンパイラはまず、ルート・ソース・ファイルが存在するディレクトリーでファイル "foo.h" を検索します。ファイルが見つかった場合は、それをインクルードして検索は終了します。そうでない場合、コンパイラは、INCDIR に入力されたディレクトリーを、まず、"tmp/dir1" から探します。ファイルが見つかった場合は、このファイルをインクルードします。ディレクトリーが存在しない場合、またはそのディレクトリーにファイルが存在しない場合、コンパイラは現行作業ディレクトリー (「.」で表される) 内のサブディレクトリー "dir2" で引き続き検索します。同様に、ファイルが見つかった場合は、このファイルをインクルードします。そうでない場合は、INCDIR パス内のディレクトリーはもうすべて検索し尽くしたため、デフォルトのユーザー・インクルード・パス (/QIBM/include) を使用してヘッダーを検索します。

```
<stdio.h>
```
については、同じ順序で同じロジックに従いますが、ルート・ソース・ディレクトリーでの最初の検索は行いません。

**INCLUDE 環境変数**

<table>
<thead>
<tr>
<th>INCLUDE 環境変数の値は:</th>
</tr>
</thead>
<tbody>
<tr>
<td>コロン (:) で区切られたディレクトリーのパスを含みます。</td>
</tr>
<tr>
<td>順序をオーバーライドしません。</td>
</tr>
<tr>
<td>検索順序において、デフォルトのインクルード・パスより優先順位が高くなります。</td>
</tr>
<tr>
<td>検索順序において、INCDIR およびルート・ソースのディレクトリー (ユーザー定義のインクルード検索の場合) より優先順位が低くなります。</td>
</tr>
</tbody>
</table>

C と C++ の両方のコンパイラに定義された INCLUDE 環境変数がインクルード検索に含まれる場合、インクルード検索順序は次の表のようになります。
### 表6-5 インクルード検索順序

<table>
<thead>
<tr>
<th><code>#include</code> の型</th>
<th>ディレクトリー検索順序</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;file_name&gt;</code></td>
<td>1. INCDIR パラメーターでディレクトリーを指定している場合、コンパイラはまず、そのディレクトリー内で <code>file_name</code> を検索します。&lt;br&gt;2. 複数のディレクトリーが指定されている場合、コンパイラはコマンド行での出現順にそれらのディレクトリーを検索します。&lt;br&gt;3. INCLUDE 環境変数が定義されている場合、コンパイラは INCLUDE パスでの出現順にディレクトリーを検索します。&lt;br&gt;4. ディレクトリー <code>/QIBM/include</code> を検索します。</td>
</tr>
<tr>
<td><code>#include &quot;file_name&quot;</code></td>
<td>1. 現在のソース・ファイルが存在しているディレクトリーを検索します。現在のソース・ファイルとは、ディレクティブ <code>#include &quot;file_name&quot;</code> を含むファイルです。&lt;br&gt;2. INCDIR パラメーターでディレクトリーを指定している場合、コンパイラはそのディレクトリー内で <code>file_name</code> を検索します。&lt;br&gt;3. 複数のディレクトリーが指定されている場合、コンパイラはコマンド行での出現順にそれらのディレクトリーを検索します。&lt;br&gt;4. INCLUDE 環境変数が定義されている場合、コンパイラは INCLUDE パスでの出現順にディレクトリーを検索します。&lt;br&gt;5. ディレクトリー <code>/QIBM/include</code> を検索します。</td>
</tr>
</tbody>
</table>

注: この機能は、ソース・ストリーム・ファイルのコンパイルでのみ使用可能です。

**STDINC/*NOSTDINC コマンド・オプション**: CRTCMOD/CRTCPPMOD コマンドと CRTBNDC/CRTBNDCPP コマンドの OPTION パラメーターに、*STDINC/*NOSTDINC コマンド・オプションが追加されました。

**STDINC コマンド・オプションと *NOSTDINC コマンド・オプションは、CRTCMOD/CRTCPPMOD コマンドと CRTBNDC/CRTBNDCPP コマンドで動作します。**

**NOSTDINC オプションでは、検索順序からデフォルトのインクルード・パス (IFS ソース・ストリーム・ファイルの場合は `/QIBM/include、ソース・ファイル・メンバーの場合は QSYSINC) を除去できます。一方、*STDINC オプションでは、順序の最後にデフォルトのインクルード・パスを保持します。*STDINC がデフォルトです。**

**STDINC オプションは、ソース・ファイル・メンバーの以前の SYSINC パラメーターと同様に動作します。これらのオプションと古いパラメーター値との関係は、以下のとおりです。**

<table>
<thead>
<tr>
<th>SYSINC の値</th>
<th>同等の新しいコマンド・オプション</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>YES</em></td>
<td><em>STDINC</em></td>
</tr>
<tr>
<td><em>NO</em></td>
<td><em>NOSTDINC</em></td>
</tr>
</tbody>
</table>

**INCDIRFIRST/*NOINCDIRFIRST コマンド・オプション:**

**INCDIRFIRST オプションでは、ユーザー・インクルード検索で検索順序の最初に (つまり、ルート・ソース・ファイル・ディレクトリーより前に)、INCDIR パラメーターでリストされたディレクトリーを処理で**
表6-7. INCDIRFIRST コマンド・オプション

<table>
<thead>
<tr>
<th>include の型</th>
<th>ディレクトリ検索順序</th>
</tr>
</thead>
</table>
| #include "file_name" | 1. INCDIR パラメーターでディレクトリを指定している場合、コンパイラは、そのディレクトリ内で file_name を検索します。
2. 複数のディレクトリが指定されている場合、コンパイラはコマンド行での出現順にそれらのディレクトリを検索します。
3. 現在のルート・ソース・ファイルが存在しているディレクトリを検索します。
4. INCLUDE 環境変数が定義されている場合、コンパイラは INCLUDE パスでの出現順にディレクトリを検索します。
5. *NOSTDINC コンパイラ・オプションが選択されていない場合は、デフォルトのインクルード・ディレクトリである /QIBM/include を検索します。 |

*SYSINCPath*/NOSYSINCPath コマンド・オプション: *SYSINCPath*/NOSYSINCPath コマンド・オプションは、モジュールの作成およびバイナド・プログラムの作成のコマンドで動作します。

*SYSINCPath オプションでは、ユーザー・インクルードの検索パスをシステム・インクルードの検索パスに変更します。これは、ユーザー #include ディレクトリ内の二重引用符 (#include "file_name") を不等号括弧 (#include <file_name>) に変更することと同じです。 *NOSYSINCPath がデフォルト値です。

ソース・ストリーム・ファイルの指定に関する考慮事項: プログラムまたはモジュールの作成時に  
SRCSTMF パラメーターを指定する際、*MODULE オプジェクトにはソース・ファイル属性情報は含まれません。

SRCFILE/SRCMBR パラメーターを使用してソースが指定されている場合、INCDIR パラメーターおよび INCLUDE 環境変数は無視されます。ソースファイル・システム内にある場合、組み込みファイルの検索にライブラリー・リストが使用されます。ライブラリー・リスト (*LIBL) には、ディレクトリー・ファイル構造の概念はありません。

ソース・ファイルが SRCSTMF パラメーターで指定されていない場合、組み込みファイルの検索にそのジョブの *LIBL が使用されます。

絶対インクルード・パス名に関する制約事項: 絶対 (完全) パス名を #include ディレクトリに指定する場合、INCDIR パラメーター および INCLUDE 環境変数は無効です。

例:

次のことを前提とします。
* C/C++ ソース・ファイル内に #include "myinc.h" のようなステートメントがある。
• QSYS ファイル・システムから、以下のコマンドで SRCSTMF パラメーターを使用してソース・メンバーをコンパイルしようとしている。

CRTCMD MODULE(MYSOURCE) SRCSTMF('/qsys.lib/goodness.lib/qcppsrc.file/mysource.mbr')

ILE C/C++ は、/qsys.lib/goodness.lib/qcppsrc.file/myinc.h というものをつけようとします。.h は QSYS ファイル・システムでは有効なオブジェクト・タイプではないことから、これは無効な統合ファイル・システムのファイル名です。

QSYS ファイル・システム内にあるヘッダー・ファイルを使用する場合は、以下のいずれかを実行する必要があります。

• 以下に示すように、ソース・コード内でパスを指定する。

```c
#include "/qsys.lib/goodness.lib/h.file/myinc.mbr"
```

• [6-44 ページの「統合ファイル・システム・ソースの使用例」]で示すように、検索バスを適切に設定する。その後、以下に示すように、include ステートメントからそのパスを除外することができます。

```c
#include "myinc.mbr"
```

ソース・ファイルおよびヘッダー・ファイルに関する推奨事項: 階層ファイル・システムを持つ他のプラットフォーム (例えば Microsoft Windows、UNIX、または OS/2 オペレーティング・システムなど) から移植する場合は、統合ファイル・システム (IFS) の方がそれらのファイル・システムとの互換性が高いことを考慮してください。現在の C/C++ ソース・コードが変更されないようにするには、ソース・ファイルおよびヘッダー・ファイルを IFS 内に置いてください。

プリプロセッサー出力
SRCSTMF を OPTION(*PPONLY) と共に指定すると、プリプロセッサーは新しい拡張子 .i を使用してストリーム・ファイルを現行ディレクトリに書き込みます。例えば、SRCSTMF('/home/MOE/mainprogram.c') を OPTION(*PPONLY) と共に指定すると、プリプロセッサーは mainprogram.i という名前のストリーム・ファイルとして出力を現行ディレクトリに書き込みます。この書き込みが行われるようにするには、ユーザーが現行ディレクトリに対する *RW 権限を付与されている必要があります。

リスト出力
コンバイラーは、リスト出力をユーザー指定の IFS ファイルに送信できるほか、スプール・ファイルにも送信できます。Prolog はソース・ファイルをバス名から識別します。

```c
모듈名 . . . . . . . . . : TEST
ライブラリ . . . . . . . : MOE
ソース・ストリーム・ファイル : /home/MOE/src/mainprogram.c
```

注: SRCFILE() および SRCSTMF() が指定されている場合、ソース・ストリーム・ファイルは Prolog に含まれません。

また、このリストは、組み込みファイルをバス名から識別します。

```c
**** FILE TABLE SECTION ****
 0 = hello.cpp
 1 = /QIBM/include/iostream.h
 2 = /QIBM/include/string.h
```

リストに含まれる内容は以下のとおりです。

• 任意の #include ディレクトリに指定されたファイル
• SRCSTMF() または SRCFILE()/SRCMBR() オプションで指定または暗黙指定されたファイル
注: この状況は、データベース・ファイルまたはストリーム・ファイル・ソースのいずれでも発生します。

OUTPUT オプションの形式は OUTPUT(*NONE  | *PRINT  | filename) となります。ここで、*PRINT はコンパイラに出力をスプール・ファイルに送信させ、filename はコンパイラに出力をユーザー指定の IFS ファイルに送信させます。

コード・ページおよび CCSID
ソース物理ファイルの場合、コンパイラは ILE C ソースの CCSID を考慮します。ストリーム・ファイルのコンパイルの場合は、類似のスキームが存在します。ストリーム・ファイルにはコード・ページ属性があります。
コンパイラはソース・ファイルを変換し、その際にコード・ページをルート・ソースに変換します。

ソース・ストリーム・ファイルは、ASCII システムでマウント・ファイルを使用して入力された可能性があります。その場合、コンパイラが、ストリーム・ファイルと関連付けられた ASCII コード・ページ (例えば 437 など) から EBCDIC (例えば 37 など) に変換します。

Unicode ワイド文字リテラルのサポートは、コンパイラ・コマンドに LOCALETYP(*LOCALEUCS2) を指定することで、プログラムをビルトする際に使用可能にすることができます。詳しくは、9-9 ページの『国際ロケールのサポート』を参照してください。

ほとんどのファイル転送ユーティリティは、ASCII ベースのファイル・システムをストリーム・ファイル・ソースの作成に使用できるようにするため、自動的に変換を行うように構成することができます。

回避すべき危険性
テキスト・ファイルの最後に EOF マーカーを挿入する ASCII エディターを使用して、ワークステーションで作成したソース・ファイルはすべて、ILE C/C++ コンパイラでコンパイルしたときに、無効文字の警告メッセージを生成します。これには、メイン・ソース・ファイルが含まれます。この問題は、ソース・ファイルが、IBM i のルート (/) ファイル・システムにコピーまたは保存されたときに起こります。これは、ASCII と EBCDIC のコード・ポイントの変換が原因です。

ファイルの最後の文字に言及している無効文字のメッセージを受け取ったら、ファイルに EOF マーカーがある可能性があります。この問題を回避する方法の 1 つとして、EOF マーカーを追加しないエディターを使用します。

あるいは、ファイル転送プロトコル (FTP) ユーティリティを使用することもできます。FTP では、ルート(/) ファイル・システム・ファイルがコード・ページ 819 または 37 になります。put コマンドにより前にターゲット IBM i に以下のいずれかの FTP コマンドが実行されると、コード・ページ 819 のファイルになります。

- ascii
- quote type a

put コマンドにより前にターゲット IBM i にコマンド quote type e を実行した場合は、put によって、コード・ページ 37 (EBCDIC) のファイルになります。このファイルは、FTP を使用してルート・ファイル・システムに転送されると、このファイルが転送前に存在したかどうかには関係なく、(上記で説明した、前に実行したコマンドに応じて) コード・ページ 819 またはコード・ページ 37 で作成されます。

コード・ページ 37 で統合ファイル・システムに転送されたファイルは、コンパイルに失敗します。
統合ファイル・システム・ソースの使用例

統合ファイル・システム名の最も基本的な項目で、パス情報は指定されていません。

```
#include "special/mystuff.h"

class test : public base
{
//

また、INCDIR の値が /mydir に設定されているとします。この場合、インクルードされたヘッダー・ファイルは、まず、/goodness/special/mystuff.h として検索され、次に /mydir/special/mystuff.h として検索されます。

ストリーム入出力の使用

次のセクションでは、以下を使用するためのストリーム入出力の要件について説明します。

- 大きなファイル
- オープン・モード
- 行終了文字
```
大きなファイル: C または C++ ランタイム環境では、「モジュールの作成」および「バインド済みプログラムの作成」コマンド・ブロントのシステム・インターフェース・キーワード（SYSIFCOPT）で
*IFSIO オプションを指定することで、2 GB までのサイズのファイルのストリーム入出力が使用可能になります。

いずれかのコマンドで SYSIFCOPT キーワードを使用する際は、次の形式に従ってください。
CRTPPMOD MODULE(QTEMP/IFSIO) SRCFILE(QCPPLE/QACSRC) SYSIFCOPT(*IFSIO)
CRTBNDCPP PGM(QTEMP/IFSIO) SRCFILE(QCPPLE/QACSRC) SYSIFCOPT(*IFSIO)

*IFSIO オプションが指定されている場合、コンパイルは __IFS_IO__ マクロを定義します。__IFS_IO__が定義されている場合、<stdio.h> でストリーム入出力と関連付けられたプロトタイプは定義されなくなります。ヘッダー・ファイル <ifs.h> は <stdio.h> によって組み込まれます。これによって、統合ファイアル・システム対応の C ストリーム入出力に関連付けられたすべての構造体およびプロトタイプが宣言されます。

64 ビット・バージョンの統合ファイル・システム・インターフェースでは、2 GB までおよびそれを超えるサイズのファイルで ILE C ストリーム入出力を使用することができます。（2 GB を超えるファイルの C++ ストリーム入出力はサポートされていません。）64 ビットのインターフェースを使用可能にするには、CRTPPMOD または CRTBNDCPP コマンド・ブロントで *IFS64IO オプションを
SYSIFCOPT キーワードと共に指定します。このオプションが指定されると、コンパイラは
__IFS64_IO__ マクロを定義します。このマクロは、例えば、open() 関数を open64() 関数に再マップし、
64 ビットの索引付けを可能にします。

オープン・モード:

C++

 ifstream(), ofstream(), および ofstream() クラスには新しいオープン・モード ios::text があります。
このモードは、ファイルをテキスト・モードで開きます。

デフォルトでは、入出力ストリームはバイナリー・モードで開きます。

行終了文字:
• 入力または出力が (例えば read() または write() メソッドを使用して) フォーマットされていない場合、改行（\n）文字は出力時に \n に展開されず、\r 文字は入力時にストリップされません。
• 入力または出力が (>> または << 演算子を使用して) フォーマットされている場合、改行（\n）
文字は出力時に \n に展開されませんが、\r 文字はすべて入力時にストリップされます。

復帰改行（\r）文字を追加したい場合は、fopen() 関数を crln=Y (デフォルト) と共に使用してください。
ファイル・システムおよび装置の処理

このトピックでは、以下を行うための方法について説明します。
- 外部ファイル記述の検索
- 切断モードでの作業
- 外部記述の物理および論理データベース・ファイルの組み込み
- 物理および論理データベース・ファイルと分散データの使用
- コミットメント制御の使用
- 表示装置ファイル、印刷出力ファイル、ICF ファイル、テーブ・ファイル、ディスケット・ファイル、および保管ファイルの使用
- 装置属性のフィードバック域の使用

プログラムでの外部記述ファイルの使用

このトピックでは、以下を行うための方法について説明します。
- 外部記述ファイルを作成する
- レベル検査を使用して、プログラムがコンパイルされた記述がまだ機能することを検証する
- C/C++ 構造体でのフィールド位置合わせの問題を回避する
- プログラムでの外部フィールド定義の組み込み
- 標識を定義して使用する
- プログラムで物理および論理データベース・ファイルを組み込む
- プログラムで装置ファイルを組み込む
- プログラムで複数のレコード様式を組み込む
- プログラムでパック 10 進数データを組み込む

外部記述データベース・ファイルの作成

外部記述ファイル は、フィールド記述がファイルの一部として保管されているファイルのことです。記述には、ファイルのタイプ（データや装置など）、レコード様式、および各フィールドとその属性の記述に関する情報が含まれています。

以下の一覧かを使用して、外部記述データベース・ファイルを作成できます。
- SQL400 データベース
- 外部記述ファイルに DDS を使用した対話式データ定義ユーティリティー (IDDD)
- データ記述仕様（DDS）

ILE C プリプロセッサーは、#pragma mapinc ディレクティブを #include ディレクティブと同様に使用した場合、外部ファイル記述から C 構造体タイプ定義を自動的に作成します。

注: IFS ファイルをコンパイルする場合は、#pragma mapinc ディレクティブを使用することはできません。プログラムへの IFS ファイルの組み込みについて詳しくは、10-1 ページの「GENCSRC ユーティリティと #pragma mapinc ディレクティブ」を参照してください。
#pragma mapinc ディレクティブは、等のファイアル・フォーマットおよびフィールドのみをコンパイラに識別します。ILE C プログラムにファイル記述は組み込みません。ファイル記述を組み込むには、ILE C プログラムで #include ディレクティブをコーディングする必要があります。

#include で #pragma mapinc ディレクティブの include-name パラメーターを参照します。include ディレクティブは、ソース・プログラム内で #pragma mapinc ディレクティブの後にコーディングする必要があります。

例えば、ファイル EXAMPLE/TEST のレコード様式 FMT に関する入力フィールドのタイプ定義を組み込むには、以下のステートメントが、以下に示されている順序でプログラム内に記載されている必要があります。

#pragma mapinc("tempname","EXAMPLE/TEST(FMT)","input","d",""
#include "tempname"

タイプ定義の作成

ILE C プログラムに組み込むタイプ定義構造体を作成するには、オプション・パラメーターを使用します。ヘッダー記述も作成されます。このヘッダー記述には、外部ファイルに関する情報が含まれます。

C++ ユーザーは、タイプ定義の作成に GENCSRC ユーティリティを使用する必要があります。

C ユーザーは、タイプ定義の作成に GENCSRC ユーティリティーおよび #pragma mapinc ディレクティブのいずれも使用できます。

注: GENCSRC ユーティリティーと #pragma mapinc ディレクティブの違いについては詳しくは、10-1 ページの『GENCSRC ユーティリティーと #pragma mapinc ディレクティブ』を参照してください。

コンパイラ・リストでタイプ定義を表示するには、OPTION(*SHOWUSR) を指定します。

注: OPTION(*SHOWINC) は、GENCSRC または #pragma mapinc によって作成された組み込みファイルを展開し、またシステム組み込みファイルも展開します。

ヘッダー記述の作成

各フォーマットのヘッダー記述には、以下の情報が含まれています。
- 外部ファイルのファイル名またはライブラリ名
- ファイル・タイプ (物理、論理、装置)
- ファイルの作成日
- レコード様式名
- レコード様式レベル ID (レベル検査情報)

例えば、以下のディレクティブを使用して下のヘッダーが作成されます。

#pragma mapinc("payroll","example/test(fmt1)","input",""
#include "payroll"

/* ------------------------------------------------------------------ */
/* PHYSICAL FILE: EXAMPLE/TEST */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT1 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* ------------------------------------------------------------------ */
以下に、型構造体の型定義の例を示します。

typedef struct {
  
  } LIBRARY_FILE_FORMAT_tag_t;

#include <stdio.h>

#define MAPINC FILE_FORMAT

ライブラリ名およびファイル名に表示される文字で C 言語により有効と認識されないものはすべて、下線 (_) 文字に変換されます。

注: 特殊文字 #、@、または $ はライブラリ名およびファイル名では使用しないでください。これらの文字がライブラリ名およびファイル名で使用されると、それらも下線 (_) 文字に変換されます。

構造体名のタグは、構造体定義に含まれるフィールドのタイプを示しています。タグに使用できる値は次のとおりです。

表7-1 フィールド・タイプおよびタグの値

<table>
<thead>
<tr>
<th>フィールド・タイプ</th>
<th>テグ</th>
</tr>
</thead>
<tbody>
<tr>
<td>人力</td>
<td>i</td>
</tr>
<tr>
<td>出力</td>
<td>o</td>
</tr>
<tr>
<td>議方</td>
<td>両方</td>
</tr>
<tr>
<td>キー</td>
<td>キー</td>
</tr>
<tr>
<td>標識</td>
<td>indic</td>
</tr>
<tr>
<td>nullflds</td>
<td>nmap/nkmap</td>
</tr>
</tbody>
</table>

他のリストされたフィールド・タイプに使用される命名規則とは異なり、フィールド・タイプ Lvchk が指定されると、作成された構造体の型の配列の名前は _LVCHK_T となります。

複数のフォーマットに対して外部ファイル記述を組み込む場合は、複数のフォーマット名 (format1 format2) または (*ALL) を #pragma mapinc ディレクティブで指定します。ヘッダー記述および型定義は、各フォーマットに対して作成されます。

lname オプションが指定されていて、#pragma mapinc ディレクティブ内のファイル名の長さが 10 文字を超える場合、コンパイラが生成する型定義ではシステム生成の 10 文字の名前が使用されます。

レコード様式名の指定

レコード様式 は、すべてのフィールド、およびレコード内のそれらのフィールドの配置を記述したものでです。#pragma mapinc ディレクティブで名前を指定することで、ILE プログラムに外部記述ファイルからのレコード様式を組み込むことができます。複数の様式名を指定できます、あるいは、特殊値 *ALL を指定して、ファイルのすべてのレコード様式を組み込むこともできます。

作業しているファイルに複数のレコード様式が含まれている場合、_Rformat() 関数を使用して、後続の入出力操作用の様式を設定します。
レコード様式関数は、表示装置、ICF、および印刷出力ファイルを処理する際に役立ちます。論理ファイルにも複数のレコード様式を含めることができます。

_Rfomat() を使用して名前を明示的に設定しない限り、装置ファイルのレコード様式名はデフォルトでブランクになります。_Rformat() にプランク名を渡すことで、様式名をプランクにリセットできます。

```c
#pragma mapinc ディレクティブに指定されたオプション (input、output、both、key、indicators、または nullfields) に一致するフィールドがレコード様式に含まれていない場合、ヘッダー記述の後に以下のコメントが表示されます。

/* FORMAT HAS NO FIELDS OF REQUIRED TYPE */
```

注: レコード様式名で #、@、または $ を使用することはできません。これらの文字は ILE ID で許可されず、タイプ定義名で表示できません。#、@、または $ が含まれているレコード様式名がある場合、その文字は、それぞれ小文字の p、a、および d に変換されます。

レコード・フィールド名の指定
レコード・フィールド名を指定する際、次の考慮事項に注意してください。

- すべての DDS キーワードは ILE ライブラリーおよびコンパイラによってサポートされています。
  TEXT キーワードと関連付けられた実際のコメントは、生成される型定義で大文字に変換されます。
  ALIAS キーワードはサポートされており、これは生成される型定義に別名のフィールド名を組み込みます。
- DDS 変数名でサポートされている特殊文字の一部は、ILE コンパイラおよびライブラリーではサポートされていません。フィールド名で特殊文字 (@、#、または $) を使用すると、それらの文字は、生成される型定義で小文字の a、p、および d に変換されます。

注: フォーマット名に無効な C 文字が含まれている場合、それらは下線 (_) 文字に変換されます。

型定義へのデータベース・ファイルの組み込み
データベース・ファイルの入力バッファーと出力バッファーは同じ形式になります。入力を指定する際、外部記述データベース・ファイルで INPUT または BOTH として定義されているフィールドが型定義に組み込まれます。両方を指定する際、INPUT、OUTPUT、BOTH のいずれかとして定義されたフィールドが型定義に組み込まれます。

すべてのフィールドが BOTH または INPUT として定義されている場合は、1 つの型定義構造体のみが型定義内に生成されます。

構造体の型 (KEY フィールド) の定義
KEY フィールドに対する個別の構造体の型定義をフォーマットに組み込むには、#pragma mapinc ディレクティブで KEY オプションを指定します。構造体定義の各フィールドの横に、外部記述ファイル内でキー・フィールドがどのように定義されているかを示すコメントがリストされます。

```c++
C++ ユーザーは、構造体の型定義に GENCSRC ユーティリティーを使用する必要があります。
```

```c
C ユーザーは、構造体の型定義に GENCSRC ユーティリティーまたは #pragma mapinc ディレクティブのいずれかを使用することができます。
```

注: GENCSRC ユーティリティーと #pragma mapinc ディレクティブの違いについて詳しくは、IBM i: ILE C/C++ プログラマーの手引きの『GENCSRC ユーティリティーと #pragma mapinc ディレクティブ』を参照してください。
例:
以下の ILE C プログラムには、外部記述データベース・ファイル CUSMSTL を組み込むための #pragma mapinc ディレクティブが含まれます。

```c
#include "custmf"
```

以下の例には、ライブラリー MYLIB 内のファイル T1520DD8 に対する DDS が含まれています。

```
A* CUSTOMER MASTER FILE -- T1520DD8
  A    R CUSREC       TEXT('Customer master record')
  A    CUST          5    TEXT('Customer number')
  A    NAME          20   TEXT('Customer name')
  A    ADDR          20   TEXT('Customer address')
  A    CITY          20   TEXT('Customer city')
  A    STATE         2    TEXT('State abbreviation')
  A    ZIP           5 0   TEXT('Zip code')
  A    ARBAL          10 2  TEXT('Accounts receivable balance')
  A    K CUST       
A* 
A*
```

図 7-2. T1520DD8 — 顧客レコード用の DDS ソース

プログラム T1520EDF は、#pragma mapinc ディレクティブを使用して T1520DD8 に定義されたファイル・フィールド構造体を生成します。
/* This program contains the #pragma mapinc directive to
 * include the externally described database file T1520DDDB.
 * This program reads customer information from a terminal and issues
 * a warning message if the customer's balance is less than $1000.
 */

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>

#pragma mapinc("custmf","QGPL/T1520DDDB(cusrec)","both key","_P")
#include "custmf"

int main(void)
{
    /* Declare x of data structure type QGPL_T1520DDDB_CUSREC_both_t. */
    /* The data structure type was defined from the DDS specified. */
    QGPL_T1520DDDB_CUSREC_both_t x;

    /* Get information from entry. */
    printf("Please type in the customer name (max 20 char).\n");
    gets(x.NAME);
    printf("Please type in the customer balance.\n");
    scanf("%D(10,2)",&x.ARBAL);

    /* Prints out warning message if x.ARBAL<1000. */
    if (x.ARBAL<1000)
    {
        printf("%s has a balance less than $1000!\n", x.NAME);
    }
}

図 7-3. T1520EDF — 外部記述データベース・ファイルを組み込むための ILE C ソース

型定義は、ILE C ソース・プログラム内で指定された #pragma ディレクティブに基づく ILE C ソース・リスト内に作成されます。

出力は以下のようになります。

Please type in the customer name (max 20 char).
> James Smith
Please type in the customer balance.
> 250.58
James Smith has a balance less than $1000!
実行キーを押して端末セッションを終了してください。

プログラム・リストの DDS の部分は以下のようになります。
ファイルのロング・ネームの使用

`#pragma mapinc` ディレクティブでは、最大で 128 文字の長さのファイル名および最大で 30 文字の長さのレコード・フィールド名がサポートされます。

SQL ロング・ネーム・フォーマットをサポートするために、`LNAME` オプションが `#pragma mapinc` に追加されています。SQL ロング・ネームは、名前の最初の 5 文字に 5 桁の固有の数字が続く 10 文字のショート・ファイル名にマップされます。例えば、LONGSQLTABLENAME のシステム・ショート・ネームは、LONGS000001 です。

ロング・レコード・フィールド名は、10 文字のショート・ネームにマップされません。`LNAME` オプションを指定した場合、ファイル名のロング・ネーム・フォーマットが使用されているものと想定されます。ファイル名が 10 文字を超えている場合、その名前は、関連付けられたショート・ネームに内部的に変換されます。このショート・ネームは、外部ファイル定義を抽出するために使用されます。10 文字以下の正規のショート・ネームを指定した場合、変換は行われません。

### C

`#pragma mapinc` ディレクティブは、`LNAME` オプションの指定の有無を問わず、生成されるタイプ定義で 30 文字のレコード・フィールド名を使用します。ロング・ネーム・フォーマットを使用して指定されたファイル名では、生成されるタイプ定義は、関連付けられた正規の 10 文字のショート・ファイル名を使用します。

### C++

C++ ユーザーは、`GENCSRC` ユーティリティーを使用して、外部ファイルからタイプ定義を作成する必要があります。

### C

C ユーザーは、`GENCSRC` ユーティリティーおよび `#pragma mapinc` ディレクティブのいずれを使用しても外部ファイルからタイプ定義を作成できます。

注: `GENCSRC` ユーティリティーと `#pragma mapinc` ディレクティブの違いについて詳しくは、10-1 ページの「GENCSRC ユーティリティーと `#pragma mapinc` ディレクティブ」を参照してください。

/* ------------------------------------------------------------------------*/
/* PHYSICAL FILE: QGPL/T1520DD8 */
/* FILE CREATION DATE: 93/08/14 */
/* RECORD FORMAT: CUSREC */
/* FORMAT LEVEL IDENTIFIER: 4E9D9ACA60E00 */
/* ------------------------------------------------------------------------*/
typedef _Packed struct {
    char CUST[5]; /* Customer number */
    char NAME[20]; /* Customer name */
    char ADDR[20]; /* Customer address */
    char CITY[20]; /* Customer city */
    char STATE[2]; /* State abbreviation */
    decimal(5,0) ZIP; /* Zip code */
    decimal(10,2) ARBAL; /* Accounts receivable balance */
}QGPL_T1520DD8_CUSREC_both_t;

typedef _Packed struct {
    char CUST[5]; /* DDS - ASCENDING */
    /* STRING KEY FIELD */
}QGPL_T1520DD8_CUSREC_key_t;

図7-4. プログラム T1520EDF からの出力リスト — 顧客マスター・レコード
記述を検証するためのレベル検査

外部記述ファイルを使用する ILE C/C++ プログラムのコンパイル時には、コンパイラーが、プログラムで参照されるファイルのレコード・レベル記述およびフィールド・レベル記述を抽出し、それらの記述をコンパイルされるプログラムの一部とします。プログラムを実行するときに、プログラムのコンパイル時に使用した記述が、現行の記述であるかどうかを検証できます。このプロセスは、レベル検査 と呼ばれています。

関連ヘッダー・ファイルが作成されるときに、サーバーは、各レコード構成に固有のレベル ID を割り当てます。以下的情報により、レコード ID が決定されます。

- レコード種別名
- フィールド名
- レコード構成の合計数
- レコード構成内のフィールド数
- フィールド属性（例えば、長さおよび小数点以下の桁数）
- レコード構成内のフィールドの順番

注: 大規模レコード構成 (多くのフィールド) を持つファイルは、構成が多少異なっていても、同じ構成のレベル ID を持つことができます。コピー先ファイルとコピー先ファイルのレコード構成名が同一の場合、これらのファイルをコピーしている時に問題が発生する可能性があります。

前のリストのデータ記述仕様 (DDS) 項目のいずれかを変更した場合、レベル ID も変更されます。

ファイルを作成または変更し、レベル検査が必要であることを指定した場合、以下のようにになります。

- システムがレベル ID を検査して、使用しているレコード構成の記述がプログラムのコンパイル時以降に変更されたかどうかを判別します。
- プログラムがファイルを処理できないほどその情報が変更されている場合、システムは、プログラムにこの状態を通知します。

この変更がプログラムで使用するフィールドに影響する場合は、プログラムをあらためてコンパイルしてそれが適切に実行できるようにしなければなりません。

C++ ユーザーは、レベル検査に GENCSRC ユーティリティーを使用する必要があります。

C ユーザーは、レベル検査に GENCSRC ユーティリティーおよび #pragma mapinc ディレクティブのいずれも使用できます。

注: GENCSRC ユーティリティーと #pragma mapinc ディレクティブの違いについて詳しくは、10-1 ページの『GENCSRC ユーティリティーと #pragma mapinc ディレクティブ』を参照してください。

プログラムの使用するフィールドに影響しない変更を行った場合には、そのファイルに対してオーバーライド コマンドを入力し LVLCHK(*NO) を指定することで、コンパイルし直さずにプログラムを実行することができます。例えば、あるフィールドがデータベース・ファイルのレコード構成の終わりに追加された場合に、プログラムがその新しいフィールドを使わないとします。LVLCHK(*NO) を指定したデータベース・ファイル・オーバーライド (OVRDBF) コマンドを使用すれば、プログラムをコンパイルし直さずに実行することができます。

注: DDM でデータベース・ファイル・オーバーライド (OVRDBF) コマンドを使用して、プログラムで指定されたローカル・データベース・ファイルを DDM ファイルでオーバーライドできます。DDM ファイルにより、プログラムによって、ローカル・データベース・ファイルではなく、関連リモート・ファイルが使用されるようになります。

IBM i: ILE C/C++ プログラマーの手引き
レベル検査を使用することで、ファイルの保全性が確保されます。これにより、予測不能な結果の可能性が警告されます。

レベル検査に替わる方法としては、ファイル記述を表示して分析し、変更によりプログラムが影響を受けるかどうかを判別します。記述を表示するには、「ファイル・フィールド記述の表示 (DSPFD)」コマンドを使用することができます。あるいは、元倉ステートメント人力ユーティリティ (SEU) がある場合は、ファイルの DDS が入っているソース・ファイルを表示することができます。ファイルに定義されている様式レベル ID を表示するには、ファイル記述表示 (DSPFD) コマンドを使用します。

注: レベル ID を表示する場合には、ファイル ID ではなく、レコード様式 ID が比較されることに注意してください。

**GENCSRC ユーティリティを使用したレベル検査**

GENCSRC ユーティリティを使用して、C/C++ プログラムで使用する外部記述ファイル情報を取得します。ユーティリティによって以下の処理が実行されます。

- 組み込みファイル用の型定義構造体を含む C/C++ ヘッダー・ファイルを作成する。
- C/C++ 組み込みファイルの作成をサポートする。

**SLTFLD キーワードを使用してレベル検査をオンにします。**

注: SLTFLD キーワード用のオプション・リストについては、10-1 ページの「GENCSRC ユーティリティ」と #pragma_mapinc ディレクティブを参照してください。

GENCSRC コマンドでキーワード SLTFLD を値*LVLCHK と共に指定すると、タイプ _LVLCHK_T (構造体の配列) の生成に加え、タイプ _LVLCHK_T の変数も生成されます。タイプ _LVLCHK_T のこの変数の名前は、以下の一部または全部に基づいています。

- OBJ キーワード用に指定されたファイル名 (7-10 ページの図 7-5 を参照)
- メンバー名 (7-10 ページの図 7-6 を参照)
- TYPEDEFPFX キーワードの値 (7-11 ページの図 7-7 を参照)
- 組み込み名

注: 以下のそれぞれの図で、組み込み名は実際には組み込みファイル (すなわち SRCFILE/SRCMBR または SRCSTMF キーワード) です。

SLTFLD(*LVLCHK) がデフォルトの TYPEDEFPFX(*OBJ) と共に指定されている場合、レベル検査構造体の名前は OBJ キーワードに指定されたファイル名と組み込み名に基づいています (10-1 ページの表 10-1 を参照)。GENCSRC コマンドは、以下の例に示すように、mylib myfile mymbr lvlchkという名前のレベル検査構造体を生成します。
GENCSRC OBJ('/QSYS.LIB/MYLIB.LIB/MYFILE.FILE') SRCFILE(INCLIB/H)
SRCMBR(MYMBR) SLTFLD(*LVLCHK) TYPEDEFPFX(*OBJ)

または
GENCSRC OBJ('/QSYS.LIB/MYLIB.LIB/MYFILE.FILE')
SRCSTMF('incdir/mymbr.h') SLTFLD(*LVLCHK) TYPEDEFPFX(*OBJ)

/* ----------------------------------------------------------------------- */
// PHYSICAL FILE : MYLIB/MYFILE
// FILE LAST CHANGE DATE : 2001/09/13
// RECORD FORMAT : FORMAT1
// FORMAT LEVEL IDENTIFIER : 38A624C5F3B51
/* ----------------------------------------------------------------------- */

_LVLCHK_T mylib_myfile_mymbr_lvlchk = {
  .
};

注: レベル検査名はソース・ロケーション (ライブラリー、ファイル、メンバー) に依存します。

図7-5. デフォルトの TYPEDEFPFX(*OBJ) を使用した SLTFLD(*LVLCHK) の例

SLTFLD(*LVLCHK) が TYPEDEFPFX(*NONE) と共に指定されている場合、レベル検査構造体の名前は
メンバー名に基づいています。図7-6のコマンドは、mymbr_lvlchk という名前のレベル検査構造体を生成
します。

GENCSRC OBJ('/QSYS.LIB/MYLIB.LIB/MYFILE.FILE') SRCFILE(INCLIB/H)
SRCMBR(MYMBR) SLTFLD(*LVLCHK) TYPEDEFPFX(*NONE)

または
GENCSRC OBJ('/QSYS.LIB/MYLIB.LIB/MYFILE.FILE')
SRCSTMF('incdir/mymbr.h') SLTFLD(*LVLCHK) TYPEDEFPFX(*NONE)

/* ----------------------------------------------------------------------- */
// PHYSICAL FILE : MYLIB/MYFILE
// FILE LAST CHANGE DATE : 2001/09/13
// RECORD FORMAT : FORMAT1
// FORMAT LEVEL IDENTIFIER : 38A624C5F3B51
/* ----------------------------------------------------------------------- */

_LVLCHK_T mymbr_lvlchk = {
  .
};

注: レベル検査名はソース・ロケーション (ライブラリー、ファイル、メンバー) に依存します。

図7-6. デフォルトの TYPEDEFPFX(*NONE) を使用した SLTFLD(*LVLCHK) の例

SLTFLD(*LVLCHK) が TYPEDEFPFX(prefix_name) と共に指定されている場合、レベル検査構造体の名前は
prefix_name の後に OBJ キーワードおよび SRCFILE/SRCMBR または SRCSTMF キーワードに基づく
ファイル名 (組み込みファイル) が続いたものになります。7-11ページの図7-7のコマンドは、
MYPREFIX_mylib_myfile_mymbr_lvlchk という名前のレベル検査構造体を提供します。
#pragma mapinc ディレクティブを使用したレベル検査

#pragma mapinc ディレクティブは、DDS ファイルを変換してファイルを直接組み込む機能を提供します。

#pragma mapinc ディレクティブで LVLCHK オプションを指定すると、以下が生成されます。

- タイプ _LVLCHK_T の構造体の配列
- タイプ _LVLCHK_T の変数

#pragma mapinc ディレクティブに指定された対応フォーマットのレベル検査情報が各配列要素に含まれるように、配列が初期化されます。最後の 2 つの配列要素は常に空ストリングで、構造体の各フィールドに 1 つずつとなります。

変数の名前は LIBRARY_FILE_INCLUDE_lvlchk です。ここで LIBRARY、FILE、および INCLUDE はそれぞれ library_name、file_name、および include_name です。

_Ropen varparm パラメーターに lvlchk キーワードを指定し、ファイルの構成が変更されると、_Ropen のファイル・ポインタはヌル値を返し、CPF4131 メッセージが表示されます。

注: #pragma mapinc ディレクティブの LVLCHK オプションの使用方法について詳細は、「ILE C/C++ コンパイラー参照」を参照してください。

以下の図は、ファイルがオープンされた際にレベル検査を実行するための #pragma mapinc ディレクティブおよび LVLCHK オプションを示しています。
/* This program illustrates how to use level check information. */
/* This example uses ILE C record I/O. See the ILE C */
/* Programmer's Reference for descriptions of the record I/O */
/* functions. */
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#pragma mapinc("DD3FILE","MYLIB/T1520DD3(purchase)","key lvlchk","_P")
#include "DD3FILE"

int main(void)
{
  _RFILE *in;
  char new_purchase[21] = "PEAR 1002022244";
  /* Open the file for processing in keyed sequence. File is created */
  /* with the default access path. */
  if ( (in = _Ropen("MYLIB/T1520DD3", "rr+ varparm = (lvlchk)",
        &MYLIB_T1520DD3_DD3FILE_lvlchk)) == NULL)
    {
      printf("Open failed\n");
      exit(1);
    }
  /* Update the first record in the keyed sequence. The function */
  /* _Rlocate locks the record. */
  _Rlocate(in, NULL, 0, __FIRST);
  _Rupdate(in, new_purchase, 20);
  /* Force the end of data. */
  _Rfeod(in);
  _Rclose(in);
}

/* ------------------------------------------------------------------------*/
/* PHYSICAL FILE: MYLIB/T1520DD3 */
/* FILE CREATION DATE: 93/09/02 */
/* RECORD FORMAT: PURCHASE */
/* FORMAT LEVEL IDENTIFIER: 322C4B361172D */
/* ------------------------------------------------------------------------*/
typedef _Packed struct {
  char SERIALNUM[10];
} MYLIB_T1520DD3_PURCHASE_key_t;

typedef _Packed struct {
  unsigned char format_name[10];
  unsigned char sequence_no[13];
} _LVLCHK_T[];
_LVLCHK_T MYLIB_T1520DD3_DD3FILE_lvlchk = {
  "PURCHASE " , "322C4B361172D",
  "", "
}

/* 7-10. プログラムからの出カリスト */

7-12  IBM i: ILE C/C++ プログラマーの手引き
C/C++ 構造体におけるフィールド位置合わせの問題の回避

ILE C/C++ 構造体で定義されたすべてのフィールドは、自然境界で位置合わせされます。例えば、int フィールドは、4 パイト長であり、4 パイト境界に保管されます。外部記述ファイルを作成した場合、システムは、外部記述データの境界合わせを制限しません。バックされた構造体は、外部記述データの基準合わせに一致するため、構造体をバックする必要が生じることがあります。

DDS で定義されたフィールドが位置合わせされている (例えば、すべてが文字フィールドである) 場合、構造体をバックせずに生成されたタイプ定義を使用できます。

位置合わせの問題を回避するには、_P オプションを指定して、バックされた構造体を生成します。例えば、ファイル EXAMPLE/CUSTMSTL からのレコード構造 custrec の input および key フィールドのバックされたタイプ定義構造体を組み込むには、以下のステートメントが、以下に示されている順序でプログラム内に記載されている必要があります。

```c
#include "custmf"

#pragma mapinc("custmf","EXAMPLE/CUSTMSTL(custrec)","input key","_P")
...
#include "custmf"
```

プログラムでの外部フィールド定義の組み込み

DDS キーワード INDARA が指定されていない場合、応答標識が組み込まれます。この場合、INPUT、OUTPUT、または BOTH オプションを使用します。

**C++** C++ ユーザーは、GENCSRC ユーティリティを使用して、外部ファイル定義を作成する必要があります。

**C** C ユーザーは、GENCSRC ユーティリティおよび #pragma mapinc ディレクティブのいずれを使用しても外部ファイル定義を作成できます。

注: GENCSRC ユーティリティと #pragma mapinc ディレクティブの違いについて詳しくは、10-1 ベージの「GENCSRC ユーティリティと #pragma mapinc ディレクティブ」を参照してください。

図 7-11 に示されている DDS が ILE C プログラムに組み込まれた場合、7-14 ベージの図 7-12 に示されている構造体定義が生成されます。

```c
#pragma mapinc("test","example/phonelist(phone)","input","")
#include "test"
```

A R PHONE
A CF03(03 'EXIT')
A 1 35 'PHONE BOOK'
A DSPATR(HI)
A 7 28 'Name:'
A NAME 11 A 7 34
A 9 25 'Address:'
A ADDRESS 20 A 9 34
A 11 25 'Phone #:'
A PHONE_NUM 8 A 11 34
A 23 34 'F3 - EXIT'
A DSPATR(HI)

図 7-11. 表示装置ファイルの DDS ソース
INPUT オプション
外部記述装置ファイルで INPUT または BOTH として定義されているフィールドを組み込む際は、INPUT オプションを指定してください。

注: DDS キーワード INDARA が外部ファイル記述に指定されていない場合に限り、オプションおよび応答標識が型定義構造体に組み込まれます。

OUTPUT オプション
外部記述装置ファイルで OUTPUT または BOTH として定義されているフィールドを組み込む際は、OUTPUT オプションを指定してください。

注: DDS キーワード INDARA が外部ファイル記述に指定されていない場合に限り、オプションおよび応答標識が型定義構造体に組み込まれます。

BOTH オプション
BOTH を指定すると、2 つの型定義構造体が生成されます。

・1 つつの型定義には、INPUT または BOTH として定義されたすべてのフィールドが含まれます。もう 1 つつの型定義には、OUTPUT または BOTH として定義されたすべてのフィールドが含まれます。
・すべてのフィールドが BOTH として定義され、分離標識域が指定されていないとき、指定されたフォーマットに対して 1 つの型定義構造体が生成されます。

注: DDS キーワード INDARA が外部ファイル記述に指定されていない場合に限り、オプションおよび応答標識が型定義構造体に組み込まれます。

1 つのレコード形式のみに対する外部ファイル記述を組み込む場合、2 つの型定義を含む型定義共用体が自動的に作成されます。この型定義共用体の名前は LIBRARY_FILE_FMT_both_t です。#pragma mapinc ディレクティブで共用体タイプの名前を指定すると、生成される名前は union-type-name_t です。

C++ ユーザーは型定義に GENCSRC ユーティリティーを使用する必要があります。

C ユーザーは、型定義に GENCSRC ユーティリティーまたは #pragma mapinc ディレクティブのいずれかを使用することができます。

注: GENCSRC ユーティリティーと #pragma mapinc ディレクティブの違いについて詳しくは、10-1 ページの『GENCSRC ユーティリティーと #pragma mapinc ディレクティブ』を参照してください。
図7-13. 装置ファイル用の DDS ソース

上記の DDS が ILE C プログラムに含まれている場合、以下の構造体が生成されます。

```c
#pragma mapinc("example/screen1","example/test(fmt)","both","d")
#include "example/screen1"
```

/* DEVICE FILE: EXAMPLE/Test
/* FILE CREATION DATE: 93/09/01
/* RECORD FORMAT: FMT
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7
/* INDICATORS FOR FORMAT FMT
/* INDICATOR 50
/* INDICATOR 51
/* INDICATOR 99
*/

typedef struct {
  char NAME[11];
} EXAMPLE_TEST_FMT_i_t;
typedef struct {
  char ADDRESS[20];
  char PHONE_NUM[8];
} EXAMPLE_TEST_FMT_o_t;
typedef union {
  EXAMPLE_TEST_FMT_i_t EXAMPLE_TEST_FMT_i;
  EXAMPLE_TEST_FMT_o_t EXAMPLE_TEST_FMT_o;
} EXAMPLE_TEST_FMT_both_t;
```

図7-14. 装置ファイル用の構造体定義

これは、装置ファイル EXAMPLE/TEST のフォーマット FMT がプログラムに含まれている際に作成される構造体定義を示しています。外部ファイル記述には 3 つの標識 IN50、IN51、および IN99、ならびに DDS キーワード INDARA が含まれています。オプション INDICATOR が #pragma mapinc ディレクティブで指定されなかったため、標識はコメントとして表示され、構造体には含まれません。

標識の定義および使用

レコード様式の標識は、装置ファイルでのみ許可され、別個の標識構造体として、またはレコード様式構造体内のメンバーとして定義できます。

C++ ユーザーは、構造体定義に GENCSRC ユーティリティーを使用する必要があります。

C ユーザーは、構造体定義に GENCSRC ユーティリティーおよび #pragma mapinc ディレクティブのいずれも使用できます。
ファイル・バッファーでの標識の作成

DDS でキーワード INDARA を指定しない場合、標識は読み取りまたは書き込みの対象となるファイル・バッファーの一部として作成されます。使用される標識のみが、構造体の型定義内で宣言されます。それらの標識は char として宣言され、#pragma mapinc ディレクティブに INPUT、OUTPUT、または BOTH オプションが指定されている場合に作成されます。

分離標識域の作成

標識を分離した構造体として使用するには、以下の項目を指定する必要があります。

- ファイルの外部記述内の DDS キーワード INDARA
- #pragma mapinc ディレクティブまたは GENCSRC コマンドにおける INDICATORS オプション
- ファイルを開く際の indicators=y の使用

また、入出力操作を実行する前に、レコード・ファイルに _Rindara() 関数を使用して、分離標識域のアドレスも設定する必要があります。

注: #pragma mapinc ディレクティブで標識を指定して外部ファイル記述で DDS キーワード INDARA を使用しないと、コンパイラ時に警告メッセージが表示されます。

標識が要求されていて、フォーマット内に存在する場合、99 バイトの構造体定義が作成されます。この構造体定義には、DDS に定義された各標識のフィールド宣言が含まれます。各フィールドの名前は INXX となります。ここでの XX は DDS の標識番号です。標識間のバイトのシークエンスは INXX_INYY として定義されます。ここで、XX はシークエンス中の最初の未定義バイトで、YY は最後の未定義バイトです。

```
A R FMT       INDARA
A             CF01(50)
A             CF02(51)
A             CF03(99 'EXIT')
A 1 35 'PHONE BOOK'
A DSPATR(HI)
A 7 28 'Name:'
A NAME 11M 1 7 34
A 9 25 'Address:'
A ADDRESS 20A 0 9 34
A 11 25 'Phone #:'
A PHONE_NUM 8A 0 11 34
A 23 34 'F3 - EXIT'
```

図7-15. 標識用の DDS ソース

#pragma mapinc("example/temp","exindic/test(fmt)","indicators",""
#include "example/temp"

この DDS が ILE C/C++ プログラムに含まれている場合、以下の構造体が生成されます。
/* DEVICE FILE: EXINDIC/TEST */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* INDICATORS FOR FORMAT FMT */
/* INDICATOR 50 */
/* INDICATOR 51 */
/* INDICATOR 99 */

typedef struct {
  char IN01_IN49[49]; /* UNUSED INDICATOR(S) */
  char IN50;
  char IN51;
  char IN52_IN98[47]; /* UNUSED INDICATOR(S) */
  char IN99;
} EXINDIC_TEST_FMT_indic_t;

/* INDICATORS FOR RECORD FORMAT FMT */
/* INDICATOR 50 */
/* INDICATOR 51 */
/* INDICATOR 99 */

/* DEVICE FILE: EXINDIC/TEST */
/* CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* INDICATORS FOR RECORD FORMAT FMT */
/* INDICATOR 50 */
/* INDICATOR 51 */
/* INDICATOR 99 */

プログラムでの物理および論理データベース・ファイルの組み込み
外部データベース・ファイル記述を組み込むには、#pragma mapinc ディレクティブで
INPUT、BOTH、KEY、NULLFLDS、または LVLCHK オプションを使用します。OUTPUT または
INDICATOR のいずれかのオプションを指定した場合、エラー・メッセージが生成されます。

INPUT、KEY、または BOTH のいずれかのオプションを指定した場合、データベース・ファイルで説明し
たのと同じ方法を使用して、分散 (DDM) ファイルの外部ファイル記述を組み込むことができます。
OUTPUT または INDICATOR を指定した場合、エラー・メッセージが発行されます。
C++ ユーザーは、レベル検査および外部ファイル記述の組み込みのために GENCSRC ユーティリティを使用する必要があります。

C ユーザーは、レベル検査および外部ファイル記述のために、GENCSRC ユーティリティおよび #pragma mapinc ディレクティブのいずれも使用できます。

注: GENCSRC ユーティリティと #pragma mapinc ディレクティブの違いについて詳しくは、10-1 ページの『GENCSRC ユーティリティと #pragma mapinc ディレクティブ』を参照してください。

プログラムでの装置ファイルの組み込み

外部装置ファイル記述を組み込むには、#pragma mapinc ディレクティブで INPUT、OUTPUT、BOTH、INDICATOR オプションを使用します。装置ファイルには、KEY フィールドは含まれません。そのため、装置ファイルで KEY オプションを指定することはできません。

C ユーザーは、GENCSRC ユーティリティを使用して、外部ファイルからタイプ定義を作成することができます。

C++ ユーザーは、GENCSRC ユーティリティを使用して、外部ファイルからタイプ定義を作成する必要があります。

注: GENCSRC ユーティリティと #pragma mapinc ディレクティブの違いについて詳しくは、10-1 ページの『GENCSRC ユーティリティと #pragma mapinc ディレクティブ』を参照してください。

論理ファイルでの複数の外部記述レコード様式の組み込み

1 つの論理ファイルに複数の様式を組み込むには、#pragma mapinc ディレクティブに、複数のレコード様式名または (*ALL) を指定します。複数の様式を指定した場合、様式ごとにヘッダー記述およびタイプ定義が作成されます。共用体タイプの名前を指定した場合、共用体タイプ定義が作成されます。

C++ ユーザーは、GENCSRC ユーティリティを使用して、外部ファイルからタイプ定義を作成する必要があります。

C ユーザーは、GENCSRC ユーティリティーおよび #pragma mapinc ディレクティブのいずれを使用しても外部ファイルからタイプ定義を作成できます。

注: GENCSRC ユーティリティと #pragma mapinc ディレクティブの違いについて詳しくは、10-1 ページの『GENCSRC ユーティリティと #pragma mapinc ディレクティブ』を参照してください。

タイプ定義共用体には、各様式に作成された構造体定義が含まれています。key オプションを指定すると、キー・フィールド用に作成される構造体定義は、共用体定義に組み込まれません。共用体定義の名前は、union-type-name_t です。union-type-name に指定した名前は、大文字に変換されません。

以下の図では、BOTH および KEY オプションが指定された 2 つのレコード様式を使用して論理ファイル用に作成されたタイプ定義を示します。タグ buffer_t が含まれたタイプ定義共用体も生成されます。

C

#pragma mapinc("pay","lib1/pay(fmt1 fmt2)","both key","","buffer","Pay")
#include "pay"

7-18 IBM i: ILE C/C++ プログラマーの手引き
/* LOGICAL FILE: PAY */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT1 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* --------------------------------------------------------*/
typedef struct {
  
}Pay_FMT1_both_t;

typedef struct {
  
}Pay_FMT1_key_t;

/* LOGICAL FILE: PAY */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT2 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* --------------------------------------------------------*/
typedef struct {
  
}Pay_FMT2_both_t;

typedef struct {
  
}Pay_FMT2_key_t;

typedef union {
  Pay_FMT1_both_t;  Pay_FMT1_both;
  Pay_FMT2_both_t;  Pay_FMT2_both;
}buffer_t;

図7-18. 複数の様式の構造体定義
注: キー・フィールドに対してタイプ定義共用体は作成されません。

format-name パラメーターに *ALL または複数のレコード様式を指定した場合、複数の様式の構造体定義が作成されます。

複数の様式および input または output オプションを指定した場合、様式ごとに 1 つの構造体が作成されます。以下に、プログラムで以下のステートメントを組み込んだ場合に作成される構造体定義を示します。

装置ファイル TESTLIB/FILE には、2 つのレコード様式 FMT1 および FMT2 が含まれています。各レコード様式のファイル記述には、OUTPUT として定義されたフィールドが含まれています。

```
#pragma mapinc("example","testlib/file(fmt1 fmt2)","output","z","unionex")
#include "example"
```
図7-19. 装置ファイル用の構造体定義

オプションとして both が指定された場合、様式ごとに 2 つの構造体定義が作成されます。以下に、装置ファイル EXAMPLE/TEST に対して 2 つの様式 FMT1 および FMT2 を組み込み、both オプションを指定した場合に作成される構造体定義を示します。

```c
#include "test"

#pragma mapinc("test","example/test(fmt1 fmt2)","both","z","unionex")
#include "test"
```

すべてのフィールドが BOTH として定義されていて、typedef struct に標識がない場合、指定した様式ごとに 1 つの typedef struct のみが生成されます。以下に、入力フィールドおよび出力フィールドの別個のタイプ定義構造体を示します。
/* プログラムでの外部記述パック 10 進数データの使用 */

ILE C/C++ コンパイラおよびライブラリでは、パック 10 進数データがサポートされます。

#pragma mapinc ディレクティブの d オプションを使用すると、コンパイラは、DDS ファイルに定義されているすべてのパック 10 進数フィールドに対してパック 10 進数変数を生成します。#pragma mapinc ディレクティブの p オプションまたは GENCSRC コマンドの PKDDECFLD パラメーターを使用した場合、パック 10 進数値を保管するために文字配列が生成されます。

C++ ユーザーは、GENCSRC ユーティリティーを使用して、外部ファイルからタイプ定義を作成する必要があります。

C ユーザーは、GENCSRC ユーティリティーおよび #pragma mapinc ディレクティブのいずれを使用しても外部ファイルからタイプ定義を作成できます。

注: GENCSRC ユーティリティーと #pragma mapinc ディレクティブの違いについて詳しくは、10-1 ページの「GENCSRC ユーティリティーと #pragma mapinc ディレクティブ」を参照してください。
次に、これらの文字配列は、ILE C/C++ プログラムで使用するために ILE C/C++ 数値データ・タイプに変換する必要があります。d オプションも p オプションも指定されていない場合、d オプションがデフォルトとして使用されます。バック 10 進数データ・タイプの使用例については、「8-126 ページの『C プログラムでのバック 10 進数データの使用』または 8-139 ページの『C++ プログラムでのバック 10 進数データの使用』を参照してください。

注: プログラムにバック 10 進数フィールドが定義されている DDS ファイルがある場合、ソース・コードは以下を満たしている必要があります。
- <decimal.h> ヘッダー・ファイルを組み込む
- <bcd.h> ヘッダー・ファイルを組み込む
注: ILE C/C++ コンパイラでコンパイラされるソースに <decimal.h> が組み込まれている場合、<bcd.h> は自動的に組み込まれます。

ILE C/C++ ソース・コードで #pragma mapinc d オプションまたは GENCSRC PKDDECFLD バラメータを使用してください。

注: ILE C/C++ コンパイラおよびライブラリーでは、ゾーン 10 進数データはサポートされません。ゾーン・フィールドがある場合、#pragma mapinc および GENCSRC により、*CHAR タイプに変換されます。

文字列配列に保管されたバック 10 進数データを整数または浮動小数点 (倍精度) に変換するか、その逆の変換を行うために、関数 QXXDTP()、QXXITOP()、QXXPT0D()、QXXPT0I() を使用できます。

文字配列に保管されたゾーン 10 進数データを整数または浮動小数点 (倍精度) に変換するか、その逆の変換を行うために、関数 QXXDTOZ()、QXXIT0Z()、QXXZT0D()、および QXXZTOI() を使用できます。

MI cpynv() 関数を使用して、バックまたはゾーン 10 進数データを ILE C/C++ 数値データ・タイプに変換することもできます。これは、ILE C/C++ 数値データ・タイプをバックまたはゾーン 10 進数データに変換するために使用できます。

変換関数は ILE C/C++ コンパイラに組み込まれているため、これらの関数を使用している EPM C コードを保存できます。

データベース入出力操作を実行する場合、整数または浮動小数点フィールドが含まれた論理ファイルを使用して、物理ファイル内のバックおよびゾーン・フィールドを再定義できます。論理ファイルを介して入力または出力操作を実行する際に、IBM i はデータを自動的に変換します。

### プログラムでのデータベース・ファイルおよび分散ファイルの使用
このトピックでは、以下を行うための方法について説明します。
- 到着順アクセス・バスを使用して、あるファイルから別のファイルにデータをコピーする
- キー・シーケンス・アクセス・バスを使用して、レコード・ファイル内のデータを更新する
- データ・ファイルからレコードを読み取って印刷する
- コミットメント制御条件を指定する

### データベース・ファイルおよび分散ファイル
データベース・ファイルには、システムに永続的に保管されるデータが含まれています。オブジェクト・タイプは *FILE です。
データベース・ファイルは、物理ファイルとしても論理ファイルとしても作成および使用できます。データベース・ファイルには、データまたはソース・ステートメントのいずれかを含めることができます。

ILE C/C++ プログラムは、分散 (DDM) を介してリモート・システム上のファイルにアクセスします。DDM によって、システム上のアプリケーション・プログラムが、リモート・システムに保管されているファイルをデータベース・ファイルとして使用することができます。DDM ファイルをサポートするために ILE C/C++ プログラム内で特別なステートメントは必要ありません。

DDM ファイルは、ローカル（ソース）システム上でユーザーまたはプログラムによって作成されます。このファイル（オブジェクト・タイプが *FILE）は、リモート（ターゲット）システム上に保有されているファイルを指定します。ローカル IBM i がリモート IBM i を見つけたり、ターゲット・ファイルのデータにアクセスするのに必要な情報が、DDM ファイルによって提供されます。

物理ファイルおよび論理ファイル
物理ファイルには、IBM i 上に保管される実際のデータ、およびプログラムへのデータの表のされ方がまたはプログラムからのデータの受け取り方の記述が入っています。これには、1 つのレコードタイプが入っており、また 1 つ以上のメンバーが入っています。

物理ファイルにはキー順アクセス・パスを使用することができます。このことは、データはファイルの中の 1 つ以上の複数のキー・フィールドに基づいて順番に ILE C/C++ プログラムに与えられる、ということを意味します。

論理ファイルにはデータは入っていません。このファイルには、1 つまたは複数の物理ファイルで見つかるレコードの記述が入っています。論理ファイルは、1 つまたは複数の物理ファイルのビューまたは表現です。複数の様式が入っている論理ファイルは、複数様式論理ファイルとして参照されます。

プログラムが複数のレコードタイプの入っている論理ファイルを処理する場合には、_Rformat() 関数を用いて使用したい様式を設定することができます。

論理ファイルに対しては、一部の操作は実行できません。オープン・モード w、w+、wb、または wb+ でストリーム・ファイル処理用に論理ファイルを開いた場合、ファイルは開かれますが、消去されていません。オープン・モード wr または wr+ でレコード・ファイル処理用に論理ファイルを開いた場合、ファイルは開かれますが、消去されません。

データベース・ファイル内のレコードの記述
データベース・ファイル内のレコードは、フィールド・レベル記述またはレコード・レベル記述のいずれかを使用して記述することができます。

フィールド・レベル記述には、すべてのフィールドについての記述と、このレコード内でのそれらの配列についての記述が含まれます。フィールドおよびその配列についての記述は ILE C/C++ プログラム内ではなくデータベース・ファイル内に保持されるため、フィールド・レベル記述によって作成されたデータベース・ファイルは外部記述ファイルと呼ばれています。7-1 ページの『プログラムでの外部記述ファイルの使用』を参照してください。

レコード・レベル記述は、レコードのコンテンツではなく、レコードの長さのみを記述します。レコード・レベル記述によって作成されたデータベース・ファイルはプログラム記述ファイルと呼ばれています。これは、ILE C/C++ プログラムでレコードのフィールド記述を記述しなければならないことを意味します。

外部記述ファイルの定義： ILE C/C++ プログラムは外部記述ファイルとプログラム記述ファイルのいずれかを使用することができます。外部記述ファイルを使用する場合、ILE C/C++ コンパイラは外部記述フ
ファイルから情報を取り出し、自動的にフィールド情報をプログラムに組み込むことができます。プログラム側でフィールド情報を定義する必要はありません。詳しくは、7-1ページの『プログラムでの外部記述ファイルの使用』を参照してください。

外部記述データベース・ファイルを定義するには、以下のいずれかを使用します。

* DB2® for IBM i
* 対話式データ定義ユーティリティー (IDDU)
* データ記述仕様 (DDS) ソース

データ記述仕様は、定形様式でシステムに入力されるデータベース・ファイルの記述で、ファイルを作成する際に使用されます。この記述は、レコードを構成するフィールドを定義する 1 つ以上のレコード様式からなります。また、ファイルからレコードを取り出す順序を決定するアクセス・パス情報が含まれる場合もあります。

**データ・ファイルとソース・ファイル**
データ・ファイルには実際のデータが含まれます。

データ・ファイルのレコードはメンバーにグループ化されます。ファイルのすべてのレコードを 1 つのメンバーに入れるか、あるいは異なるメンバーにグループ化することができます。ほとんどのデータベース・コマンドおよび命令は、そのデフォルトではデータを含むデータベース・ファイルにはメンバーを 1 つだけもっているものとします。これは、ILE C/C++ プログラムがデータを含むデータベース・ファイルを処理する際、ファイルに複数のメンバーが入っていない限り、ファイルのメンバー名を指定する必要がないことを意味します。

通常、ソース・プログラムが入っているデータベース・ファイルは、複数のメンバーから構成されています。ソース・プログラムをデータベース・ファイル内のメンバーに編成することによって、プログラムをより適切に管理することができます。これらのソース・メンバーには、IBM i が IBM i オブジェクトを作成する際に使用するソース・ステートメントが含まれます。例えば、C++ ステートメントを含むソース・メンバーを使用して、プログラム・オブジェクトが作成されます。

**アクセス・パス**
アクセス・パス は、ファイル内のレコードの論理順序を記述したもので、アクセス・パスには2つのタイプ（到着順とキー・シーケンス）があります。

到着順アクセス・パスを使用して取得されたレコードは、ファイルに追加されたのと同じ順序で取得されます。これは、順次ファイルの処理と似ています。新規レコードは、物理的にファイルの末尾に保管されます。到着順アクセス・パスは、物理ファイルと論理ファイルの両方で有効です。

キー・シーケンス・アクセス・パスを使用して取得されたレコードは、レコード内の 1 つ以上のキー・フィールドの内容に基づいて取得されます。これ、他のシステムにおける索引付きファイルまたはキー付きファイルの処理に似ています。キー・シーケンス・アクセス・パスは、レコードが追加、削除、または更新されるたび、またキー・フィールドの内容が変更されると、更新されます。このアクセス・パスは、物理ファイルと論理ファイルの両方で有効です。

ファイルに複数のレコード様式が定義されている場合、各レコード様式に異なるキー・フィールドを含めることができます。ファイルのデフォルトキー（例えば、様式が指定されていない場合）は、すべてのレコード様式に共通したキー・フィールドになります。デフォルトキーがない（例えば、共通するキー・フィールドがない）場合、様式を指定していない入力操作に対して、ファイル内の最初のレコードが常に返されます。
注：ILE C/C++プログラムがファイルを開くと、デフォルトでは、ファイルの作成に使用されたアクセス・パスを使用してファイルが処理されます。arrseq=N（デフォルト）を指定した場合、ファイルは、作成された方法で処理されます。つまり、ファイルがキー・シーケンス・アクセス・パスを使用して作成された場合、ILE C/C++プログラムはキー・シーケンス・アクセス・パスを使用してファイルを処理します。arrseq=Y を指定した場合、ファイルは、到着順を使用して処理されます。つまり、ファイルがキー・シーケンス・アクセス・パスを使用して作成された場合でも、ILE C/C++プログラムは、到着順アクセス・パスを使用してファイルを処理します。

キー・フィールドの配列
キー・シーケンス・アクセス・パスは、昇順または降順で順序付けることができます。キー・フィールドを記述する際には、昇順がデフォルトです。データ記述仕様（DDS）を使用してキー・シーケンス・ファイルを作成する場合、DESCEND DDSキーーワードを使用することで、キー・フィールドを降順で配列するように指定することができます。

重複キー値
レコードに含まれるキー・フィールドのコンテンツが同じファイル内の別のレコードのキー・フィールドと同じである場合、そのファイルには重複キー値を持つレコードが含まれています。例えば、レコードにnum と date の 2 つのキー・フィールドがある場合、num と date の両方のコンテンツが複数のレコードにおいて同一のときに、重複キー値が発生します。

プログラムが重複キー値を含むレコードを処理中であることが表示されるようにするには、ファイルをオープンする_Ropen() の呼び出し時に dupkey=y を指定してください。レコードに対する入出力操作が正常に行われ、重複キー値がそのレコード内で検出されると、_RIOFB_T構造体のdup_keyフラグが設定されます。（_Rreadd()関数はこのフラグを更新しません。）

注：_Ropen()関数の呼び出し時に dupkey=yオプションを使用すると、入出力操作の速度が遅くなる可能性があります。

DDSファイルでキーワードUNIQUEを指定することで、重複キー値を避けることができます。

削除済みレコード
データベース・レコードが削除されると、該当する物理レコードに削除済みのマークが付けられますが、物理レコード自体はファイル内に残ります。削除済みレコードは、_Rwrite()関数を使用して上書きすることができます。RGZPFM（物理ファイル・メンバーの再編成）コマンドを使用すると、削除済みレコードをファイルから除去することができます。また、これらのレコードは、CRTPFコマンドでREUSEDLT(*YES)パラメーターを指定することで、書き込み操作時に再使用することが可能です。

削除済みレコードは、物理ファイル・メンバー初期設定(INZPFM)コマンドを使用して削除済みレコードのあるファイルを初期設定した場合に、そのファイルで発生することがあります。レコードが削除されると、そのレコードを読み取ることはできません。

ロック
IBM iデータベースにはデータの保全性が組み込まれています。システムは、ILE C/C++プログラムがファイルを開く方法に基づいたロック状態を判別します。この表は、有効なオープン・モードおよびそれに関連付けられたロック状態を示しています。
表7.2 各オープン・モードに対するロック状態

<table>
<thead>
<tr>
<th>オープン・モード</th>
<th>ロック状態</th>
</tr>
</thead>
<tbody>
<tr>
<td>r または rb</td>
<td>読み取り共用 (*SHRRD)</td>
</tr>
<tr>
<td>a, w, ab, wb, a+, r+, w+, ab+, rb+, wb+</td>
<td>更新共用 (*SHRUPD)</td>
</tr>
</tbody>
</table>

ファイルをオープンする前にデータベース・ファイル一時変更 (OVRDBF) コマンドまたはオブジェクト割り振り (ALCOBJ) コマンドを使用することで、ファイルのロック状態を変更することができます。例えば、ILE C プログラムは下記のように system() 関数を使用して ALCOBJ コマンドを呼び出すことができます。

```c
system("ALCOBJ OBJ((FILE *FILE *EXCLRD));");
```

ファイルが更新にオープンしている場合、_NO_LOCK オプションが指定されていなければ、データベースは読み取り中または位置指定中のレコードをすべてロックします。これは、別のプログラムによってオープンされたものか、もしくは別のファイル・ポインタを使用して同じプログラムによってオープンされたものかに関わらず、他のオープン・データ・パスに対してロック済みのレコードをロックすることはできないということです。

別のレコードの読み取りおよびロックが正常に行われると、それ以前にロックされたレコードのロックが解除されます。読み取りに対して _NO_LOCK オプションが指定されている場合には、以前ロックされたレコードに対するロックは解除されません。また、_Rrlslck() 関数を使用してレコードに対するロックを解除することも可能です。

共用

アプリケーションが C および C++ モジュールのみで構成されている場合、ファイルを共用するための望ましい方法は、1つのプログラムでファイルをオープンし、そのファイル・ポインターを他のプログラムへ渡すことです。こうすることで、何度もファイルを開く必要がなくなります。

同じジョブ内でファイルを共用することによって、そのジョブ中のプログラムはそのファイルの状況、レコード位置、およびバッファーを共用することができます。オープン・データ・パス (ODP) を同じジョブ内で実行される複数のプログラム間で共用できるようにするには、ファイルの作成、変更、または指定変更を行うコマンドで SHARE(*YES) バラメータを使用します。オープン・データ・パスとは、ファイルに対するすべての入出力操作に使用されるパスのことです。

レコード単位で処理されるストリームのオープン・データ・パスは、共用することができます。レコード・ファイルのオープン・データ・パスも共用可能です。文字単位で処理されるストリームのオープン・データ・パスは、入出力操作の実行時に予測不能な結果になる可能性があるため、共用しないでください。

C/C++ プログラムと他の言語で作成されたプログラムの間でのファイル共用は、オープン・データ・パスを共用することで行うことができます。

共用ファイルの最初のオープンによって、そのファイルのオープン・モード (例えば、入力、出力、更新、および削除のいずれのためにオープンしているか) が決まります。その後のオープンによって、最初のオープンで指定しなかったオープン・モードが指定された場合、ファイルは 2 回目にオープンされます。そのオープン・モードは無視されます。例えば、最初のオープンで 10 のオープン・モードを指定し、2 回目のオープンで IOUD を指定した場合、ファイルは 2 回目に 10 のモードでオープンします。
ヌル可能フィールド

ILE C コンパイラを使用すると、ヌル値とみなされるフィールドが含まれたレコードを持つファイルを処理することができます。_Ropen() 関数に nullcap=Y を指定する必要があります。ヌル可能フィールドがヌル値に設定されると、そのフィールドに書き込まれたデータはすべて無効になります。

nullcap=Yでファイルがオープンされると、入出力ヌル・マップと、そのファイルがキー付きである場合はキー・ヌル・マップが、データベースにより提供されます。入出力ヌル・マップは、ファイルの現行レコード様式での、各フィールドに対する1バイトで構成されています。NULL フィールド・マップは、データベースとプログラマの間で通信し、具体的にどのフィールドがヌル値と考えられるかを示すために使用されます。

<recio.h> ファイルで定義された _RFILE 構造体には、入力、出力、およびキー NULL フィールド・マップへのポインター、ならびにこれらのマップの長さ(null_map_len および null_key_map_len)が含まれます。

データベース・ファイルに対して書き込みを行う際、どのフィールドがヌル値であるかを文字 ‘1’ で指定します。フィールドが非ヌルである場合は、文字 ‘0’ を指定します。これは、out_null_map ポインターが指す NULL フィールド・マップに指定されます。ヌル可能フィールドが含まれていないファイルであるのに nullcap=Yでオープンされの場合、プログラムで NULL フィールド・マップの各フィールドを文字 ‘0’ に設定する必要があります。この設定は、ファイルにデータを書き込む前に行う必要があります。

データベース・ファイルから読み取りを行う際、フィールドがヌル値とみなされる場合は、NULL フィールド・マップの対応するバイトが文字 ‘1’ で示されます。これは、in_null_map ポインターが指す NULL フィールド・マップに指定されます。

NULL キー・フィールド・マップは、現行レコード様式のキーの各フィールドに対する1バイトで構成されています。NULL フィールドを持つキーによりデータベース・ファイルを読み取る場合、まず null_key_map が指す NULL キー・マップで、どのフィールドにヌル値が含まれるかを指示する必要があります。ヌル値とみなされるフィールドに対しては文字 ‘1’ を、その他のフィールドに対しては文字 ‘0’ を指定してください。

NULL フィールドの処理を可能にするためにオープンされたファイルの更新用に _Rupdate() 関数が呼び出される際、システム入力バッファが使用されます。その結果、データベースでは、in_null_map ポインターを使用して入力 NULL フィールド・マップを指定することが要求されます。_Rupdate() を呼び出す前に、ユーザーは入力 NULL フィールド・マップをいったんクリアし、レコードの更新に使用されるデータに基づいて (in_null_map ポインターを使用して) このマップを設定する必要があります。

#pragma mapinc ディレクティブを使用して、NULL フィールド・マップに対応する型定義を生成することができます。_RFILE 構造体内にある NULL フィールド・マップ・ポインターをこれらの型にキャストし、これらのマップを操作することができます。また、プログラム内で NULL フィールド・マップをクリアして設定する作業を支援するため、<recio.h> ファイル内で NULL フィールド・マクロも提供されています。

データベース・ファイルおよび DDM ファイルをレコード・ファイルとしてオープンする

IBM i のデータベース・ファイルをレコード・ファイルとしてオープンするには、_Ropen() 関数を以下のいずれかのモードで使用してください。

- rr または rr+
- wr または wr+
データベース・ファイルおよび DDM ファイルの有効なキーワード・パラメーターは以下のとおりです。

- arrseq
- blkrcd
- commit
- ccsid
- dupkey
- riofb
- secure
- varparm
- vlr
- rtncode

データベース・ファイルおよび DDM ファイルのレコード関数
以下のレコード関数を使用して、データベース・ファイルおよび DDM ファイルを処理します。

- _Rclose()
- _Rcommit()
- _Rdelete()
- _Rfeod()
- _Rformat() (複数フォーマットの論理ファイル)
- _Riofbk()
- _Rlocate()
- _Ropen()
- _Ropnfbk()
- _Rreadd()
- _Rreadf()
- _Rreadk()
- _Rreadl()
- _Rreadn()
- _Rreadp()
- _Rrlslck()
- _Rrollbck()
- _Rupfb()
- _Rupfb()
- _Rupdate()
- _Rwrite()
- _Rwrited()
DDM ファイルの入出力に関する考慮事項

DDM ファイルは、プログラム記述ファイルとしてアクセスすることが可能です (CRTDDMF コマンドの RMTFILE パラメーターでリモート・ファイル名を指定してください)。また、外部記述ファイルとしてアクセスすることもできます (CRTDDMF コマンドの RMTFILE パラメーターでリモート DDS ファイル名を指定してください)。

データベース・ファイルおよび DDM ファイルをバイナリー・ストリーム・ファイルとしてオープンする

IBM i のデータベース・ファイルをレコード単位の処理用にバイナリー・ストリーム・ファイルとしてオープンするには、fopen() 関数を以下のいずれかのモードで使用してください。

・ rb
・ wb
・ ab

データベース・ファイルおよび DDM ファイルの有効なキーワード・パラメーターは以下のとおりです。

・ blksize
・ lrecl
・ type
・ commit
・ arrseq
・ ccsid

データベース・ファイルまたは DDM ファイルを指定する場合、パラメーター・タイプは record でなければなりません。

注: データベース・ファイルが存在しない場合 (すなわちオープン・モードが wb または ab の場合) に作成される物理データベース・ファイルは、以下の CL コマンドを指定することと同等です。

```
CRTPF FILE(filename) RCDLEN(lrecl)
```

このファイル内でのレコードは、キーワード・パラメーター lrecl に基づくレコード長で作成されます。

DDM ファイルを作成する唯一の方法は、DDM ファイル作成 (CRTDDMF) コマンドを使用することです。 fopen() 関数を wb または ab のモードで使用する場合で、かつ DDM ファイルが起動システム上に存在するがデータベース・ファイルはリモート・システム上に存在しない場合、物理データベース・ファイルがリモート・システム上に作成されます。DDM ファイルが起動システム上に存在しない場合、物理データベース・ファイルが起動システム上に作成されます。

バイナリー・ストリーム・データベース・ファイルおよび DDM ファイルの入出力に関する考慮事項

データベース・ファイルに削除済みレコードが含まれている場合、削除済みレコードはすべてのバイナリー・ストリーム入出力関数によってスキップされます。

バイナリー・ストリームのレコード単位のファイルは、キーでは処理できません。また、それらは rb、wb、および ab モードでしかオープンすることができません。
データベース・ファイルおよび DDM ファイルのバイナリー・ストリーム関数
データベース・ファイルおよび DDM ファイルをレコード単位で処理するには、以下のいずれかのバイナリー・ストリーム関数を使用してください。

- fclose()
- fopen()
- fread()
- freopen()
- fwrite()

データベース・レコード・ファイルを到着順に処理する
到着順アクセス・パスを使用すると、1 つのファイルから別のファイルにデータをコピーすることができます。レコードはファイル内で、ファイルに追加されたときと同じ順序でアクセスされます。

手順
以下の例では、ファイル T1520ASI に追加された時と同じ順序で、データが入力ファイル T1520ASI から出力ファイル T1520ASO にコピーされます。_Rreadn() および _Rwrite() 関数が使用されます。

1. 入力ファイルを作成するには、次のように入力します。
   ```
   CRTPF FILE(MYLIB/T1520ASI) RCDLEN(300)
   ```
   これによって、物理ファイル T1520ASI が作成されます。

2. T1520ASI に以下のサンプル・データを入力します。
   ```
   joe 5
   fred 6
   wilma 7
   ```

3. 出力ファイルを作成するには、次のように入力します。
   ```
   CRTPF FILE(MYLIB/T1520ASO) RCDLEN(300)
   ```
   これによって、物理ファイル T1520ASO が作成されます。

4. プログラムを作成するには、次のように入力します。
   ```
   CRTBNDC PGM(MYLIB/T1520ASP) SCRFILE(QCPPLE/QACSRC)
   ```
   これにより、7-31 ページの図 7-21 のソース・コードを使用したプログラム T1520ASP が作成されま

5. プログラムを実行するには、次のように入力してください。
   ```
   CALL PGM(MYLIB/T1520ASP)
   ```
   物理ファイル T1520ASO には、次のデータが含まれています。
   ```
   joe 5
   fred 6
   wilma 7
   ```

ソース・コードのサンプル
/* This program illustrates how to copy records from one file to another file, using the _Rreadn(), and _Rwrite() functions. */

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

#define _RCDLEN 300

int main(void)
{
    _RFILE *in;
    _RFILE *out;
    _RIOFB_T *fb;
    char record[_RCDLEN];

    /* Open the input file for processing in arrival sequence. */
    if ( (in = _Ropen("*LIBL/T1520ASI", "rr, arrseq=Y")) == NULL )
    {
        printf("Open failed for input file\n");
        exit(1);
    }

    /* Open the output file. */
    if ( (out = _Ropen("*LIBL/T1520ASO", "wr")) == NULL )
    {
        printf("Open failed for output file\n");
        exit(2);
    }

    /* Copy the file until the end-of-file condition occurs. */
    fb = _Rreadn(in, record, _RCDLEN, __DFT);
    while( fb->num_bytes != EOF )
    {
        _Rwrite(out, record, _RCDLEN);
        fb = _Rreadn(in, record, _RCDLEN, __DFT);
    }

    _Rclose(in);
    _Rclose(out);
}

図7-21 T1520ASP — データベース・レコード・ファイルを到着順に処理するための ILE C ソース

注:
1. このプログラムは、_Ropen() 関数を使用して入力ファイル T1520ASI をオープンし、追加された時と同じ順序でレコードにアクセスします。
2. _Ropen() 関数は出力ファイル T1520ASO もオープンします。
3. _Rread() 関数はファイル T1520ASI 内のレコードを読み取ります。
4. _Rwrite() 関数はそれらのレコードをファイル T1520ASO に書き込みます。

データベース・レコード・ファイルをキー・シーケンスで処理する
キー・シーケンス・アクセス・パスを使用して、レコード・ファイルを更新することができます。レコードは、そのレコード内の1つ以上のキー・フィールドのコンテンツに基づいて配列されます。

例:
以下の例では、キー・フィールド SERIALNUM を使用して、レコード・ファイル T1520DD3 内のデータを更新します。_Rupdate() 関数が使用されます。
1. コマンド行に、次のように入力します。

```plaintext
CRTPF FILE(MYLIB/T1520DD3) SRCFILE(QCPPLE/QADDSSRC)
```

以下の DDS ソースを使用する物理ファイル T1520DD3 を作成するには、次のようにします。

```plaintext
A    R PURCHASE
A    ITEMNAME    10
A    SERIALNUM    10
A    K SERIALNUM
```

図7-22. T1520DD3 — データベース・レコードの DDS ソース

2. T1520DD3 に以下のサンプル・データを入力します。

```plaintext
orange 1000222200
grape  1000222010
apple  1000222030
cherry 1000222020
```

表示されたとおりにデータを入力しても、ファイル T1520DD3 はプログラム T1520KSP によってキー・シーケンスでアクセスされます。このため、プログラム T1520KSP がファイル T1520DD3 を読み取る順序は次のようにになります。

```plaintext
grape  1000222010
cherry 1000222020
apple  1000222030
orange 1000222200
```

3. コマンド行に、次のように入力します。

```plaintext
CRTBNDC PGM(MYLIB/T1520KSP) SRCFILE(QCPPLE/QACSRC)
```

これにより、以下のソースを使用してプログラム T1520KSP が作成されます。
/* This program illustrates how to update a record in a file using */
/* the _Rupdate() function. */

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE *in;
    char new_purchase[21] = "PEAR 1002022244";

    if ( (in = _Ropen("*LIBL/T1520DD3", "rr+")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    }

    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);
    _Rfeod(in);
    _Rclose(in);
}

CALL PGM(MYLIB/T1520KSP)

grape 1000222200
PEAR 1002022244
apple 1000222030
cherry 1000222020

レコード入出力関数を使用してデータベース・レコード・ファイルを処理する
データ・ファイルからレコードを読み取り、出力することができます。
例:
以下の例では、レコード入出力関数 _Ropen()、_Rreadl()、_Rreadp()、_Reads()、_Rreadd()、_Rreadf()、_Rrlslck()、_Rdelete()、_Ropnfbk()、および_Rclose() が使用されています。プログラム T1520REC は、データ・ファイル T1520DD4 からレコードを読み取り、出力します。

1. コマンド行に、次のように入力します。
   
   CRTPF FILE(MYLIB/T1520DD4) SRCFILE(QCPPLE/QADDSSRC)

   これにより、以下の DDS を使用する物理ファイル T1520DD4 が作成されます。

   A      R PURCHASE
   A  ITEMNAME    10
   A  SERIALNUM   10
   A      K SERIALNUM

   図7-24. T1520DD4 — データベース・レコードの DDS ソース

2. T1520DD4 に以下のサンプル・データを入力します。

   orange  1000222200
   grape   1000222010
   apple   1000222030
   cherry  1000222020

3. コマンド行に、次のように入力します。
   
   CRTBNDC PGM(MYLIB/T1520REC) SRCFILE(QCPPLE/QACSRC).

   これにより、以下のソースを使用するプログラム T1520REC が作成されます。
/* This program illustrates how to use the _Rreadp(), _Rreads(), _Rreadd(), _Rreadf(), _Rreadn(), _Ropnfbk(), _Rdelete, and _Rrlslck() functions. */

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    char buf[21];
    _RFILE *fp;
    _XXOPFB_T *opfb;
    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "*LIBL/T1520DD4", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed¥n" );
        exit ( 1 );
    }
    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s¥nFile: %10.10s¥n",
            opfb->library_name,
            opfb->file_name);
    /* Get the last record. */
    _Rreadl ( fp, NULL, 20, _DFT );
    printf ( "Fourth record: %10.10s
", *(fp->in_buf) );
    /* Get the third record. */
    _Rreadp ( fp, NULL, 20, _DFT );
    printf ( "Third record: %10.10s
", *(fp->in_buf) );
    /* Release lock on the record so another function can access it. */
    _Rrlslck ( fp );
    /* Read the same record. */
    _Rreads ( fp, NULL, 20, _DFT );
    printf ( "Same record: %10.10s
", *(fp->in_buf) );
    /* Get the second record without locking it. */
    _Rreadd ( fp, NULL, 20, _NO_LOCK, 2 );
    printf ( "Second record: %10.10s
", *(fp->in_buf) );
    /* Get the first record. */
    _Rreadf ( fp, NULL, 20, _DFT );
    printf ( "First record: %10.10s
", *(fp->in_buf) );
    /* Delete the second record. */
    _Rreadn ( fp, NULL, 20, _DFT );
    _Rdelete ( fp );
    /* Read all records after deletion. */
    _Rreadf ( fp, NULL, 20, _DFT );
    printf ( "First record after deletion: %10.10s
", *(fp->in_buf) );
    _Rreadn ( fp, NULL, 20, _DFT );
    printf ( "Second record after deletion: %10.10s
", *(fp->in_buf) );
    _Rreadd ( fp, NULL, 20, _DFT );
    printf ( "Third record after deletion: %10.10s
", *(fp->in_buf) );
    _Rclose ( fp );
}

図 7.25. T1520REC — レコード入出力関数を使用してデータベース・ファイルを処理するための ILE C ソース

_Ropen() 関数はファイル T1520DD4 をオープンします。_Ropnfbk() 関数は、ライブラリ名 MYLIB とファイル名 T1520DD4 を取得します。_Rreadl() 関数は 4 番目のレコード "cherry 1000222020" を読み取ります。_Rreadp() 関数は 3 番目のレコード "apple 1000222030" を読み取ります。_Rrlslck() 関数はこのレコードのロックを解除し、_Rreads() がこのレコードを再度読み取れるようにします。_Rreadd() 関数は 2 番目のレコード "grape 1000222010" をロックせずに読み取ります。_Rreadf() 関
4. プログラム T1520REC を実行します。コマンド行に、次のように入力します。

CALL PGM(MYLIB/T1520REC)

画面の出力は以下のようになります。

<table>
<thead>
<tr>
<th>Library: MYLIB</th>
</tr>
</thead>
<tbody>
<tr>
<td>File: T1520DD4</td>
</tr>
<tr>
<td>Fourth record: cherry</td>
</tr>
<tr>
<td>Third record: apple</td>
</tr>
<tr>
<td>Same record: apple</td>
</tr>
<tr>
<td>Second record: grape</td>
</tr>
<tr>
<td>First record: orange</td>
</tr>
<tr>
<td>First record after deletion: orange</td>
</tr>
<tr>
<td>Second record after deletion: apple</td>
</tr>
<tr>
<td>Third record after deletion: cherry</td>
</tr>
</tbody>
</table>

実行キーを押して終了セッションを終了してください。

物理ファイル T1520DD4 には、次のデータが含まれます。

<table>
<thead>
<tr>
<th>初期データ</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORANGE 1000222200</td>
</tr>
<tr>
<td>APPLE 1000222030</td>
</tr>
<tr>
<td>CHERRY 1000222020</td>
</tr>
</tbody>
</table>

単一ジョブでデータベース・ファイルの変更を同期する

コミットメント制御は、複数のファイル操作を 1 つの単位としてグループ化し、データベース・ファイルに対する変更を单一のジョブで同期できるようにするための手段です。

コミットメント制御を開始する前に、1 つの単位として処理したいデータベース・ファイルがすべて单一のコミットメント制御環境に入っていることを確認してください。この環境内にあるすべてのファイルが、同じジャーナルにジャーナル処理されなければなりません。ジャーナル・レシーバ作成 (CRTJRCV)、ジャーナル作成 (CRTJRN)、および物理ファイルのジャーナル処理開始 (STRJRNPF) の各 CL コマンドを使用して、ジャーナル処理環境の準備をしてください。

ジャーナル処理環境が確立すると、以下のコマンドを使用できます。

- コミットメント制御開始 (STRCMTCNTL)
- CALL program-name
- コミットメント制御終了 (ENDCMTCNTL)

コミットメント制御を使用して、データベース・ファイルに対する複数の変更点の定義および処理を、単一トランザクションとして行うことができます。

例:

次の例では、コミットメント制御を使用しています。購入オーダーは、日次トランザクション用には T1520DD5、月次トランザクション用には T1520DD6 の 2 つのファイルに入力され、ログに記録されます。T1520DD5 および T1520DD6 に対して行われた変更を反映するジャーナル項目は、ジャーナル JRN に保持されます。

1. ジャーナル処理環境を準備します。
   a. コマンド行に、次のように入力します。

      CRTPF FILE(QTEMP/T1520DD5) SRCFILE(QCPPLE/QADDSSRC)
これによって、以下に示す DDS ソースを使用して物理ファイル T1520DD5 が作成されます。

```
A R PURCHASE
A ITEMNAME 30
A SERIALNUM 10
```

図 7-26: T1520DD5 — 日次トランザクション用 DDS ソース

b. コマンド行で次のように入力します。

```
CRTPF FILE(QTEMP/T1520DD6) SRCFILE(QCPPLE/QADDSSRC)
```

これによって、以下に示す DDS ソースを使用して物理ファイル T1520DD6 が作成されます。

```
A R PURCHASE
A ITEMNAME 30
A SERIALNUM 10
```

図 7-27: T1520DD6 — 月次トランザクション用 DDS ソース

c. コマンド行に、次のように入力します。

```
CRTPF FILE(MYLIB/NFTOBJ) RCDLEN(19)
```

これによって、通知テキスト用の物理ファイル NFTOBJ が作成されます。

注: コミットメント制御を使用する ILE C プログラム T1520COM が実行される際に、通知テキストがファイル NFTOBJ に送られます。

d. コマンド行に、次のように入力します。

```
CRTDSPF FILE(QTEMP/T1520DD7) SRCFILE(QCPPLE/QADDSSRC)
```

これによって、以下に示す DDS ソースを使用して画面ファイル T1520DD7 が作成されます。

```
A DSPSIZ(24 80 +DS3)
A REF(QTEMP/T1520DD5)
A INDARA
A CF03(03 'EXIT ORDER ENTRY')
A R PURCHASE
A 3 32 'PURCHASE ORDER FORM'
A DSPATR(UL)
A DSPATR(HI)
A 10 20 'ITEM NAME :'
A DSPATR(HI)
A 12 20 'SERIAL NUMBER :'
A DSPATR(HI)
A ITEMNAME R I 10 37
A SERIALNUM R I 12 37
A 23 34 'F3 - Exit'
A DSPATR(HI)
A R ERROR
A 6 28 'ERROR: Write failed'
A DSPATR(BL)
A DSPATR(UL)
A DSPATR(HI)
A 10 26 'Purchase order entry ended'
```

図 7-28: T1520DD7 — 購入オーダー画面用の DDS ソース
e. コマンド行に、次のように入力します。

```
CRTJNRCV JNRCV(MYLIB/JNRCV)
```
これによって、ジャーナル・レシーバー JRNRCV が作成されます。

注: ジャーナル項目は、アプリケーションの実行時に JRNRCV に保管されます。

f. コマンド行に、次のように入力します。
CRTJRN JRN(MYLIB/JRN) JRNRCV(MYLIB/JRNRCV)

これによってジャーナル JRN が作成され、それにジャーナル・レシーバー JRNRCV が接続されます。

g. コマンド行に、次のように入力します。
STPRNPF FILE(QTEMP/T1520DD5 QTEMP/T1520DD6) JRN(MYLIB/JRN)
 IMAGES(*BOTH)

これにより、T1520DD5 および T1520DD6 に対して行われた変更のジャーナル処理がジャーナル JRN で開始されます。

h. コマンド行に、次のように入力します。
CRTBNDC PGM(MYLIB/T1520COM) SRCFILE(QCPPLE/QACSRC)

これにより、以下に示すプログラム・ソースを使用してプログラム T1520COM が作成されます。

/* This program illustrates how to use commitment control using the */
/* _Rcommit() function and to rollback a transaction using the */
/* _Rollback() function. */

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#define PF03 2
#define IND_OFF '0'
#define IND_ON '1'
int main(void)
{
  char buf[40];
  int rc = 1;
  _SYSindara ind_area;
  _RFILE *purf;
  _RFILE *dailyf;
  _RFILE *monthlyf;

  /* Open purchase display file, daily transaction file and monthly */
  /* transaction file. */
  if ( ( purf = _Ropen ( "*LIBL/T1520DD7", "ar+,indicators=y" ) ) == NULL )
  {
    printf ( "Display file did not open.\n" );
    exit ( 1 );
  }
  if ( ( dailyf = _Ropen ( "*LIBL/T1520DD5", "wr,commit=y" ) ) == NULL )
  {
    printf ( "Daily transaction file did not open.\n" );
    exit ( 2 );
  }
  if ( ( monthlyf = _Ropen ( "*LIBL/T1520DD6","ar,commit=y" ) ) == NULL )
  {
    printf ( "Monthly transaction file did not open.\n" );
    exit ( 3 );
  }

  /* The associate separate indicator area with the purchase file. */
  _Rindara ( purf, ind_area );
/* Select the purchase record format. */
_Rformat ( purf, "PURCHASE" );

/* Invite the user to enter a purchase transaction. */
/* The _Rwrite function writes the purchase display. */
_Rwrite ( purf, "", 0 );
_Rreadn ( purf, buf, sizeof(buf), __DFT );

/* While the user is entering transactions, update daily and */
/* monthly transaction files. */
while ( rc && ind_area[PF03] == IND_OFF )
{
    rc = ( (_Rwrite ( dailyf, buf, sizeof(buf) ))->num_bytes );
    rc = rc && ( _Rwrite ( monthlyf, buf, sizeof(buf) ))->num_bytes;

    /* If the databases were updated, then commit transaction. */
    /* Otherwise, rollback the transaction and indicate to the */
    /* user that an error has occurred and end the application. */
    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
        _Rrollback ( );
        _Rformat ( purf, "ERROR" );
        _Rwrite ( purf, "", 0 );
        _Rreadn ( purf, buf, sizeof(buf), __DFT );
    }
}

_Ropen() 関数は、購入画面ファイル、日次トランザクション・ファイル、および月次トランザクション・ファイルを開きます。_Rindara() 関数は、購入ファイルの分離標識域を識別します。
_Rformat() 関数は、T1520DD7 で定義された購入レコードの様式を選択します。_Rwrite() 関数は購入オーダー画面を書き込みます。入力されたデータにより、日次および月次トランザクション・ファイル T1520DD5 および T1520DD6 が更新されます。トランザクションは、_Rcommit() 関数を使用するこれらのデータベース・ファイルにコミットされます。

2. コミットメント制御下でプログラム T1520COM を実行します。コマンド行に、次のように入力します。

    STRCMCTL LCKLVL(*CHG) NFOBJ(MYLIB/NFTOBJ (*FILE)) CMTSCOPE(*JOB)
    CALL PGM(MYLIB/T1520COM)

画面は次のように表示されます。
3. 以下のサンプル・データを使用してオンライン購入オーダー・フォームに入力します。

<table>
<thead>
<tr>
<th>TABLE</th>
<th>BENCH</th>
<th>CHAIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>43623</td>
<td>62513</td>
</tr>
</tbody>
</table>

アイテムおよび製造番号が入力されると、T1520DD5 および T1520DD6 ファイルが更新されます。3つの購入オーダー・アイテムがすべて入力されると、次にトランザクション・ファイル T1520DD5 にはサンプル・データが含まれた状態になります。

4. コミットメント制御を終了します。コマンド行に、次のように入力します。

```
ENDCMCTCL
```

ジャーナル JRN には、T1520DD5 および T1520DD6 に対して行われた変更に対応する項目が含まれています。

### レコードのブロック化

レコード・ブロック化を使用すると、入力または出力のためだけに開かれているファイルの入出力操作のパフォーマンスを向上することができます。`fopen()` 関数の呼び出しで `blksize=value` パラメーターを指定するか、もしくは `_Ropen()` 関数の呼び出しで `blkrcd=y` を指定して、レコード・ブロック化をオンにします。状況によっては、ファイルの処理時にオペレーティング・システムがブロック内のレコード 1 件のみを返す場合があります。この場合、パフォーマンスは向上しません。

OVRDBF コマンドに `SEQONLY(*YES)` を指定することで、プログラムを変更せずにレコード・ブロック化をオフにすることができます。

注: レコード・ブロック化が有効である場合、プログラムとシステム間でレコードのブロックが転送されるときにだけ、入出力フィードバック構造体が更新されます。

### プログラムでの装置ファイルの使用

このトピックでは、以下を行うための方法について説明します。

- すべての装置ファイルで IBM i フィードバック域を使用する。
- 標識を使用して、プログラムとシステム間で情報を転送する。
- デフォルトのプログラム装置を設定する。
すべての装置ファイルに関する IBM i フィードバック域の使用

装置属性のフィードバック域にアクセスするには、Rdevatr() 関数を使用します。レコード入出力関数でストリーム・ファイル (type=record) を使用するには、FILE ポインターを _RFILE ポインターにキャストする必要があります。

標識を使用した情報の転送

標識 を使用して、プログラムとシステムの間で相互に情報を受け渡すことができます。表示装置ファイル、ICF ファイル、およびプリンター・ファイルは標識を利用してすることができます。標識は、文字値 1 または 0 を含むブール・データ項目です。

このセクションでは、以下について説明します。

• 標識のタイプ
  • 別個の標識域
  • メジャーリプラコードとマイナー戻りコード
  • ファイル・バッファーに標識を戻す

プログラムが読み取りまたは書き込みを行うレコードに対して、次のように標識を指定できます。

• ファイル・バッファーの一部として指定
  • 別個の標識域で指定

標識のタイプ

以下の 2 つのタイプの標識があります。

オプション標識 は、プログラムからシステムに情報を渡します。例えば、レコード内のどのフィールドを表示できるかを制御できます。

応答標識 は、人力要求の終了時にシステムからアプリケーションに情報を渡します。例えば、これを使用して、ワークステーション・ユーザーが押したファンクション・キーをプログラムに通知できます。

標識を使用するには、外部記述ファイルとして、表示装置ファイル、ICF ファイル、および印刷出力ファイルを定義する必要があります。外部記述の表示装置ファイルのデータ記述仕様 (DDS) には、標識ごとに 1 文字の INDICATOR フィールドが含まれている必要があります。標識は、プログラムによって読み取りまたは書き込みされるレコード内に入れる (標識はファイル・バッファーに入ります) か、別個の標識領域に入れます。

分離標識域

標識域 は、0 から 98 までの添字を持つ 99 個のエレメントから成る文字配列です。
DDS で INDARA キーワード (indicators=y) を指定した場合、表示装置、ICF、および印刷出力ファイルの標識が分離標識域に返されます。
_Rindara() 関数を使用して、ファイルに関連した分離標識バッファーを識別します。

DDS で INDARA キーワードを指定しなかった場合、表示装置、ICF、および印刷出力ファイルの標識は、レコード・バッファーに示されます。DDS で定義されている標識の数および順序により、レコード・バッファーの標識の数および順序が決定されます。標識は常に、レコード・バッファー内で最初に位置付けられます。
_RFILE 構造体の in_buf ポインターおよび out_buf ポインターは、ファイルの出入力レコード・バッファーを指します。

メジャー戻りコードとマイナー戻りコード
メジャー戻りコードとマイナー戻りコードを使用して、表示装置ファイル、ICF ファイル、およびプリンター・ファイルの特定状況情報を報告します。メジャー戻りコードが 00 の場合、操作は正常に完了しています。表示装置ファイル、ICF ファイル、またはプリンター・ファイルでエラーが発生した場合は、それが発生したときにプログラムでそれを処理する必要があります。

読み取り (_Rreadindv() または _Rreadn()) 操作あるいは書き込み (_Rwrite()) 操作の後、_RIOFB_T 構造の sysparm フィールドが、表示装置ファイル、ICF ファイル、またはプリンター・ファイルのメジャー/マイナー戻りコードを指します。ヘッダー・ファイル <recio.h> で _RIOFB_T 構造が宣言されます。

プログラムで、各出力操作後に戻りコードをテストし、メジャー/マイナー戻りコードに基づくエラー処理操作を定義する必要があります。

「アプリケーション表示プログラミング」の資料に、表示装置ファイルのメジャー戻りコードとマイナー戻りコード、およびその意味について説明されています。「印刷装置プログラミング」の資料に、プリンター・ファイルのメジャー戻りコードとマイナー戻りコード、およびその意味について説明されています。

例: 分離標識域に標識を戻す
DDS で INDARA キーワードを使用して、プログラムがレコード内の標識を分離標識域に読み取るまたは書き込むように指定することができます。

以下の例は、分離標識域に標識を戻す方法を示しています。DDS で指定された INDARA キーワードは、画面の標識が分離標識域に戻されることを示します。

手順:
1. 以下に示す DDS ソースを使用して表示装置ファイル T1520DD0 を作成するには、次のように入力します。
   CRTDSPPF FILE(MYLIB/T1520DD0) SRCFILE(QCPPLE/QADDSSRC)
   7-43 ページの図 7-29 は DDS ソースを示したものです。
2. 7-44 ページの図 7-30 に示されたソースを使用してプログラム T1520ID2 を作成するには、次のように入力してください。
   CTRBND PGM(MYLIB/T1520ID2) SRCFILE(QCPPLE/QACSSRC)
3. プログラム T1520ID2 を実行するには、次のように入力します。
   CALL PGM(MYLIB/T1520ID2)
   出力は以下のようにになります。
 PHONE BOOK
 Name:
 Address:
 Phone #:  

 F3 - EXIT

ソース・コードのサンプル:

A R PHONE
A CF03('EXIT')
A 135 'PHONE BOOK'
A DSPATR(HI)
A 728 'Name:'
A NAME 11A I 7 34
A 925 'Address:'
A ADDRESS 20A I 9 34
A 1125 'Phone #:'
A PHONE_NUM 8A I 11 34
A 2334 'F3 - EXIT'
A DSPATR(HI)

図 7-29. T1520DDO — 電話帳表示のための DDS ソース
/* This program uses response indicators to inform the program that */
/* F3 was pressed by a user to indicate that an input request */
/* finished. The response indicators are returned in a separate */
/* indicator area. */

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

typedef struct{
    char name[11];
    char address[20];
    char phone_num[8];
} info;

#define IND_ON '1'
#define F3 2

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    info phone_list;
    _SYSindara indicator_area;
    if (( fp = _Ropen("*LIBL/T1520DD0", "ar+ indicators=y")) == NULL )
    {
        printf("display file open failed\n");
        exit(1);
    }
    _Rindara(fp, indicator_area);
    _Rformat(fp, "PHONE");
    rfb = _Rwrite(fp, "", 0);
    rfb = _Rreadn(fp, &phone_list, sizeof(phone_list), __DFT);
    if ( indicator_area[F3] == IND_ON )
    {
        printf("user pressed F3\n");
    }
    _Rclose(fp);
}

図7-30. T1520ID2 — 分離標識域で標識を指定するための ILE C プログラム

注:
このプログラムは、応答標識 IND_ON ‘1’ と F3 2 を使用して ILE C プログラム T1520ID2 にユーザーが F3 を押したことを伝えます。_Rindara()関数は、外部記述ファイル T1520DD0と関連付けられた分離標識バッファー indicator_area にアクセスします。キーワード indicators=yes で表示装置ファイル T1520DD0 がオープンされ、標識が分離域へ戻されます。

例: ファイル・バッファーに標識を戻す
以下の例では、プログラム T1520ID1 が読み取るレコードに標識を指定する方法を示します。標識は、外部記述ファイルのファイル・バッファーに入ります。外部記述ファイルの DDS に、1 文字の標識フィールドが含まれます。

手順:
1. 表示装置ファイル T1520DD9 を作成するには、次のように入力します。
   CRTDSPF FILE(MYLIB/T1520DD9) SRCPFLE(QCPPLE/QADDSSRC)
   7-45 ページの図 7-31 は DDS ソースを示したもののです。
2. 7-46 ページの図 7-32 に示されたプログラム・ソースを使用してプログラム T1520ID1 を作成するには、次のように入力してください。
   CRTBNDC PGM(MYLIB/T1520ID1) SRCPFLE(QCPPLE/QACSRC)

7-44 IBM i: ILE C/C++ プログラマーの手引き
3. プログラム T1520ID1 を実行するには、次のように入力してください。
CALL PGM(MYLIB/T1520ID1)

出力は以下のようになります。

<table>
<thead>
<tr>
<th>PHONE BOOK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name:</td>
</tr>
<tr>
<td>Address:</td>
</tr>
<tr>
<td>Phone #:</td>
</tr>
<tr>
<td>F3 - EXIT</td>
</tr>
</tbody>
</table>

コード・サンプル:

```
A      R PHONE
A
A       CF83(G3 'EXIT')
A       1 35 'PHONE BOOK'
A       DSPATR(HI)
A       7 28 'Name: '
A       NAME 11A I 7 34
A       9 25 'Address: '
A       ADDRESS 20A I 9 34
A       11 25 'Phone #: '
A       PHONE_NUM 8A I 11 34
A       23 34 'F3 - EXIT'
A       DSPATR(HI)
```

図 7-31. T1520DD9 一 電話帳表示のための DDS ソース
/* This program uses a response indicator to inform the program that F3 was pressed by a user. The response indicator is returned in part of the file buffer. */
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

typedef struct{
    char in03;
    char name[11];
    char address[20];
    char phone_num[8];
}info;
#define IND_ON '1'

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    info phone_list;

    if ((fp = _Ropen("*LIBL/T1520DD9", "ar+")) == NULL)
    {
        printf("display file open failed\n");
        exit(1);
    }

    _Rformat(fp, "PHONE");
    rfb = _Rwrite(fp, "", 0);
    rfb = _Rreadn(fp, &phone_list, sizeof(phone_list), __DFT);
    if (phone_list.in03 == IND_ON)
    {
        printf("user pressed F3\n");
    }

    _Rclose(fp);
}

図7-32. T1520ID1 — ファイル・バッファーの一部として標識を指定するための ILE C ソース

注: このプログラムは、応答標識 IND_ON '1' を使用して ILE C プログラム T1520ID1 にユーザーが F3 を押したことを伝えます。

デフォルトのプログラム装置の設定
表示装置ファイルおよびICF ファイルのデフォルト装置を設定できます。

例:
以下の例で、_Racquire() 関数を使用して、表示装置ファイルのデフォルトのプログラム装置を明示的に設定する方法を示します。

注: この例を実行する際には、DEVICE2 の代わりに、ご使用のシステムで定義されている表示装置を使用する必要があります。

1. 以下に示している DDS を使用して表示装置ファイル T1520DDD を作成するには、以下を入力します。
   
   CRTDSPF FILE(MYLIB/T1520DDD) SRCFILE(QCPPLE/QADDSSRC) MAXDEV(2)
2. ファイル STDOUT を印刷出力ファイル QPRINT でオーバーライドするには、以下を入力します。
OVRPRTF FILE(STDOUT) TOFILE(QPRINT)
3. 以下に示しているソースを使用してプログラム T1520DEV を作成するには、以下を入力します。
CRTBNDC PGM(MYLIB/T1520DEV) SRCFILE(QCPPLE/QACSRC)
/* This program establishes a default device using the _Racquire function. */

#include <stdio.h>
#include <recio.h>
#include <signal.h>
#include <stdlib.h>

void handler ( int );

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    char buf[21];

    signal (SIGALL, handler);

    if ((fp = _Ropen("*LIBL/T1520DDD","ar+")) == NULL)
    {
        printf("Could not open the display file\n");
        exit(1);
    }
    _Racquire (fp,"DEVICE2"); /* Acquire the device. */
    /* DEVICE2 is now the */
    /* default program device. */
    _Rformat (fp,"EXAMPLE"); /* Select the record */
    /* format. */
    _Rwrite (fp,"Hello",5); /* Write to the default */
    /* program device. */
    rfb = _Rreadn(fp,buf,20,__DFT); /* Read from the default */
    /* program device. */

    buf[rfb->num_bytes] = '\0';

    printf("Response from device : %s\n",buf);
    _Rrelease (fp,"DEVICE2");
    _Rclose (fp);
}

void handler ( int sig )
{
    printf("message = %7.7s",_EXCP_MSGID);
    printf("program continues \n");
    signal (SIGALL, handler);
}

/* This program establishes a default device using the _Racquire function. */

#include <stdio.h>
#include <recio.h>
#include <signal.h>
#include <stdlib.h>

void handler ( int );

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    char buf[21];

    signal (SIGALL, handler);

    if ((fp = _Ropen("*LIBL/T1520DDD","ar+")) == NULL)
    {
        printf("Could not open the display file\n");
        exit(1);
    }
    _Racquire (fp,"DEVICE2"); /* Acquire the device. */
    /* DEVICE2 is now the */
    /* default program device. */
    _Rformat (fp,"EXAMPLE"); /* Select the record */
    /* format. */
    _Rwrite (fp,"Hello",5); /* Write to the default */
    /* program device. */
    rfb = _Rreadn(fp,buf,20,__DFT); /* Read from the default */
    /* program device. */

    buf[rfb->num_bytes] = '\0';

    printf("Response from device : %s\n",buf);
    _Rrelease (fp,"DEVICE2");
    _Rclose (fp);
}

void handler ( int sig )
{
    printf("message = %7.7s",_EXCP_MSGID);
    printf("program continues \n");
    signal (SIGALL, handler);
}

/* This program establishes a default device using the _Racquire function. */

#include <stdio.h>
#include <recio.h>
#include <signal.h>
#include <stdlib.h>

void handler ( int );

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    char buf[21];

    signal (SIGALL, handler);

    if ((fp = _Ropen("*LIBL/T1520DDD","ar+")) == NULL)
    {
        printf("Could not open the display file\n");
        exit(1);
    }
    _Racquire (fp,"DEVICE2"); /* Acquire the device. */
    /* DEVICE2 is now the */
    /* default program device. */
    _Rformat (fp,"EXAMPLE"); /* Select the record */
    /* format. */
    _Rwrite (fp,"Hello",5); /* Write to the default */
    /* program device. */
    rfb = _Rreadn(fp,buf,20,__DFT); /* Read from the default */
    /* program device. */

    buf[rfb->num_bytes] = '\0';

    printf("Response from device : %s\n",buf);
    _Rrelease (fp,"DEVICE2");
    _Rclose (fp);
}

void handler ( int sig )
{
    printf("message = %7.7s",_EXCP_MSGID);
    printf("program continues \n");
    signal (SIGALL, handler);
}

/* This program establishes a default device using the _Racquire function. */

#include <stdio.h>
#include <recio.h>
#include <signal.h>
#include <stdlib.h>

void handler ( int );

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    char buf[21];

    signal (SIGALL, handler);

    if ((fp = _Ropen("*LIBL/T1520DDD","ar+")) == NULL)
    {
        printf("Could not open the display file\n");
        exit(1);
    }
    _Racquire (fp,"DEVICE2"); /* Acquire the device. */
    /* DEVICE2 is now the */
    /* default program device. */
    _Rformat (fp,"EXAMPLE"); /* Select the record */
    /* format. */
    _Rwrite (fp,"Hello",5); /* Write to the default */
    /* program device. */
    rfb = _Rreadn(fp,buf,20,__DFT); /* Read from the default */
    /* program device. */

    buf[rfb->num_bytes] = '\0';

    printf("Response from device : %s\n",buf);
    _Rrelease (fp,"DEVICE2");
    _Rclose (fp);
}

void handler ( int sig )
{
    printf("message = %7.7s",_EXCP_MSGID);
    printf("program continues \n");
    signal (SIGALL, handler);
}

/* This program establishes a default device using the _Racquire function. */

#include <stdio.h>
#include <recio.h>
#include <signal.h>
#include <stdlib.h>

void handler ( int );

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    char buf[21];

    signal (SIGALL, handler);

    if ((fp = _Ropen("*LIBL/T1520DDD","ar+")) == NULL)
    {
        printf("Could not open the display file\n");
        exit(1);
    }
    _Racquire (fp,"DEVICE2"); /* Acquire the device. */
    /* DEVICE2 is now the */
    /* default program device. */
    _Rformat (fp,"EXAMPLE"); /* Select the record */
    /* format. */
    _Rwrite (fp,"Hello",5); /* Write to the default */
    /* program device. */
    rfb = _Rreadn(fp,buf,20,__DFT); /* Read from the default */
    /* program device. */

    buf[rfb->num_bytes] = '\0';

    printf("Response from device : %s\n",buf);
    _Rrelease (fp,"DEVICE2");
    _Rclose (fp);
}

void handler ( int sig )
{
    printf("message = %7.7s",_EXCP_MSGID);
    printf("program continues \n");
    signal (SIGALL, handler);
}

/* This program establishes a default device using the _Racquire function. */

#include <stdio.h>
#include <recio.h>
#include <signal.h>
#include <stdlib.h>

void handler ( int );

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    char buf[21];

    signal (SIGALL, handler);

    if ((fp = _Ropen("*LIBL/T1520DDD","ar+")) == NULL)
    {
        printf("Could not open the display file\n");
        exit(1);
    }
    _Racquire (fp,"DEVICE2"); /* Acquire the device. */
    /* DEVICE2 is now the */
    /* default program device. */
    _Rformat (fp,"EXAMPLE"); /* Select the record */
    /* format. */
    _Rwrite (fp,"Hello",5); /* Write to the default */
    /* program device. */
    rfb = _Rreadn(fp,buf,20,__DFT); /* Read from the default */
    /* program device. */

    buf[rfb->num_bytes] = '\0';

    printf("Response from device : %s\n",buf);
    _Rrelease (fp,"DEVICE2");
    _Rclose (fp);
}

void handler ( int sig )
{
    printf("message = %7.7s",_EXCP_MSGID);
    printf("program continues \n");
    signal (SIGALL, handler);
}

/* This program establishes a default device using the _Racquire function. */

#include <stdio.h>
#include <recio.h>
#include <signal.h>
#include <stdlib.h>

void handler ( int );

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    char buf[21];

    signal (SIGALL, handler);

    if ((fp = _Ropen("*LIBL/T1520DDD","ar+")) == NULL)
    {
        printf("Could not open the display file\n");
        exit(1);
    }
    _Racquire (fp,"DEVICE2"); /* Acquire the device. */
    /* DEVICE2 is now the */
    /* default program device. */
    _Rformat (fp,"EXAMPLE"); /* Select the record */
    /* format. */
    _Rwrite (fp,"Hello",5); /* Write to the default */
    /* program device. */
    rfb = _Rreadn(fp,buf,20,__DFT); /* Read from the default */
    /* program device. */

    buf[rfb->num_bytes] = '\0';

    printf("Response from device : %s\n",buf);
    _Rrelease (fp,"DEVICE2");
    _Rclose (fp);
}

void handler ( int sig )
{
    printf("message = %7.7s",_EXCP_MSGID);
    printf("program continues \n");
    signal (SIGALL, handler);
}
デフォルトのプログラム装置の変更

デフォルト・ファイル用のデフォルト装置は変更することができます。

例:

以下の例では, _Rpgmdev() 関数を使用してデフォルトのプログラム装置を変更する方法を説明しています。

注: この例を実行するには、ご使用のシステムで定義された 2 つの表示装置を、DEVICE1 および
DEVICE2 の代わりに使用する必要があります。

1. 以下に示す DDS を使用して表示装置ファイル T1520DDE を作成するには、次のように入力します。

```
CRTDSPF FILE(MYLIB/T1520DDE) SRCFILE(QCPPLE/QADDSSRC) MAXDEV(2)
```

2. ファイル STDOUT を印刷出力ファイル QPRINT でオーバーライドするには、以下を入力します。

```
OVRPRTF FILE(STDOUT) TOFILE(QPRINT)
```

3. 以下に示すソースを使用してプログラム T1520CDV を作成するには、次のように入力します。

```
/* This program illustrates how to change a default program device. */
/* using the _Racquire(), _Rpgmdev(), _Rreadindv() and _Rrelease() */
/* functions. */

#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct{
    char name[20];
    char address[25];
}format1;
```

図7-35. T1520DDE ー 名前およびパスワードの表示のための DDS ソース
char name[8];
char password[10];
}format2;

typedef union{
    format1 fmt1;
    format2 fmt2;
}formats;

void io_error_check(_RIOFB_T *rfb)
{
    if (memcmp(rfb->sysparm->_Maj_Min.major_rc,"00",2) ||
        memcmp(rfb->sysparm->_Maj_Min.minor_rc,"00",2))
    {
        printf("I/O error occurred, program ends.\n");
        exit(1);
    }
}

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    _XXIOFB_T *iofb;
    int size;
    formats buf;

    /* Open the device file. */
    if ((fp = _Ropen("*LIBL/T1520DDE", "ar+")) == NULL)
    {
        printf("Could not open file\n");
        exit(1);
    }
    _Racquire(fp,"DEVICE1"); /* Acquire another device. */
    /* Replace with the actual */
    /* device name. */
    _Rformat(fp,"FORMAT1"); /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite(fp,"",0); /* Set up the display. */
    io_error_check(rfb);
    _Racquire(fp,"DEVICE2"); /* Acquire another device. */
    _Rpgmdev(fp,"DEVICE2"); /* Change the default program */
    /* device. Replace with the */
    /* actual device name. */
    /* Device2 implicitly acquired at */
    /* open. */
    _Rformat(fp,"FORMAT2"); /* Set the record format for the */
    /* the display file. */
    rfb = _Rwrite(fp,"",0); /* Set up the display. */
    io_error_check(rfb);
    /* Determine which terminal responded first. */
    iофb = _Riofbk(fp);
    if (!strncmp("FORMAT1", iофb->rec_format, 10))

{ 
    _Rrelease ( fp, "DEVICE1" );
} 
else 
{ 
    _Rrelease(fp, "DEVICE2" );
} 
return(0);

ILE C プログラム T1520CDV は、_Racquire() 関数を使用して DEVICE1 という名前の別の装置を明示的に獲得します。DEVICE1 は現在のプログラム装置になります。_Rpgmdev() 関数は、DEVICE1 という名前の現在装置を DEVICE2 に変更します。_Rreadindv() 関数は、DEVICE1 からレコードを読み取ります。_Release() 関数は、DEVICE1 および DEVICE2 を解放します。

4. プログラム T1520CDV を実行するには、次のように入力します。

CALL PGM(MYLIB/T1520CDV)

出力は以下のようにになります。

```
Name: 
Password: 
```

アプリケーションが実行されると、各装置に別の画面が表示されます。データは両方の画面で入力することができますが、プログラムに返されるのは最初に入力されたデータです。プログラムからの出力は QPRINT にあります。例えば、DEVICE2 でデータが入力される前に DEVICE1 で名前 SMITH と住所 10 MAIN ST が入力された場合、ファイル QPRINT には以下が含まれます。

Data displayed on DEVICE1 is SMITH 10 MAIN ST

注: 上記の例で作成されたレコード様式は 2 つあります。1 つは 45 文字のサイズ (fmt1) で、もう 1 つは 18 文字のサイズ (fmt2) です。共用 buf には 2 つのレコード様式宣言が含まれます。

フィードバック情報の取得

IBM i システム・フィードバック領域を使用して、アプリケーションに関連したプログラム装置に関する追加情報を取得できます。

例:

以下の例では、_Riofbk() 関数を使用します。

1. 以下に示している DDS ソースを使用して表示装置ファイル T1520DDF を作成するには、以下を入力します。

   CRTPSF FILE(MYLIB/T1520DDF) SRCFILE(QCPPLE/QADDSSRC) MAXDEV(2)
2. ファイル STDOUT を印刷出力ファイル QPRINT でオーバーライドするには、以下を入力します。

```
OVRPRTF FILE(STDOUT) TOFILE(QPRINT)
```
/* This program illustrates how to use the _Riofbk function to access */ /* the I/O feedback area. */
#include <stdio.h>
#include <recio.h>
#include <signal.h>
#include <xxfdbk.h>
#include <stdlib.h>
#include <string.h>
static void handler (int);

_RFILE *fp; /* Signal handler for _Racquire exceptions */

static void handler (int sig)
{
    _XXIOFB_T *io_feedbk;
    _XXIOFB_DSP_ICF_T *dsp_io_feedbk;
    signal ( SIGIO, handler );
    io_feedbk = _Riofbk ( fp );
    dsp_io_feedbk = (_XXIOFB_DSP_ICF_T *)((char *)(io_feedbk) +
    io_feedbk->file_dep_fb_offset );
    printf ( "Acquire failed\n" );
    printf ( "Major code: %2.2s	Minor code: %2.2s
",
        dsp_io_feedbk->major_ret_code,dsp_io_feedbk->minor_ret_code );
    exit (1);
}

int main(void)
{
    char buf[20];
    _RIOFB_T *rfb;
    if (( fp = _Ropen ( "*LIBL/T1520DDF", "ar+" ) == NULL )
    {
        printf ( "Could not open the display file
" );
        exit (2);
    }
    signal ( SIGIO, handler );
    _Racquire ( fp, "DEVICE1" ); /*Acquire the device. DEVICE1 is */ /* now the default program device. */ /* NOTE : If the device is not */ /* acquired, exceptions are issued. */
    _Rformat ( fp, "EXAMPLE" ); /* Select the record format. */
    _Rwrite ( fp, "Hello", 5 ); /* Write to default program device. */
    _Rreadn ( fp, buf, 21, __DFT );
    printf ( "user entered: %20.20s
", buf );
    _Rclose ( fp );
    return(0);
}
エラーの前に signal() 関数が呼び出され、シグナル・ハンドラーが設定されます。取得操作時に例外が発生した場合、シグナル・ハンドラーが呼び出され、メジャー/マイナー戻りコードが stdout に書き込まれます。

```
Acquire failed
Major code: 82 Minor code: AA
```

表示装置ファイルおよびサブファイルの使用

表示装置ファイルおよびサブファイル

表示装置ファイル は、表示装置に表示する情報のフォーマットを定義するために使用されます。また、表示装置との間でシステムがその情報を処理する方法についても定義します。

サブファイル は、相対レコード番号によってアクセスできる同じレコード様式を持つレコードのグループが入った表示装置ファイルです。サブファイルのレコードは、表示装置に表示できます。システムは、1つの操作でレコードのグループ全体を表示装置に送り、別の1つの操作でグループを表示装置から受け取ります。両方ともオブジェクト・タイプは *FILE です。

外部記述表示装置ファイルを処理するには、以下のいずれかを使用します。
- Code/400 エディターまたは SEU を介した DDS
- 画面設計機能 (SDA) または DSU What-You-See-Is-What-You-Get (WYSIWYG) ソール

表示装置ファイルの入出力に関する考慮事項:
- ILE C/C++ プログラムは表示装置ファイルをプログラム記述ファイルまたは外部記述ファイルとして処理することができます。
  - プログラム記述の表示装置ファイルの場合、そのファイルを使用する ILE C/C++ プログラムすべてのフォーマット情報と制御情報を指定します。プログラム記述の表示装置ファイルを作成するには、CRTDSFP コマンドで SRCFILE(*NONE) を指定してください。
  - 外部記述の表示装置ファイルの場合、DDS を使用してすべてのフォーマット情報と制御情報を指定し、表示装置のレイアウトを記述します。外部記述の表示装置ファイルを作成するには、CRTDSFP コマンドの SRCFILE パラメーターで、DDS ソースを含むメンバーの名前を指定してください。
- ユーザー定義のデータ・ストリーム (UDDS) をご使用の場合、16進数 3F (X'3F') は次の表示属性まで表示装置をブランクにします。何らかの CCSID 変換が行われ、文字を別のコード・ページの対応する文字にマップできない場合、その文字は 16進数 3F にマップされます。これによって、次の表示属性まで表示画面がブランクになります。CCSID については、9-1ページの「プログラムの国際化」を参照してください。
- 追加モードを使用したファイルのクリアまたはファイルのオープンの概念は、表示装置ファイルには適用されません。
サブファイルの入出力に関する考慮事項:

- レコード様式サブファイルの入力型定義には、１、０、Ｂ、および Ｈ の使用法を含むフィールドが含まれます。制御レコード様式サブファイルの入力型定義には、１、８、および Ｈ の使用法を含むフィールドが含まれます。

- サブファイルを使用するには、例えばデータベース・ファイルからレコードを読み取り、それをサブファイルに書き込むなどして、そのサブファイルを初期化します。このときの書き込みには _Rwritten を使用する必要があります。

サブファイルを使用した入出力操作の最小化

サブファイルを使用すると、単一の操作で、複数のレコードを表示装置から読み取ったり表示装置へ書き込んだりすることができます。

例:

以下のサブファイルの例は、T1520DDG および T1520DDH の DDS を使用して名前と電話番号のリストを表示します。

1. 以下に示す DDS ソースを使用して表示装置ファイル T1520DDG を作成するには、次のように入力します。
   
   CRTDSPF FILE(MYLIB/T1520DDG) SRCFILE(QCPPLE/QADDSSRC)

   A R SFL SFL
   A NAME 10A B 10 25
   A PHONE 10A B +5
   A R SFLCTL SFLCTL(SFL)
   A SFLPAG(5)
   A SFLSIZ(26)
   A SFLDSP
   A SFLDSPCTL
   A 22 25'<PAGE DOWN> FOR NEXT PAGE'
   A 23 25'<PAGE UP> FOR PREVIOUS PAGE'

   図 7-38. T1520DDG — サブファイル表示の DDS ソース

2. 以下に示す DDS ソースを使用して物理ファイル T1520DDH を作成するには、次のように入力します。
   
   CRTPF FILE(MYLIB/T1520DDH) SRCFILE(QCPPLE/QADDSSRC)

   R ENTRY
   NAME 10A
   PHONE 10A

3. T1520DDH に以下のデータを入力します。

   - David 435-5634
   - Florence 343-4537
   - Irene 255-5235
   - Carrie 747-5347
   - Michele 634-4557

4. 以下に示すソースを使用してプログラム T1520SUB を作成するには、次のように入力します。
   
   CTRBNDCE PGM(MYLIB/T1520SUB) SRCFILE(QCPPLE/QACSRC)

   /* This program illustrates how to use subfiles. */
   #include <stdio.h>
   #include <stdlib.h>
   #include <reclib.h>
   #define LEN 10
#define NUM_RECS 20
#define SUBFILENAME "*LIBL/T1520DDG"
#define PFILENAME "*LIBL/T1520DDH"

typedef struct{
    char name[LEN];
    char phone[LEN];
}pf_t;
#define RECLEN sizeof(pf_t)

void init_subfile(_RFILE *, _RFILE *);

int main(void)
{
    _RFILE *pf;
    _RFILE *subf;
    /* Open the subfile and the physical file. */
    if ((pf = _Ropen(PFILENAME, "rr")) == NULL)
    {
        printf("can't open file \"%s\"n", PFILENAME);
        exit(1);
    }
    if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL)
    {
        printf("can't open file \"%s\"n", SUBFILENAME);
        exit(2);
    }
    /* Initialize the subfile with records from the physical file. */
    init_subfile(pf, subf);
    /* Write the subfile to the display by writing a record to the */
    /* subfile control format. */
    _Rformat(subf, "SFLCTL");
    _Rwrite(subf, "", 0);
    _Rreadn(subf, "", 0, __DFT);
    /* Close the physical file and the subfile. */
    _Rclose(pf);
    _Rclose(subf);
}

void init_subfile(_RFILE *pf, _RFILE *subf)
{
    _RIOFB_T *fb;
    int i;
    pf_t record;
    /* Select the subfile record format. */
    _Rformat(subf, "SFL");
    for (i = 1; i <= NUM_RECS; i++)
    {
        fb = _Rreadn(pf, &record, RECLEN, __DFT);
        if (fb->num_bytes != EOF)
        {
            fb = _Rwrited(subf, &record, RECLEN, i);
            if (fb->num_bytes != RECLEN)
            {
                printf("error occurred during write\n\n");
                exit(3);
            }
        }
    }
}

このプログラムは、_Ropen() を使用してサブファイル T1520DDG と物理ファイル T1520DDH をオープンします。その後、このサブファイルは物理ファイルからのレコードで初期化されます。サブファイル・レコードは、_Rwrited() 関数を使用して表示装置に書き込まれます。
5. プログラム T1520SUB を実行して出力を確認するには、次のように入力してください。

CALL PGM(MYLIB/T1520SUB)

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>David</td>
<td>435-5634</td>
</tr>
<tr>
<td>Florence</td>
<td>343-4537</td>
</tr>
<tr>
<td>Irene</td>
<td>255-5235</td>
</tr>
<tr>
<td>Carrie</td>
<td>747-5347</td>
</tr>
<tr>
<td>Michele</td>
<td>643-4557</td>
</tr>
</tbody>
</table>

表示装置ファイルおよびサブファイルをバイナリー・ストリーム・ファイルとしてオープン

レコード単位の処理のバイナリー・ストリーム・ファイルとして IBM i 表示装置ファイルまたはサブファイルをオープンするには、以下のいずれかのモードで fopen() 関数を使用します。

- rb
- ab+
- wb
- ab

表示装置ファイルを作成する方法は、CRTDSPF コマンドのみです。fopen() 関数を使用し、表示装置ファイルが存在しない場合は、物理データベース・ファイルが作成されます。

有効なキーワード・パラメーターは次のとおりです。

- type
- lrecl
- indicators

バイナリー・ストリーム・サブファイルの入出力に関する考慮事項: バイナリー・ストリーム・サブファイルに対しては、メッセージ・サブファイルのみがサポートされています。

バイナリー・ストリーム表示装置ファイルのプログラム装置: 表示装置ファイルと関連付けられたプログラム装置は、ワークステーションです。デフォルト装置の設定は、fopen() 関数を使用して暗黙的にその装置を獲得することで行います。

表示装置ファイルおよびサブファイルのバイナリー・ストリーム関数: 以下のバイナリー・ストリーム関数を使用して、表示装置ファイルおよびサブファイルを処理します。

- fclose()
- fopen()
- fread()
- freopen()
- fwrite()

表示装置ファイルをレコード・ファイルとしてオープン

IBM i 表示装置ファイルまたはサブファイルをレコード・ファイルとしてオープンするには、以下のいずれかのモードで _Ropen() 関数を使用します。
レコード表示装置ファイルの入出力に関する考慮事項:

表示装置ファイルと関連付けられたプログラム装置は、ワークステーションです。デフォルト装置の設定は、_Ropen() 関数を使用して暗黙的にその装置を獲得することで行います。暗黙的に獲得されるプログラム装置は、CRTDSPF、CHGDSPF、またはOVRDSPF コマンドの DEV パラメーターによって決まります。DEV パラメーターに *REQUESTER が指定されている場合、プログラムを呼び出したプログラム装置が暗黙的に獲得されます。この装置が、表示装置ファイルに対する入出力操作のデフォルトのプログラム装置になります。

CRTDSPF、CHGDSPF、またはOVRDSPF コマンドの DEV パラメーターに *NONE が指定されている場合、_Racquire() 関数を使用してプログラム装置を明示的に獲得する必要があります。この場合、その明示的に獲得されたプログラム装置が、装置ファイルに対するそれ以降の入出力操作のデフォルト装置になります。

デフォルトのプログラム装置は、以下の方法で変更することができます。

- _Racquire() 関数を使用して、明示的に別のプログラム装置を獲得します。このとき獲得された装置が现行のプログラム装置になります。
- _Rpmsgdev() 関数を使用して、ファイルと関連付けられた現行のプログラム装置を以前獲得した装置に変更します。このプログラム装置は、そのファイルに対するそれ以降の入出力操作に使用できます。
- _Rreadindv() 関数を使用して送信動揺された装置から読み取る場合、読み取られる実際のプログラム装置がデフォルト装置になります。
- _Rrelease() 関数を使用して、ファイルから装置を解放します。装置を解放すると、その装置はそれ以降入出力操作に使用できません。

レコード・サプライファイルの入出力に関する考慮事項:

サプライファイル・レコード・フォーマットに対して入出力操作を行っても、データは表示装置に表示されません。表示装置との間でデータを転送するには、サプライファイル制御レコード・フォーマットの読み取りまたは書き込みを行う必要があります。サプライファイル・レコード・フォーマットとサプライファイル制御フォーマットを識別するには、_Rformat() 関数を使用します。

_Reqfram() 関数で指定したフォーマットがサプライファイル・レコード・フォーマットを参照する場合、表示装置との間でデータは転送されません。

次の変更済みサプライファイル・レコードを読み取るには、_Rreadnc() 関数を使用します。この関数は、ファイル内の現在位置から、次の変更済みレコードを検索します。これが最初の読み取り操作である場合は、サプライファイル内の最初の変更済みレコードが読み取られます。変更済みレコードを検査する前にファイルの終わりに到達すると、_RIOFB_T 構造体の num_bytes フィールドに EOF が返されます。
システム間通信機能ファイルの使用
システム間通信機能 (ICF) ファイルでは、異なるシステム上の2つのプログラム間で送受信されるデータのレイアウトを定義し、リモート・システムとの通信に使用される構成オブジェクトにユーザーをリンクします。ICFプログラミングの資料に、ICFファイルに関する情報があります。

システム間連絡関数ファイルの入出力に関する考慮事項
- ILE C/C++プログラムはICFファイルをプログラム記述ファイルまたは外部記述ファイルとして処理することができます（システム・ファイル QSYS/QICDMF にはシステム提供のレコード様式が含まれています）。
- 追加モードを使用したファイルのクリアまたはファイルのオープンの概念は、ICFファイルには適用されません。追加モード (ar+ または ab+)を使用してICFファイルをオープンする場合、そのファイルは入出力のためにオープンされます。
- データの可変長を書き込む場合、DDS でキーワード VARLEN を使用する必要があります。
- CRTICFF コマンドですべてのICFファイルに対してプログラム装置の最大数を2以上に設定することで、ICFロケール・モードをアプリケーション・レベルで使用不可にすることができます。
ICF ファイルをバイナリー・ストリーム・ファイルとしてオープンする

IBM i の ICF ファイルをレコード単位の処理用にバイナリー・ストリーム・ファイルとしてオープンするには、fopen() 関数を以下のいずれかのモードで使用してください。

- rb
- wb
- ab
- ab+

注: ICF ファイルを作成する唯一の方法は、CRTICFF コマンドを使用することです。fopen() 関数を使用する場合で、かつ ICF ファイルが存在しない場合、物理データベース・ファイルが作成されます。

有効なキーワード・パラメーターは次のとおりです。

- type
- indicators

バイナリー・ストリーム ICF ファイルの入出力に関する考慮事項

fwrite() 関数は、正常に書き込みされたエレメントの数を返します。PDATA を使用する場合、fwrite() 関数によって返された値は PDATA を考慮に入れません。PDATA を使用する際、すべてのデータが正常に書き込みされても、errno は ETRUNC に設定されます。

バイナリー・ストリーム ICF ファイルのプログラム装置: ICF ファイルと関連付けられたプログラム装置は、通信セッションです。デフォルト装置の設定は、fopen() 関数を使用して暗黙的にその装置を獲得することで行います。

ICF ファイルのバイナリー・ストリーム関数

以下のバイナリー・ストリーム関数を使用して、ICF ファイルを処理します。

- fclose()
- fopen()
- fread()
- freopen()
- fwrite()

ICF ファイルをレコード・ファイルとしてオープンする

IBM i の ICF ファイルをレコード・ファイルとしてオープンするには、_Ropen() 関数を以下のいずれかのモードで使用してください。

- rr
- rr+
- wr+
- ar+
- wr
- ar

有効なキーワード・パラメーターは次のとおりです。

- indicators
- riofb
レコード ICF ファイルの入出力に関する考慮事項

_Rwrite() 関数は、通信回線を介して正常に転送された文字数を返します。PDATA を使用している場合、_fwrite() 関数とは異なり、_Rwrite() 関数によって返される値 (num_bytes) には、PDATA が含まれません。

レコード ICF ファイルのプログラム装置：ICF ファイルと関連付けられたプログラム装置は、通信セッションです。デフォルト装置の設定は、_Ropen() 関数を使用して暗黙的にその装置を得ることで行います。暗黙的に得られるプログラム装置は、CRTCFF、OVRICFF、または CHGICFF コマンドの ACQPGMDEV パラメーターによって決まります。プログラム装置名が ACQPGMDEV パラメーターに指定されている場合、装置ファイルがオープンされる前に、装置ファイルにプログラム装置を定義する必要があります。これを行うには、ADDICFDEVE または OVRICFDEVE コマンドの PMDEV パラメーターで装置名を指定してください。

CRTCFF、OVRICFF、または CHGICFF コマンドの ACQPGMDEV パラメーターに *NONE が指定されている場合、_Racquire() 関数を使用してプログラム装置を明示的に獲得する必要があります。

デフォルトのプログラム装置は、以下的方法で変更することができます。

- _Racquire() 関数を使用して、明示的に別のプログラム装置を獲得します。このとき獲得された装置が現行のプログラム装置になります。
- _Rpgmdev() 関数を使用して、ファイルと関連付けられた現行のプログラム装置を以前獲得した装置に変更します。このプログラム装置は、そのファイルに対するそれ以降の入出力操作に使用できます。
- _Rreadindv() 関数を使用して送信動作された装置から読み取る場合、読み取られる実際のプログラム装置がデフォルト装置になります。
- _Rrelease() 関数を使用して、ファイルから装置を解放します。装置を解放すると、その装置はそれ以降入出力操作に使用できません。

プログラム装置を解放するには、_Rrelease() 関数を使用してください (このときプログラム装置は事前に獲得済みでなければなりません)。これにより、装置がオープン・ファイルから切り離されます。このため、この装置に対しては入出力操作を行えなくなります。この装置を解放後に使用したい場合は、再度これを獲得する必要があります。

すべてのプログラム装置は、ファイルをクローズしたときに暗黙的に解放されます。装置ファイルが共用オープン・データ・パスがある場合、プログラム装置は最後のクローズ操作によって解放されます。

ICF ファイルのレコード関数

以下のレコード関数を使用して、ICF ファイルを処理します。

- _Racquire()
- _Rclose()
- _Rdevatr()
- _Rfeod()
- _Rformat()
- _Rindara()
- _Riofbk()
- _Ropen()
- _Ropnfbk()
例:

以下の例では、ソース・プログラムからユーザー ID とパスワードを取得し、それをターゲット・プログラ
ムに送信します。ターゲット・プログラムはユーザー ID とパスワードにエラーがないかチェックし、
ソース・プログラムに応答を送信します。

注: この例を実行するには、ターゲット・プログラム T1520TGT がリモート・システム上に存在している
必要があります。また、プログラム T1520ICF のあるソース・システムとプログラム T1520TGT のあるタ
ーゲット・システムの間の通信回線がアクティブでなければなりません。拡張プログラム間通信機能
(APPC) も必要となります。

1. 物理ファイル T1520DDA を作成するには、次のように入力します。
   CRTPF FILE(MYLIB/T1520DDA) SRCFILE(QCPPLE/QADDSSRC)

   A           UNIQUE
   A     R PASSWRDF
   A     USERID     8A
   A     PASSWRD    10A
   A     K USERID

   図 7-39. T1520DDA — パスワードおよびユーザー ID のための DDS ソース

2. 以下に示す DDS ソースを使用して ICF ファイル T1520DDB を作成するには、次のように入力しま
   す。
   CRTICFF FILE(MYLIB/T1520DDB) SRCFILE(QCPPLE/QADDSSRC)
   ACQPGMDEV(CAPPC2)

   A     R SNDPASS
   A     FLDI     18A
   A     R CHKPASS
   A     FLDI     1A
   A     R EVOKPGM
   A     EVOKE(MYLIB/T1520TGT)
   A     SECURITY(2 'PASSWRD' +
   A                3 'USRID')

   図 7-40. T1520DDB — パスワードおよびユーザー ID を送信するための DDS ソース

3. 以下に示す DDS ソースを使用して ICF ファイル T1520DDC を作成するには、次のように入力しま
   す。
   CRTICFF FILE(MYLIB/T1520DDC) SRCFILE(QCPPLE/QADDSSRC) ACQPGMDEV(CAPPC1)

7-62 IBM i: ILE C/C++ プログラマーの手引き
ファイル・システムおよび装置の処理 7-63

4. システム内装置 INTRAC を作成します。コマンド行から、次のように入力します。
   CRTDEVINTR DEVD(INTRAC) RMTLOCNAME(INTRAC) ONLINE(*NO)

5. システム内装置 INTRAC をオンに構成変更します。コマンド行から、次のように入力します。
   VRYCFG CFGOBJ(INTRAC) CFGTYPE(*DEV) STATUS(*ON) RANGE(*OBJ)

6. ICF ファイル T1520DDB のプログラム装置項目を追加するには、次のように入力します。
   ADDICFDEVFILE(MYLIB/T1520DDB) PGMDEV(CAPPC2) RMTLOCNAME(CAPPC1)
   MODE(CAPPCMOD)

7. ICF ファイル T1520DCC のプログラム装置項目を追加するには、次のように入力します。
   ADDICFDEVFILE(MYLIB/T1520DCC) PGMDEV(CAPPC1) RMTLOCNAME(+REQUESTER)
   MODE(CAPPCMOD)

8. 以下に示すソースを使用してプログラム T1520ICF を作成するには、次のように入力します。

   /* This program sends a userid and password to a target program */
   /* on another system. The target program returns the userid and */
   /* password. This program verifies the returned values. */
   #include <stdio.h>
   #include <recio.h>
   #include <stdlib.h>
   #include <string.h>
   #define ID_SIZE 8
   #define PASSWD_SIZE 10
   #define RCD_SIZE ID_SIZE + PASSWD_SIZE
   #define ERROR '2'
   #define VALID '1'

   _RFILE *fp;
   void ioCheck(char *majorRc)
   {
     if ( memcmp(majorRc, "00", 2) != 0 )
     {
       printf("Fatal I/O error occurred, program ends\n");
       _Rclose(fp);
       exit(1);
     }
   }

   int main(void)
   {
     _RIOFB_T *fb;
     char idPass[RCD_SIZE];
     char buf[RCD_SIZE + 1];
     char passwordCheck=ERROR;

     /* Open the source file T1520DDB. */
     if ( (fp = _Ropen("*LIBL/T1520DDB", "ar+")) == NULL )
     {
       printf("Could not open SOURCE ICF file\n");
       exit(2);
     }

     /* Start the target program T1520TGT. */
     _Racquire(fp, "DEVI"); /* acquire device */
     _Rformat(fp, "EVOKPGM");
     fb = _Rewrite(fp, ",", 0);
Get the user-id and password. */
membzer(idPass, ' ', RCD_SIZE);
printf("Enter user-id (maximum 8 characters):\n");
scanf("%s", buf);
memcpy(idPass, buf, strlen(buf));
printf("Enter password (maximum 10 characters):\n");
scanf("%s", buf);
memcpy(idPass + ID_SIZE, buf, strlen(buf));
/* Send data to the TARGET program T1520TGT. */
_Rformat(fp, "SNDPASS");
fb = _Rwrite(fp, idPass, RCD_SIZE);
ioCheck(fb->sysparm->_Maj_Min.major_rc);
/* Receive data from TARGET program T1520TGT. */
_Rformat(fp, "CHKPASS");
fb = _Rreadn(fp, &passwordCheck, 1, __DFT);
ioCheck(fb->sysparm->_Maj_Min.major_rc);
/* If a problem, such as a communications line is down, occurs in the */
/* TARGET program, then end the program. */
/* Otherwise, print the password verification. */
if ( passwordCheck == ERROR )
{
    _Rclose(fp);
    exit(3);
}
else if ( passwordCheck == VALID )
{
    printf("Password valid\n");
}
else
{
    printf("Password invalid\n");
}
_Rclose(fp);
return(0);

_Ropen() 関数はレコード・ファイル T1520DDB をオープンします。_Rformat() 関数は、ファイル T1520DDB 内のレコード様式 EVOKPGM にアクセスします。T1520DDB 内の EVOKE ステートメントによりターゲット・プログラム T1520TGT を呼び出されます。_Rformat() 関数は、ファイル T1520DDB 内のレコード様式 SNDPASS にアクセスします。ユーザー ID とパスワードがターゲット・プログラム T1520TGT に送信されます。_Rformat() 関数は、ファイル T1520DDB 内のレコード様式 CHKPASS にアクセスします。受信されたパスワードとユーザー ID は、その後検査されます。

以下のソースを使用してプログラム T1520TGT を作成するには、次のように入力します。

CRTBNDC PGM(MYLIB/T1520TGT) SRCFILE(QCPPLE/QACSRC)

/* This program checks the user id and password. */
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>
#define ID_SIZE 8
#define PASSWD_SIZE 10
#define RCD_SIZE ID_SIZE + PASSWD_SIZE
#define ERROR '2'
#define VALID '1'
#define INVALID '0'
int main(void)
{
    _RFILE *icff;
    _RFILE *pswd;
    _RIOFB_T *fb;
    char rcv[RCD_SIZE];
    char pwrdd[RCD_SIZE];
    char vry;

    /* Open the TARGET file T1529DDC. */
    if ( (icff = _Ropen("QGPL/T1520DDC", "ar+")) == NULL )
    {
        printf("Could not open TARGET icf file T1520DDC\n");
        exit(1);
    }

    /* Open the PASSWORD file T1520DDA. */
    if ( (pswd = _Ropen("QGPL/T1520DDA", "rr")) == NULL )
    {
        printf("Could not open PASSWORD file T1520DDA\n");
        exit(2);
    }

    /* Read the information from the SOURCE program T1520ICF. */
    _Racquire(icff, "DEV1");
    _Rformat(icff, "RCVPASS");
    fb = _Rreadn(icff, &rcv, RCD_SIZE, __DFT);

    /* Check for errors and send response to SOURCE program. */
    if ( memcmp(fb->sysparm->_Maj_Min.major_rc, "00", 2) != 0 )
    {
        vry = ERROR;
    }
    else
    {
        fb = _Rreadk(pswd, &pwrdd, RCD_SIZE, __DFT, &rcv, ID_SIZE);
        if ( fb->num_bytes == RCD_SIZE &&
             memcmp(pwrdd + ID_SIZE, rcv + ID_SIZE, PASSWD_SIZE) == 0 )
        {
            vry = VALID;
        }
        else
        {
            vry = INVALID;
        }
    }

    _Rformat(icff, "VRYPASS");
    _Rwrite(icff, &vry, 1);
    _Rclose(icff);
    _Rclose(pswd);
    return(0);
}

_Ropen() 関数はファイル T1520DDC をオープンします。_Ropen() 関数はパスワード・ファイル T1520DDDA をオープンします。_Rformat() 関数は、ファイル T1520DDC 内のレコード様式 RCVPASS にアクセスします。_Rreadn() 関数は、ソース・プログラム T1520ICF からパスワードと ユーザー ID を読み取ります。エラーのチェックが行われ、応答がソース・プログラム T1520ICF に 送信されます。

10. プログラム T1520ICF を実行するには、次のように入力します。
CALL PGM(MYLIB/T1520ICF)

プログラムの呼び出し後、ユーザー ID とパスワードを入力できます。パスワードが正しければ、表示装置に Password valid と表示されます。パスワードが正しくない場合は、Password invalid と表示されます。

出力は以下のようになります。

Password valid
実行キーを押して端末セッションを終了してください。

プリンター・ファイルの使用
プリンター・ファイルには、プログラム記述ファイル (CRTPRTF コマンドで SRCFILE(*NONE) を指定) または外部記述ファイルを使用してアクセスできます。オブジェクト・タイプは *FILE です。ADTS/400:拡張印刷機能 の資料に、プリンター・ファイルに関する情報があります。

プログラム記述ファイルでは、先頭文字用紙制御 (FCFC) が許可されます。これを使用するには、プリンター・ファイルの各データ・レコードの先頭位置に先頭文字用紙制御コードを組み込みます。プリンター・ストリーム・ファイルと fwrite() 関数を使用する必要があります。

外部記述プリンター・ファイルの処理には、以下のいずれかを使用します。
- SEU または CODE/400 エディターによる DDS。
- 報告書設計ユーティリティー (RLU) または DSU WYSIWYG ツール。

印刷出力ファイルの入出力に関する考慮事項
先頭文字書式制御を使用したい場合は、プログラム記述の印刷出力ファイルを使用する必要があります。

印刷出力ファイルをバイナリー・ストリーム・ファイルとしてオープンする
IBM i の印刷出力ファイルをレコード単位の処理にバイナリー・ストリーム・ファイルとしてオープンするには、fopen() 関数を以下のいずれかのモードで使用してください。

- wb
- ab

注: 印刷出力ファイルを作成する唯一の方法は、CRTPRTF コマンドを使用することです。fopen() 関数を使用する場合で、かつ印刷出力ファイルが存在しない場合、物理データベース・ファイルが作成されます。

有効なキーワード・パラメーターは次のとおりです。

- type
- lrecl
- indicators
- recfm

印刷出力ファイルのバイナリー・ストリーム関数: 以下のバイナリー・ストリーム関数を使用して、印刷出力ファイルを処理します。

- fclose()
- fopen()
- freopen()
印刷出力ファイルをレコード・ファイルとしてオープンする
IBM i の印刷出力ファイルをレコード・ファイルとしてオープンするには,_Ropen() 関数を以下のいずれかのモードで使用してください。
- wr
- ar

有効なキーワード・パラメーターは次のとおりです。
- lrecl
- indicators
- riofb
- secure

印刷出力ファイルのレコード関数
以下のレコード関数を使用して、印刷出力ファイルを処理します。
- _Rclose()
- _Rfeof()
- _Rformat()
- _Rindara()
- _Riofbk()
- _Ropen()
- _Ropnfbk()
- _Rupfb()
- _Rwrite()

例：
以下の例では、プログラム記述の印刷出力ファイルで先頭文字式制御を使用しています。従業員の名前と
通し番号が物理ファイルから読み取られ、印刷出力ファイルに書き込まれます。
1. 印刷出力ファイル T1520FCP を作成するには、次のように入力します。
   CRTPTF FILE(MYLIB/T1520FCP) CTLCHAR(*FCFC) CHLVAL((1 (13)))
2. 物理ファイル T1520FCI を作成するには、次のように入力します。
   CRTPF FILE(MYLIB/T1520FCI) RCDLEN(30)
3. 名前と通し番号を次のように T1520FCI に入力します。
   Jim Roberts 1234567890
   Karen Smith 2314563567
   John Doe 5646357324
4. 以下に示すソースを使用してプログラム T1520FCF を作成するには、次のように入力します。
   CRTBNDC PGM(MYLIB/T1520FCF) SRCFILE(QCPPLE/QACSRC)
/* This program illustrates how to use a printer stream file, the */
/* _fwrite() function and the first character forms control. */
#include <stdio.h>
#include <string.h>
#define BUF_SIZE 53
#define BUF_OFFSET 20

int main(void)
{
    FILE  *dbf;
    FILE  *prtf;
    char buf [BUF_SIZE];
    char tmpbuf [BUF_SIZE];

    /* Open the printer file using the first character forms control. */
    /* recfm and lrecl are required. */
    prtf = fopen (*LIBL/T1520FCP", "wb type=record recfm=fa lrecl=53" );
    dbf = fopen (*LIBL/T1520FCI", "rb type=record blksize=0" );

    /* Print out the header information. */
    memset ( buf, ' ', BUF_SIZE );

    /* Use channel value 1. */
    strncpy ( buf, "1 EMPLOYEE INFORMATION",47 );
    fwrite ( buf, 1, BUF_SIZE, prtf );

    /* Use single spacing. */
    strncpy ( buf," --------------------",47 );
    fwrite ( buf, 1, BUF_SIZE, prtf );

    /* Use triple spacing. */
    strncpy ( buf, "- NAME SERIAL NUMBER"
        ,BUF_SIZE );
    fwrite ( buf,1, BUF_SIZE, prtf );
    strncpy ( buf, " ---- -------------"
        ,BUF_SIZE );
    fwrite ( buf,1, BUF_SIZE, prtf );

    /* Print out the employee information. */
    while ( fread ( tmpbuf, 1, BUF_SIZE, dbf ))
    {
        memset ( buf, ' ', BUF_SIZE );

        /* Use double spacing. */
        buf[0] = '0';
        strncpy ( buf + BUF_OFFSET, tmpbuf, strlen(tmpbuf) );
        fwrite ( buf, 1, BUF_SIZE, prtf );
    }
    fclose ( prtf );
    fclose ( dbf );
}

図7-42. T1520FCF 一大頭文字書式制御を使用するための ILE C ソース

fopen() 関数は、レコード単位の処理を使用して印刷装置ストリーム・ファイル T1520FCP をオープンします。また, fopen() 関数は、レコード単位の処理のために物理ファイル T1520FCI もオープンします。 strncpy() 関数はレコードを印刷バッファにコピーします。fwrite() 関数は従業員レコードを印刷出力します。

5. プログラム T1520FCF を実行するには、次のように入力します。
CALL PGM(MYLIB/T1520FCF)

出力ファイルは以下のようにになります。
### EMPLOYEE INFORMATION

<table>
<thead>
<tr>
<th>NAME</th>
<th>SERIAL NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim Roberts</td>
<td>1234567890</td>
</tr>
<tr>
<td>Karen Smith</td>
<td>2314563567</td>
</tr>
<tr>
<td>John Doe</td>
<td>5646357324</td>
</tr>
</tbody>
</table>

印刷出力ファイルは以下のようになります。

### EMPLOYEE INFORMATION

<table>
<thead>
<tr>
<th>NAME</th>
<th>SERIAL NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim Roberts</td>
<td>1234567890</td>
</tr>
<tr>
<td>Karen Smith</td>
<td>2314563567</td>
</tr>
<tr>
<td>John Doe</td>
<td>5646357324</td>
</tr>
</tbody>
</table>

**テープ・ファイルへの書き込み**

テープ・ファイルにレコードを書き込むことができます。テープ・ファイルとは、テープ装置に使用される装置ファイルです。オブジェクト・タイプは *FILE です。

**テープ・ファイルの入出力に関する考慮事項**

ILE C/C++ プログラムは、順次処理する方法でのみ、テープ・ファイルを処理することができます。ILE C/C++ プログラムは、プログラム記述ファイルとしてのみテープ・ファイルを処理することができます。

**テープ・ファイルをバイナリー・ストリーム・ファイルとしてオープンする**

IBM i のテープ・ファイルをレコード単位の処理用にバイナリー・ストリーム・ファイルとしてオープンするには、fopen() 関数を以下のいずれかのモードで使用してください。

- rb
- wb
- ab

注: テープ・ファイルを作成する唯一の方法は、CRTTAPF コマンドを使用することです。fopen() 関数を使用する場合で、かつテープ・ファイルが存在しない場合、物理データベース・ファイルが作成されます。

有効なキーワード・パラメーターは次のとおりです。

- type
- lrecl
- recfm
- blksize
バイナリー・ストリーム・テープ・ファイルの入出力に関する考慮事項:

バイナリー・ストリーム・テープ・ファイルのブロック化: ご使用のプログラムでテープ・ファイルを処理する場合、レコードをブロック化するとパフォーマンスを向上できる可能性があります。

注: fopen() 関数の blksize パラメーターで指定する値は、CRTTAPF または CHGTAPF コマンドで指定した値をオーバーライドします。それでも OVRTAPF コマンドで BLKLEN パラメーターをオーバーライドすることは可能です。

BLKLEN または blksize に 0 を指定すると、システムがブロック・サイズを計算します。いずれかのパラメーターに、0 から 32767 までの文字数の値を指定することができます。

テープ・ファイルのバイナリー・ストリーム関数
以下のバイナリー・ストリーム関数を使用して、テープ・ファイルを処理します。

- fclose()
- fopen()
- fread()
- freopen()
- fwrite()

テープ・ファイルをレコード・ファイルとしてオープンする
IBM i のテープ・ファイルをレコード・ファイルとしてオープンするには、_Ropen() 関数を以下のいずれかのモードで使用してください。

- rr
- wr
- ar

有効なキーワード・パラメーターは次のとおりです。

- blkrcd
- lrecl
- secure
- riofb

レコード・テープ・ファイルの入出力に関する考慮事項:

_Rfeod の使用: _Rfeod() 関数は、テープ・レコード・ファイルを使用した入出力操作のためにオープンされているファイルに対して有効です。入力操作に対して、この関数は EOF を返し、ファイルの最後のボリュームにテープを配置します。出力操作に対しては、バッファされていないデータがすべてテープに書き込まれるよう制限します。

_Rfeov の使用: _Rfeov() 関数は、入出力操作のためにオープンされているテープ・レコード・ファイルに対して有効です。入力操作に対して、この関数は EOF をシグナル通知し、次のボリュームにテープを配置します。出力操作に対しては、書き込みされていないデータがテープに書き込まれるよう制限します。ボリューム終了トレーラーがテープに書き込まれます。これにより、このトレーラーの後にはデータの書き込みを行うことができないことを意味します。_Rfeov() 関数の後に行われる書き込み操作はすべて、新規ボリュームで実行されます。
レコード・テーブル・ファイルのブロック化： ご使用のプログラムでテーブル・ファイルを処理する場合、入出力操作をブロック化するとパフォーマンスを向上できる可能性があります。レコードをブロック化するには、_Ropen() 関数で blkrcd=Y キーワードを使用してください。

テープ・ファイルのレコード関数
以下のレコード関数を使用して、テープ・ファイルを処理します。

- _Rclose()
- _Rfeed()
- _Rfeov()
- _Riofbk()
- _Ropen()
- _Ropfbk()
- _Rreadn()
- _Rupfb()
- _Rwrite()

例：
以下の例は、テープ・ファイルへの書き込み方法を示しています。

1. テープ・ファイル T1520TPF を作成するには、次のように入力します。CRTTAPF FILE(MYLIB/T1520TPF) DEV(TAP01) SEQNBR(*END)
 LABEL(CSOURCE) FILETYPE(*SRC)
2. メンバー CSOURCE を使用してソース物理ファイル QCSRC を作成するには、次のように入力します。
 CRTSRCPF FILE(MYLIB/QCSRC) MBR(CSOURCE)

CRTSRCPF コマンドは、MYLIB のメンバー CSOURCE を使用して物理ファイル QCSRC を作成します。以下のステートメントがテーブル・ファイルにコピーされます。

```c
/* This program SQITF is called by the command SQUARE. This */
/* program then calls another ILE C program SQ to perform */
/* calculations and return a value. */
#include <stdio.h>
#include <decimal.h>
#pragma linkage(SQ, OS) /* Tell compiler this is external call, */
/* do not pass by value. */
int SQ(int);
int main(int argc, char *argv[])
{
  int *x;
  int result;
  x = (int *) argv[1];
  result = SQ(*x);
  /* Note that although the argument is passed by value, the compiler */
  /* copies the argument to a temporary variable, and the pointer to */
  /* the temporary variable is passed to the called program SQ. */
  printf("The SQUARE of %d is %d\n", *x, result);
}
```

図7-43. プログラム T1520TAP のサンプル・ソース・ステートメント

3. 以下に示すソースを使用してプログラム T1520TAP を作成するには、次のように入力します。
 CR TNDC PGM(MYLIB/T1520TAP) SRCFILE(QCPPLE/QACSRC)
/* This program illustrates how to write to a tape file. */

#include <stdio.h>
#include <string.h>
#include <recio.h>
#include <stdlib.h>
#define RECLEN 80

int main(void)
{
    _RFILE *tape;
    _RFILE *fp;
    char buf [92];
    int i;

    /* Open the source physical file containing the C source. */
    if (( fp = _Ropen ( "*LIBL/QCSRC(CSOURCE)", "rr blkrcd=y" )) == NULL )
    { printf ( "could not open C source file\n" );
        exit (1);
    }

    /* Open the tape file to receive the C source statements */
    if (( tape = _Ropen ( "*LIBL/T1520TPF", "wr lrecl=92 blkrcd=y" )) == NULL )
    { printf ( "could not open tape file\n" );
        exit (2);
    }

    /* Read the C source statements, find their sizes */
    /* and add them to the tape file. */
    while (( _Rreadn ( fp, buf, sizeof(buf), __DFT )) -> num_bytes != EOF )
    { memmove ( buf, buf+12, RECLEN );
        _Rwrite ( tape, buf, RECLEN );
    }

    _Rclose ( fp);
    _Rclose ( tape);
    return(0);
}

図7-44. T1520TAP — テープ・ファイルへの書き込みのための ILE C ソース

このプログラムはソース物理ファイル T1520TPF をオープンします。_Ropen() 関数ファイル QCSRC には、ソース・ステートメントを持つメンバー CSOURCE が含まれています。_Ropen() 関数は、テープ・ファイル T1520TPF をオープンして C ソース・ステートメントを受信します。_Rreadn() 関数は C ソース・ステートメントを読み取り、そのサイズを検出し、それらのステートメントをテープ・ファイル T1520TPF に追加します。

4. プログラム T1520TAP を実行するには、次のように入力してください。

CALL PGM(MYLIB/T1520TAP)

プログラムの実行後、テープ・ファイルには CSOURCE からのソース・ステートメントが含まれます。

ディスケット・ファイルへの書き込み
ディスケット・ファイルにレコードを書き込むことができます。
ディスクケット・ファイルは、ディスクケット装置に使用される装置ファイルです。オブジェクト・タイプは *FILE です。

ディスクケット・ファイルの入出力に関する考慮事項
ディスクケット装置には、プログラム記述ファイルを使用してのミアクセスできます。ILE C/C++ プログラムは、順次処理する方法でのみ、ディスクケット・ファイルを処理することができます。

追加モードを使用したファイルのクリアまたはファイルのオープンの概念は、ディスクケット・ファイルには適用されません。

ファイルがオープンされる際には、ディスクケット・ファイルのラベル名が必要となります。このラベル名は、ディスクケット・ファイル一時変更 (OVRDKTF) コマンドを使用して指定できます。

ディスクケット・ファイルが入力のためにオープンされている場合で、かつ

- lrecl パラメーターが指定されていないかゼロに指定されている場合、ディスクケット上の名前のデータ・ファイル・ラベルのレコード長を使用して、読み取り対象のレコードの長さが決まります。
- lrecl パラメーターがディスクケット・ファイルのレコードの長さを超える場合、読み取り対象のレコードはブランクで埋め込まれます。
- lrecl パラメーターがディスクケット・ファイルのレコードの長さに満たない場合、読み取り対象のレコードは切り捨てられます。
- ディスクケット・ファイルのファイル・タイプがソース・ファイルである場合、各レコードの先頭に日時と開始順序番号が追加されます。レコードの書き込み時にはこれらのデータを削除する必要があり、open ステートメントの lrecl パラメーターに 12 バイトを追加しなければなりません。

注: 出力の結果必ずディスクケット・ファイルに対する入出力操作になるとは限りません。入出力バッファーヒには、ディスクケットのトラック全体を満たすのに十分なデータが含まれていなければなりません。

出力のためにディスクケット・ファイルをオープンする場合、データ・ファイルの満了日がシステム日付と同じかそれより前であれば、そのディスクケット上に存在するすべてのファイルが削除されます。

ディスクケット・ファイルをバイナリー・ストリーム・ファイルとしてオープンする

IBM i のディスクケット・ファイルをレコード単位の処理用にバイナリー・ストリーム・ファイルとしてオープンするには、fopen() 関数を以下のいずれかのモードで使用してください。

- rb
- wb

注: ディスクケット・ファイルを作成する唯一の方法は、CRTDKTF コマンドを使用することです。fopen() を使用する場合で、かつディスクケット・ファイルが存在しない場合、物理データベース・ファイルが作成されます。

有効なキーワード・パラメーターは次のとおりです。

- type
- lrecl
- blksize

バイナリー・ストリーム・ディスクケット・ファイルの入出力に関する考慮事項:

バイナリー・ストリーム・ディスクケット・ファイルのブロック化: ご使用のプログラムでディスクケット・ファイルを処理する場合、入出力操作をブロック化するとパフォーマンスを向上できる可能性があります。
ディスケット・ファイルのバイナリー・ストリーム関数
以下のバイナリー・ストリーム関数を使用して、ディスケット・ファイルを処理します。

- fclose()
- fopen()
- fread()
- freopen()
- fwrite()

ディスケット・ファイルをレコード・ファイルとしてオープンする
IBM i のディスケット・ファイルをレコード・ファイルとしてオープンするには、_Ropen()関数を以下のいずれかのモードで使用してください。

- rr
- wr

有効なキーワード・パラメーターは次のとおりです。

- blkrcd
- lrecl
- secure
- riofb

レコード・ディスケット・ファイルの入出力に関する考慮事項: _Rfeod()関数は、入出力操作のためにオープンされているディスケット・レコード・ファイルに対して有効です。この関数により、EOF がシグナル通知されます。出力操作に対して、この関数がデータの書き込みを行うことはありません。

レコード・ディスケット・ファイルの読み取りおよび書き込み: ディスケット・ファイルから読み取りを行う場合、そのディスケット・ファイル内の次の順序のレコードが処理されます。ディスケット・ファイルの読み取りには _Rreadn()関数を、ディスケット・ファイルの書き込みには _Rwrite()関数を使用してください。

レコード・ディスケット・ファイルのブロック化: ご使用のプログラムでディスケット・ファイルを処理する場合、レコードをブロック化するとパフォーマンスを向上できる可能性があります。_Ropen()に blkrcd=Y を指定した場合、システムは、プログラムに 1 つのブロックとして転送されるレコードの数を計算します。

ディスケット・ファイルのレコード関数
以下のレコード関数を使用して、ディスケット・ファイルを処理します。

- _Rclose()
- _Rfeod()
- _Riofbk()
- _Ropen()
- _Ropfbk()
- _Rreadn()
- _Rupfb()
例:
以下の例は、ディスケット・ファイルへの書き込み方法を示しています。
1. ディスケット・ファイル T1520DKF を作成するには、次のように入力します。

```
CRTDKT FILE(MYLIB/T1520DKF) DEV(DKT02) LABEL(FILE1)
EXCHTYPE(*I) SPOOL(*NO)
```

2. 次のように入力します。

```
CRTPF FILE(MYLIB/T1520DDI) SRCPFLE(QCPPLE/QADDSSRC)
SHARE(*YES)
```

以下の DDS ソースを使用して物理ファイル T1520DDI を作成するには、次のように入力します。

```
A          R CUST
A          NAME       20A
A          AGE        3B
A          DENTAL     6B
```

3. データベース・ファイル T1520DDI に以下のレコードを入力します。

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dave</td>
<td>35</td>
<td>350</td>
</tr>
<tr>
<td>Mary</td>
<td>54</td>
<td>444</td>
</tr>
<tr>
<td>Mike</td>
<td>25</td>
<td>545</td>
</tr>
<tr>
<td>Alex</td>
<td>32</td>
<td>512</td>
</tr>
</tbody>
</table>

4. DENTAL フィールドの値が 400 より大きいレコードのみを選択するには、次のように入力します。

```
OPNQRYF FILE((MYLIB/T1520DDI)) QRYSLT('DENTAL +GT 400') OPNSCOPE(*JOB)
```

5. 以下に示すソースを使用してプログラム T1520DSK を作成するには、次のように入力します。

```
CRTBND CPG(MYLIB/T1520DSK) SRCPFLE(QCPPLE/QACSRC)
```
/* This program illustrates how to write to a diskette file. */

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

#define BUF_SIZE 30

int main(void)
{
    _RFILE *dktf;
    _RFILE *dbf;
    char buf [BUF_SIZE];

    /* Open the diskette file */
    if (( dktf = _Ropen ( "*LIBL/T1520DKF", "wr blkrcd=y lrec=100" )) == NULL 
    {
        printf ( "DISKETTE file did not open \n" );
        exit ( 1 );
    }

    /* Open the database file. */
    if ( ( dbf = _Ropen ( "*LIBL/T1520DDI", "rr" ) ) == NULL)
    {
        printf ( "DATABASE file did not open\n" );
        exit ( 2 );
    }

    /* Copy all the database records meeting the OPMQRF selection */
    /* criteria to the diskette file. */
    while (( _Rreadn ( dbf, buf, BUF_SIZE,__DFT) ) -> num_bytes != EOF )
    {
        _Rwrite ( dktf, buf, BUF_SIZE );
    }
    _Rclose ( dktf );
    _Rclose ( dbf );
}

图7-45. T1520DSK — ディスクケット・ファイルにレコードを書き込むための ILE C ソース

_Ropen() 関数は、ディスクケット・ファイル T1520DKF およびデータベース・ファイル T1520DDI をオープンします。_Rreadn() 関数はすべてのデータベース・レコードを読み取ります。_Rwrite() 関数は、DENTAL フィールドの値が 400 をを超えるデータベース・レコードをすべてディスクケット・ファイル T1520DKF にコピーします。

6. プログラム T1520DKSK を実行するには、次のように入力します。

CALL PGM(MYLIB/T1520DSK)

ディスクケット・ファイルへの出力は以下のようにになります。

<table>
<thead>
<tr>
<th>Name</th>
<th>No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary Smith</td>
<td>444</td>
</tr>
<tr>
<td>Mike Tomas</td>
<td>545</td>
</tr>
<tr>
<td>Alex Michaels</td>
<td>512</td>
</tr>
</tbody>
</table>

プログラムの実行後、ディスクケット・ファイルには、選択基準を満たすレコードのみが含まれます。
保管ファイルの使用
保管ファイルは、ディスクまたはテープに保存されるデータを保管したり、ネットワークを介して送信されたオブジェクトを受信したりするために使用できる、補助ストレージに割り振られたファイルです。オブジェクト・タイプは *FILE です。マニュアル「バックアップおよび回復」に、保管ファイルに関する情報が含まれています。

保管ファイルの入出力に関する考慮事項
ILE C/C++ プログラムは、順次処理する方法であるため、保管ファイルを処理することができます。読み取りまたは書き込みされるすべてのレコードの長さは 528 文字でなければなりません。別の保管ファイルに書き込まれたレコードを、ILE C/C++ プログラムによって変更することはできません。

保管ファイルをバイナリー・ストリーム・ファイルとしてオープンする
IBM i の保管ファイルをレコード単位の処理用にバイナリー・ストリーム・ファイルとしてオープンするには、fopen() 関数を以下のいずれかのモードで使用してください。

- rb
- wb
- ab

注: 保管ファイルを作成する唯一の方法は、CRTSAVF コマンドを使用することです。 fopen() 関数を wb モードまたは ab モードで使用する場合で、かつ保管ファイルが存在しない場合、物理データベース・ファイルが作成されます。

有効なキーワード・パラメーターは次のとおりです。
- lrecl
- type

バイナリー・ストリーム保管ファイルの入出力に関する考慮事項
バイナリー・ストリーム保管ファイルに関しては、特に考慮事項はありません。

保管ファイルのバイナリー・ストリーム関数
以下のバイナリー・ストリーム関数を使用して、保管ファイルを処理します。

- fclose()
- fopen()
- fread()
- freopen()
- fwrite()

保管ファイルをレコード・ファイルとしてオープンする
IBM i の保管ファイルをレコード・ファイルとしてオープンするには、_Ropen() 関数を以下のいずれかのモードで使用してください。

- rr
- wr
- ar

有効なキーワード・パラメーターは次のとおりです。

ファイル・システムおよび装置の処理 7-77
レコード保管ファイルの入出力に関する考慮事項
保管ファイルが入力のためにオープンされている場合、_Rfeod() 関数は EOF をプログラムに戻します。
保管ファイルが出力のためにオープンされている場合、_Rfeod() 関数により、そのファイルに書き込まれたすべてのデータが補助記憶域へ強制書き込みされます。この関数の呼び出し後に保管ファイルに対する読み取りまたは書き込みを継続したい場合は、いったんファイルをクローズしてから再度オープンする必要があるります。

保管ファイルのレコード関数
以下のレコード関数を使用して、保管ファイルを処理することができます。
- _Rclose()
- _Rfeod()
- _Riofbk()
- _Ropen()
- _Ropnfbk()
- _Rreadn()
- _Rupfb()
- _Rwrite()
IBM i 機能を使用した処理

このトピックでは、以下を行うための方法について説明します。

- ILE 制御境界および例外処理について理解する
- プログラムでのポインターの使用
- ILE C/C++ 呼び出し規則の使用
- ILE C から ILE C++ へのプログラムの移植
  - 複数言語環境でのプログラムおよびプロシージャーの呼び出し
- C プログラムでのパック 10 進データ型の使用
- C++ プログラムでのパック 10 進データ型の使用
- C++ プログラムでのテンプレートの使用
- デラスベース記憶域の使用
- 実行時型情報を使用したキャスト

プログラムでの例外の処理

このトピックでは以下について説明します。

- ILE 言語固有のエラー処理
- 例外メッセージ
  - システムが例外を処理する方法
  - ストリーム・ファイルおよびレコード・ファイルのエラーの検出
  - ILE 例外ハンドラーの使用
    - 直接モニター・ハンドラー
    - ILE 条件ハンドラー
- C/C++ シグナル・ハンドラーの使用
  - シグナル・ハンドラーと ILE 例外ハンドラーの同時使用
  - ネストされた例外の処理
  - キャンセル・ハンドラーの使用
  - 各種例外処理方法を使用する ILE C ソース・コードの例

ILE 言語固有のエラー処理

8-2 ページの図 8-1 に、OPM 言語固有のエラー処理と比較して、ILE C/C++ 言語固有のエラー処理の複雑さを示します。
図8-1. OPMおよびILEのエラー処理

OPMプログラムの場合、言語固有のエラー処理は、各呼び出しスタック項目に1つ以上の処理ルーチンを提供します。例外がOPMプログラムに送信されると、システムは該当するルーチンを呼び出します。

ILE C/C++プログラムでは、言語固有のエラー処理は、同じ機能に加え、追加タイプの例外ハンドラーを提供します。この追加タイプのハンドラーにより、以下が可能です。

• 例外メッセージを変更して、例外が処理されたことを示す
• 言語固有のエラー処理をバイパスする

ILE C/C++の追加タイプのハンドラーは、以下のとおりです。

• 直接モニター・ハンドラー
• ILE 条件処理プログラム

例外メッセージ

例外メッセージと見なされるメッセージ・タイプは、以下に示すもののみです。

(*ESCAPE)

プログラムが異常終了させたエラーを示します。メッセージ・タイプが*ESCAPEの場合、再開カーソルによって指されている呼び出しスタック項目に機能チェックが送信されます。

(*STATUS)

プログラムが実行中である処理のステータスを記述します。メッセージ・タイプが*STATUSの場合、プログラムは、例外をログに記録することなく再開します。

(*NOTIFY)

呼び出し側プログラムからの修正処置または応答を必要とする状態を記述します。メッセージ・タイプが*NOTIFYの場合、デフォルト応答が送信されます。

機能チェック

プログラムが予期しなかった終了状態を記述します。メッセージが機能チェックである場合、制御境界に対して呼び出しスタックはキャンセルされ、CEE9901が制御境界の呼び出し元に送信されます。
例外メッセージおよびそれがどのように送信されるのかについて詳しくは、「ILE 概念」を参照してください。

システムが例外を処理する方法
IBM i の例外メッセージ体系は、例外処理と条件処理の両方を実装するために使用されます。例外処理と条件処理が相互作用する場合があります。例えば、ユーザ作成条件ハンドラー登録 (CEEHDLR) パインドラ可能 API で登録された ILE 条件ハンドラーは、プログラム・メッセージ送信 (QMHSNDPM) API で送信された例外メッセージを処理するために使用されます。

例外ハンドラーという用語は、IBM i 例外ハンドラーおよび ILE 条件ハンドラーのいずれの意味でも使用されています。

注: 以下について詳しくは、「ILE 概念」を参照してください。
・例外および条件の処理
・例外からの回復

呼び出しメッセージ待ち行列による ILE プロシージャーおよび関数の処理方法
メッセージ待ち行列は、各 IBM i プロセッサ内のすべての呼び出しスタック項目ごとに存在します。これは呼び出しメッセージ待ち行列と呼ばれるものです。呼び出しスタックに新しい項目が現れるとすぐに、システムが新しい呼び出しメッセージ待ち行列を作成します。ILE C/C++ では、プロシージャーの名前により呼び出しメッセージ待ち行列が識別されます。プロシージャー名が固有でない場合は、モジュール名、プログラム名、またはサービス・プログラムを指定することもできます。ハンドラーが例外メッセージを認識しないと判断した場合、その例外メッセージは次にハンドラーにパーソルートすることができます。

ILE で制御境界が例外処理に与えた影響
未処理機能チェックが発生した場合、または HLL 終了 verb が使用されるたびに、アプリケーションの境界を表す呼び出しスタック項目の呼び出し元に制御が渡されます。このような呼び出しスタック項目は、制御境界と呼ばれます。

制御境界は以下のいずれかです。
・直前の呼び出しスタック項目が別の非デフォルト活動化グループにある ILE 呼び出しスタック項目。
・直前の呼び出しスタック項目が OPM プログラムである ILE 呼び出しスタック項目。

制御境界について詳しくは、「ILE 概念」を参照してください。

監視されていない例外および未処理例外
監視されていない例外が発生すると、実行中のプログラムは機能チェックを出し、メッセージをジョブ・ログに送信します。プログラムのモジュールがデバッグ・データを使用して作成されており、デバッグ・モードの場合、ILE ソース・デバッガーには、該当するモジュールが表示され、必要であれば、プログラムはデバッグ・モードに追加されます。

未処理機能チェックが行われると、ILE は、アプリケーションの境界を示している呼び出しスタック項目の呼び出し元に制御を渡します。このような呼び出しスタック項目は、制御境界と呼ばれます。

制御境界について詳しくは、以下を参照してください。
・『ILE で制御境界が例外処理に与える影響』
・ILE 概念
未処理例外のデフォルト処理について詳しくは、「ILE 概念」を参照してください。

未処理例外を含む ILE C ソース・コード例：図 8-2 は、未処理の *ESCAPE 例外を含む ILE C ソース・コードを示しています。

void fred(void)  
{  
    char *p = NULL;  
    *p = 'x';  /* *ESCAPE exception */  
}  
int main(void)  
{  
    fred();  
}  

図 8-2 未処理例外を含む ILE C ソース・コード

図 8-2 では、次のようになります。
1. fred() 関数に例外が送信されます。
2. main() 関数は制御境界です。

注: 説明の目的で、多くの例において main() 関数を制御境界と呼んでいます。プログラムが *NEW 活動化グループで実行される場合、プログラム入日プロシージャー（PEP）がそのプログラムの制御境界となります。
3. fred() 関数には例外ハンドラーがないため、例外は main() にパーケレートされます。
4. main() 関数には例外ハンドラーがなく、main() が制御境界であるため、システムはデフォルト・アクションを実行します。
5. 例外のタイプが *ESCAPE であるため、fred() 関数に機能チェックが送信されます。

注: 他のタイプの例外に対して実行されるアクションについては、8-2 ページの『例外メッセージ』を参照してください。
6. 機能チェックは関数 main() にパーケレートされ、再度デフォルトが実行されます。
7. 例外のタイプが機能チェックであるため、main() および fred() 関数の呼び出しスタック項目は取り消され、CEE9901 例外が関数 main() の呼び出し元に送信されます。

ネストされた例外
ネストされた例外は、別の例外の処理中に発生した例外です。

この場合、次のようになります。
• 最初の例外の処理は一時的に中断されます。
• システムは、処理カーソルおよび再開カーソルの位置などのすべての関連情報を保管します。
• 例外処理は、最新に生成された例外から再開されます。
• 処理カーソルおよび再開カーソルの新しい位置は、システムによって設定されます。
• 新しい例外が適切に処理された後、元の例外の処理活動が正常に戻されます。

ネストされた例外が発生した場合、以下の項目はいずれも依然として呼び出しスタック上に存在しています。
• 元の例外に関連する呼び出しスタック項目
• 元の例外ハンドラーに関連する呼び出しスタック項目
例外処理ループの可能性を低くするために、システムは、ネストされた例外のパーコレーションを元の例外ハンドラーの呼び出しスタック項目で停止します。次に、システムはネストされた例外を機能チェック・メッセージにプロモートし、その機能チェック・メッセージを同じ呼び出しスタック項目にパーコレートします。

ネストされた例外または機能チェック・メッセージが処理されない場合、アプリケーションが異常終了します。この場合には、メッセージ CEE9901 が制御境界の呼び出し元に送信されます。

ネストされた例外の処理中に再開カーソルを移動すると、元の例外を暗黙的に変更することができます。このためには、以下のステップを行います。
1. 元の例外を引き起こした呼び出しスタック項目より前にプロモートされた呼び出しスタック項目に再開カーソルを移動します。
2. 使用中のハンドラーから戻ることによって、処理を再開します。

ストリーム・ファイルおよびレコード・ファイルのエラーの検出
ストリーム・ファイルのエラーを検出するには、以下を検査します。
• 関数の戻り値
• errno 値
• メジャー/マイナー戻りコード (システム例外の場合)

レコード・ファイルのエラーを検出するには、_RIOFB_T 構造体の値を検査します。

注: ILE C/C++ レコード入出力関数によって生成される例外メッセージのリストについては、「ILE C/C++ ランタイム・ライブラリー関数」の表『例外マッピングに対するレコード入出力エラー・マクロ』を参照してください。

ストリーム・ファイルまたはレコード・ファイルのエラーを検出するには、グローバル変数 _EXCP_MSGID を検査します。

関数の戻り値の検査
C と C++ のランタイム・ライブラリー関数の多くには、エラー検査の目的で戻り値が関連付けられています。以下に例を示します。
• _f.eof() 戻数は、ファイルがあるポリュームから次のポリュームに移動された場合、1 を返します。
• fopen() 戻数は、ファイルが正常に開かなかった場合に、NULL を返します。

ILE C/C++ 関数の戻り値については、「ILE C/C++ ランタイム・ライブラリー関数」を参照してください。

各ランタイム・ライブラリー関数が正常に完了したことを確認するには、プログラムで関数の戻り値を検査する必要があります。

例:
以下の図では、fopen() 戻数の戻り値を検査する方法を示します。
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp;
    if (( fp = fopen ( "MYLIB/QCSRC(TEST)" , "ab" ) ) == NULL )
    {
        printf ("Cannot open file QCSRC(TEST)\n");
        exit (99);
    }
}

図8-3. fopen() の戻り値を検査する ILE C ソース

errno 値の検査
<errno.h> ヘッダー・ファイルには、定義されているエラー状態の宣言が含まれています。多数の C 関数は、エラーのタイプに応じて、errno を特定の値に設定します。これらの値は、<errno.h> ヘッダー・ファイルにも定義されています。errno の実装には、関数呼び出しが含まれています。

以下の errno マクロは、IBM i システム例外を示しています。
• EIOERROR: 復旧不能入出力エラーが発生しました。
• EIORECERR: 復旧可能入出力エラーが発生しました。

errno マクロ値のリストについては、「ILE C/C++ ランタイム・ライブラリ関数」を参照してください。

Errno の初期化: errno はライブラリ関数によってリセットされないため、関数を呼び出す前には必ずプログラムで errno を 0 (ゼロ) に初期化する必要があります。チェックしたい関数を呼び出した直後に、errno の値を調べてください。また、エラーが発生した後も、errno をゼロに初期化する必要があります。

Errno 値の表示と出力: ご使用のプログラムで strerror() および perror() 関数を使用して、errno の値を出力することができます。strerror() 関数は、errno と関連付けられたエラー・メッセージ・ストリングをポインタを戻します。 perror() 関数はメッセージを stderr に出力します。関数の呼び出し後、すぐに perror() および strerror() 関数を使用する必要があります。これは、後続の呼び出しによって errno 値が変更されてしまう可能性があるためです。

例: fopen() 関数に対する errno 値のチェック: 次の図に、fopen() 関数に対する errno 値をチェックする方法を示します。

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    FILE *fp;
    errno = 0;
    fp = fopen("Nofile", "r");
    if ( errno != 0 )
    {
        perror("Error occurred while opening file.\n");
        exit(1);
    }
}

図8-4. fopen() に対する errno 値をチェックするための ILE C ソース

8-6 IBM i: ILE C/C++ プログラマの手引き
メジャー/マイナー戻りコードの検査
プログラムで表示装置、ICF、または印刷出力ファイルをストリーム・ファイルとして処理している場合、<stdio.h>で定義されている外部変数 _C_Maj_Min_rcを検査できます。以下の図では、この構造体の定義を示します。

typedef struct _Major_Minor_rc 
{
    char major_rc[2];
    char minor_rc[2];
} _Major_Minor_rc;
extern _Major_Minor_rc _C_Maj_Min_rc;
図8.5. _C_Maj_Min_rc タイプ定義

レコード・ファイルについてのシステム例外の検査
レコード・ファイル・エラーを検出するために、<recio.h>で定義されている _RIOFB_T 構造体の一部の値を検査できます。num_bytes フィールドと sysparm フィールドにはとともに、レコード・ファイル入出力エラーに関する情報が含まれています。

以下の図で、_RIOFB_T 構造体のタイプ定義を示します。

typedef struct { 
    unsigned char *key; /*key; */
    _Sys_Struct_T *sysparm; /*sysparm; */ 1, 3
    unsigned long rrn; /*rrn; */
    long num_bytes; /*num_bytes; */ 1
    short blk_count; /*blk_count; */
    char blk_filled_by; /*blk_filled_by; */
    int dup_key : 1; /*dup_key : 1; */
    int icf_locate: 1; /*icf_locate: 1; */
    int reserved1 : 6; /*reserved1 : 6; */
    char reserved2[20]; /*reserved2[20]; */
} _RIOFB_T;
図8.6. _RIOFB_T タイプ定義

注:
1. プログラムで表示装置、ICF、印刷出力ファイルをレコード・ファイルとして処理している場合、_RIOFB_T 構造体の num_bytes フィールドの値および _RIOFB_T 構造体の sysparm 領域内のメジャー/マイナー戻りコード・フィールドを検査できます。num_bytes フィールドは、入出力操作が成功したかどうかを示します。
2. プログラムでデータベース・ファイルをストリーム・ファイルとして処理している場合は、_RIOFB_T 構造体のいくつかのフィールドの値を検査できます。
3. sysparm フィールドは、表示装置、ICF、または印刷出力ファイルのメジャー/マイナー戻りコードが含まれた構造体を指します。_Sys_Struct_T 構造体の定義を以下に示します。
typedef struct {
  /* System specific information */
  void *sysparm_ext;
  _Maj_Min_rc_T _Maj_Min;
  char reserved1[12];
} _Sys_Struct_T;

図8-7. _Sys_Struct_T タイプ定義

以下の図で、_Maj_Min_rc_T 構造体の定義を示します。

typedef struct {
  char major_rc[2];
  char minor_rc[2];
} _Maj_Min_rc_T;

図8-8. _Maj_Min_rc_T タイプ定義

グローバル変数 _EXCP_MSGID の検査
グローバル変数 _EXCP_MSGID は、ストリームまたはレコード入出力関数が例外を受け取った場合に常に設定されます。<stdlib.h> ヘッダー・ファイルで宣言されているグローバル変数 _EXCP_MSGID には、例外メッセージ ID が含まれています。IBM i 例外発生後の _EXCP_MSGID 設定については、「ILE C/C++ ランタイム・ライブラリー関数」のセクション『ランタイムに関する考慮事項』にある表『例外マッピングに対するレコード入出力エラー・マクロ』を参照してください。

ILE 例外ハンドラーの使用

例外ハンドラーのタイプ
ILE で使用できる例外ハンドラーのタイプは、以下のとおりです。

・ 直接モニター・ハンドラー。これは、#pragma exception_handler ディレクティブで有効にされます。
・ ILE 条件ハンドラー。これにより、バインド可能 API CEEHDLR を使用して、条件ハンドラーを実行時に登録できます。
・ C/C++ signal() 関数。

ILE 直接モニター・ハンドラーの使用
直接モニター・ハンドラー は、例外クラスおよびメッセージ ID に基づいた例外をモニタします。これにより、限定数の C/C++ ソース・ステートメントの例外モニターを直接登録できます。通常、直接モニターが最速のハンドラーです。

プラグマ・ディレクティブの使用: ILE C/C++ では、#pragma exception_handler および #pragma disable_handler ディレクティブにより、直接モニター・ハンドラーが使用可能です。

#pragma exception_handler により、直接モニター・ハンドラーは #pragma exception_handler から
#pragma disable_handler までの範囲において使用可能になります。このとき、その間におけるプログラ
ム・ロジックは考慮されません。

これらのディレクティブを使用する際、ソース・コードに <except.h> ヘッダー・ファイルを組み込む必要があります。

注: #pragma exception_handler ディレクティブがモニターできるのは、例外メッセージ・リストに記載される例外メッセージ・タイプのみです。
直接モニター・ハンドラーは、次のいずれかです。

- `#pragma exception_handler` を含む関数内で定義されたラベルに続くコード。
- 戻りタイプが `void` の関数。この関数は、タイプが `_INTRPT_Hndlr_Parms_T` の構造体を指すポインタという 1 つのパラメーターを使用します。 `_INTRPT_Hndlr_Parms_T` タイプは `<except.h>` ファイルに定義されています。

通信域変数の使用：通信変数は、`#pragma exception_handler` で指定できます。この変数の使用法は、ハンドラーがラベルが関数かによって異なります。ハンドラーがラベルの場合、通信域は、タイプが `_INTRPT_Hndlr_Parms_T` (<except.h> で定義) の標準例外ハンドラー・パラメーター・ブロックのストレージとして使用されます。

構造体 `_INTRPT_Hndlr_Parms_T` の定義は、以下のとおりです。

```c
typedef _Packed struct {
   unsigned int Block_Size; /* Size of the parameter block */
   _INVFLAGS_T TgtFlags;   /* Target invocation flags */
   char reserved[8];      /* reserved */
   _INVPTR Target;        /* Current target invocation */
   _INVPTR Source;        /* Source invocation */
   _SPCPTR ComArea;       /* Communications area */
   char Compare_Data[32]; /* Compare Data */
   char Msg_Id[7];        /* Message ID */
   char reserved1;        /* 1 byte pad */
   _INTRPT_Mask_T Mask;   /* Interrupt class mask */
   unsigned int Msg_Ref_Key; /* Message reference key */
   unsigned short Exception_Id; /* Exception ID */
   char Signal_Class;     /* Internal signal class */
   char Priority;         /* Handler priority */
   short Severity;        /* Message severity */
   char reserved3[4];     /* */
   int Msg_Data.Len;      /* Length of available message data */
   char Mch_Dep_Data[10]; /* Machine dependent data */
   char Tgt.Inv_Type;     /* Invocation type (in MIMCHOBS.H)*/
   _SUSPENDPTR Tgt_Suspend; /* Suspend pointer of target */
   char Ex_Data[48];      /* First 48 bytes of exception data */
} _INTRPT_Hndlr_Parms_T;
```

図 8-9. 構造体 `_INTRPT_Hndlr_Parms_T` の定義

システムは、制御をラベルに渡す前に、構造体を埋めます。例外ハンドラー・ラベル・ブロックに必要なストレージが、`com_area` で定義されているストレージを超過した場合、残りのバイトは切り捨てられます。ハンドラーが関数の場合は、システムは、タイプが `_INTRPT_Hndlr_Parms_T` の構造体へのポインタをそのまま関数に渡します。通信域へのポインタは、構造体内で使用可能です。

直接モニター・ハンドラーのスコープ：直接モニター・ハンドラーは、コンパイル時に、`#pragma exception_handler` ディレクティブと `#pragma disable_handler` ディレクティブ間のコードにスコープ設定されます。例えば、`#pragma exception_handler` ディレクティブは、プログラム・ロジックとは無関係に、コードのブロックにスコープ設定されます。

次の図に例を示します。
volatile int ca=0;
if (ca != 0){
    #pragma exception_handler(my_handler, ca,0,_C2_MH_ESCAPE)
}
else {
    raise(SIGINT); /* Signal will be caught by my_handler */
}
#pragma disable_handler

図8-10. 直接モニター・ハンドラーのスコープを設定するための ILE C ソース

図8-10において、以下のとおりです。

• #pragma exception_handler ディレクティブは、raise() 関数への呼び出し周辺で有効になっています。
• 条件式 if (ca != 0) は、直接モニター・ハンドラーの有効化に影響しません。条件式 if (ca != 0) のロジック・パスが切られることはありません。代わりに、my_handler が有効になります。

例外クラスの使用:

注: #pragma exception_handler ディレクティブがモニターできるのは、例外メッセージのリストに記載された例外メッセージ・タイプのみです。

例外クラス は、例外のタイプ (例えば、*ESCAPE、*NOTIFY、*STATUS、機能チェック) を示し、マシン例外の場合は低レベル・タイプ (pointer-not-valid または divide-by-zero) を示します。

例外が #pragma exception_handler ディレクティブで指定されている 1 つ以上の例外に分類される場合、ハンドラーが制御を得ます。

注:

「ILE C/C++ ランタイム・ライブラリ関数」の『ランタイムに関する考慮事項』セクションに、例外クラスの表が含まれています。

次の図に例を示します。

#include <except.h> 1
/* Just monitor for pointer not valid exceptions */
#pragma exception_handler(eh, 0, _C1_POINTER_NOT_VALID, 0)
/* Monitor for all *ESCAPE messages */
#pragma exception_handler(eh, 0, _C2_MH_ESCAPE) 2
/* Although the following is valid, there is no need to specify */
/* _C1_POINTER_NOT_VALID because it is covered by _C2_MH_ESCAPE */
#pragma exception_handler(eh, 0, _C1_POINTER_NOT_VALID, _C2_MH_ESCAPE) 2
/* To monitor for only specific messages, use the extended form of */
/* pragma exception_handler. */
/* The following #pragma will only monitor for MCH3601 *ESCAPE msg. */
#pragma exception_handler(eh, 0, _C2_MH_ESCAPE, _CTLA_HANDLE, "MCH3601") 2

図8-11. 例外クラスを使用するための ILE C ソース

注:

1. IBM i マシン例外クラスのマクロは、ILE C/C++ 組み込みファイル <except.h> に定義されています。
2. マシン例外をモニターするために、マシン例外クラスを指定するか、すべての *ESCAPE 例外を指定できます。図8-11では、すべてのマシン例外が *ESCAPE タイプ例外にマップされています。メッセージ・タイプが *ESCAPE の場合、再開カーソルによって指されている呼び出しスタック項目に機能チェックが送信されます。
3. class1 および class2 の例外クラス値をモニターできます。class2 の値は、以下の例外クラスの 1 つのみでなければなりません。
   • _C2_MH_ESCAPE
   • _C2_MH_STATUS
   • _C2_MH_NOTIFY
   • _C2_MH_FUNCTION_CHECK

制御処置の指定: #pragma exception_handler ディレクティブでは、例外処理時に実行する制御処置を指定できます。

指定できる 5 つの制御処置 (<except.h> ヘッダー・ファイルで定義) は、以下のとおりです。

_CTLA_INVOKE
この制御処置により、ディレクティブで指定された関数が呼び出されます。例外は処理されません。例外が活動状態のままになるため、QMHCHGEM または ILE 条件処理 API の 1 つを使用して処理する必要があります。

_CTLA_HANDLE
この制御処置により、ディレクティブで指定された関数またはラベルが制御を受け取ります。また、例外が暗黙的に処理され、ログに記録されます。ハンドラーによって制御されるときには、例外は既に活動状態ではありません。

_CTLA_HANDLE_NO_MSG
この制御処置は、例外がログに記録されない点を除いて、_CTLA_HANDLE と同じです。ハンドラーに渡されるパラメーター・ブロックのメッセージ参照キーは、ゼロになります。

_CTLA_IGNORE
この制御処置は、暗黙的に例外を処理してログに記録し、ディレクティブで指定されたハンドラー関数に制御を渡しません。つまり、指定された関数は無視されます。例外は活動状態でなくなり、処理は、例外を引き起こした命令の直後の命令から再開されます。

_CTLA_IGNORE_NO_MSG
この制御処置は、*NOTIFY メッセージがログに記録される点を除いて、_CTLA_IGNORE と同じです。

例:

以下の図では、#pragma exception_handler ディレクティブで制御処置パラメーターを指定する方法を示します。
#include <except.h>
#include <stdio.h>

void myhandler(void) {
  printf("In handler - something's wrong!\n");
  return;
}

int main(void) {
  int *ip;
  volatile int com_area;
  #pragma exception_handler(myhandler, com_area, 0, _C2_ALL, \n  _CTLA_IGNORE) 1
  *ip = 5;
  printf("Passed the exception.\n"); 2
}

図8-12．例外を処理するための ILE C ソース

注:
1. 制御処置 _CTLA_IGNORE により、ハンドラー側数を呼び出すことなく、例外が処理されます。
2. このコードの出力は、メッセージ「Passed the exception」と、その後に続く例外メッセージです。このコードにより、例外メッセージ「MCH3601 (Pointer not set)」が生成されます。

メッセージ ID の指定: #pragma exception_handler ディレクティブでは、1つ以上の固有または汎用メッセージ ID を指定できます。ディレクティブで1つ以上の ID を指定した場合、ID がディレクティブで指定されている ID のいずれかに一致する例外が発生したときにのみ、直接モニター・ハンドラーが有効になります。

ディレクティブでメッセージ ID を指定するには、実行する制御処置を指定する必要があります。例外のクラスは、ディレクティブで指定されているクラスのいずれかでなければなりません。

例:

以下の例では、単一の固有メッセージ MCH3601 のモニターを有効にする #pragma exception_handler ディレクティブを示します。

#pragma exception_handler (myhandler, com_area, 0, _C2_ALL, \n _CTLA_HANDLE, "MCH3601")

例:

以下に、複数の浮動小数点例外のモニターを有効にする #pragma exception_handler ディレクティブの例を示します。

#pragma exception_handler (myhandler, com_area, _C1_ALL, _C2_ALL, \n _CTLA_IGNORE, "MCH1206 MCH1207 MCH1209 MCH1213")

注: 汎用メッセージ ID を指定する機能を使用して、ディレクティブを単純化できます。

例:

以下の例では、ID が MCH12 で開始されている任意の例外に対してモニターが有効になります。

#pragma exception_handler (myhandler, com_area, _C1_ALL, _C2_ALL, \n _CTLA_IGNORE, "MCH1200")

直接モニター・ハンドラーを使用するソース・コードの例: 以下の図では、プログラム MYPGM のソースを示します。

8-12 IBM i: ILE C/C++ プログラマーの手引き
/* This program illustrates how to use direct monitor handlers. */
#include <stdio.h>
#include <signal.h>
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#include <except.h> /* Include except.h even though it is included */
#define FILE_NAME "QTEMP/MY_FILE"
#define RCD_LEN 80
#define NUM_RCD 5
#pragma datamodel(p128)
typedef struct error_code{
    int byte_provided;
    int byte_available;
    char exception_id[7];
    char reserve;
    char exception_data[1];
}error_code_t;
static int handle_flag;
#pragma linkage(QMHCHGEM, OS)
void QMHCHGEM(_INVPTR *, int, unsigned int, char *,
               char *, int, error_code_t *);
/* The signal handler. */
#pragma datamodel(pop)
static void sig_handler(int sig)
{ /* MYPGM *PGM */
    #include <except.h>
    #include <stdio.h>
    void my_handler(_INTRPT_Hndlr_Parms_T * __ptr128 parms);
    void main_handler(_INTRPT_Hndlr_Parms_T * __ptr128 parms);
    void fred(void)
    {
        char *p = NULL;
        #pragma exception_handler(my_handler, 0,0, _C2_MH_ESCAPE)
        *p = 'x'; /* exception */
        #pragma disable_handler
    }
    int main(void)
    {
        #pragma exception_handler(main_handler, 0,0, _C2_MH_ESCAPE)
        fred();
    }
}

図8-13. T1520XH1 — 直接モニター・ハンドラーを使用する ILE C ソース — main()
printf("In signal handler\n");
printf("Exception message ID is %7.7s\n", _EXCP_MSGID);
}
/* The direct monitor handler. */
static void exp_handler(_INTRPT_Hndlr_Parms_T * __ptr128 exp_info)
{
error_code_t error_code;
printf("In direct monitor handler\n");
printf("Exception message ID is %3.3s%04x\n", 
exp_info->Compare_Data, 
(unsigned) exp_info->Exception_Id);
/* Call QMHCHGEM API to handle the exception. */
if ( handle_flag )
{
    error_code.byte_provided = 8;
    QMHCHGEM(&exp_info->Target), 0, exp_info->Msg_Ref_Key, 
    "HANDLE ", ",", 0, &error_code);
}
/* The function to read a file. */
static void read_file(_RFILE *fp)
{
    int i = 1;
    while ( _Rreadn(fp, NULL, RCD_LEN, __DFT)->num_bytes != EOF )
    {
        printf("Read record %d\n", i++);
    }
}
int main(void)
{ 
    _RFILE *fp;
    int i;
    volatile int com;
    char buf[RCD_LEN];
    char cmd[100];
    /* Create a file. */
    sprintf(cmd, "CRTPF FILE(%s) RCDLEN(%d)", FILE_NAME, RCD_LEN);
    system(cmd);
    /* Open the file for write. */
    if ( (fp = _Ropen(FILE_NAME, "wr")) == NULL )
    {
        printf("Open for write fails\n");
        exit(1);
    }
    /* Write some data into the file. */
    memset(buf, '1', RCD_LEN);
    for ( i = 0; i < NUM_RCD; i++ )
    {
        _Rwrite(fp, buf, RCD_LEN);
    }
    _Rclose(fp);
    /* Open the file for the first read. */
    if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        printf("Open for the first read fails\n");
        exit(2);
    }
    /* Read until end-of-file. */
    *Since no signal handler or direct monitor handler is set up,
    *the EOF exception is ignored. The default value for SIGIO is
    */
    SIG_IGN.
    *i = 1;
    printf("The first read starts\n");
    while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
    {
        printf("Read record %d\n", i++);
    }
}
_Rclose(fp);  
printf("The first read finishes
");

/* Set up a direct monitor handler and a signal handler. */
/* Tell the direct monitor handler to handle the exception. */
/* The direct monitor handler (exp_handler) calls the message */
/* handler API QMHCHGEM with the parameter *HANDLE. This marks the */
/* exception as handled. */
/* Use exception classes to handle machine exceptions. */
handle_flag = 1;
#pragma exception_handler(exp_handler, com, 0,   
   _C2_MH_ESCAPE | _C2_MH_NOTIFY | _C2_MH_STATUS)
signal(SIGIO, sig_handler);
/* Open the file for the second read. */
if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{
   printf("Open for the second read fails
");
   exit(3);
}

/* Read until end of file. */
/* When the EOF exception is generated, the direct monitor handler */
/* is called first. Since it marks the exception as handled, */
/* the signal handler is not called. */
i = 1;
printf("The second read starts
");
while ( _Rreadn(fp, buf, RCD_LEN, _DFT)->num_bytes != EOF )
{
   printf("Read record %d\n", i++);
}
_Rclose(fp);
printf("The second read finishes\n");

/* Set up a direct monitor handler and a signal handler. */
/* Set the global variable handle_flag to zero so that the */
/* direct monitor will not handle the exception. */
handle_flag = 0;
#pragma exception_handler(exp_handler, com, 0,   
   _C2_MH_ESCAPE | _C2_MH_NOTIFY | _C2_MH_STATUS)
signal(SIGALL, sig_handler);
/* Open the file for the third read. */
if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{
   printf("Open for the third read fails
");
   exit(4);
}

/* Read until end-of-file. */
/* When the EOF exception is generated, the direct monitor handler */
/* is called first. Since the exception is not marked as */
/* handled, the signal handler is then called. */
i = 1;
printf("The third read starts\n");
while ( _Rreadn(fp, buf, RCD_LEN, _DFT)->num_bytes != EOF )
{
   printf("Read record %d\n", i++);
}
_Rclose(fp);
printf("The third read finishes\n");

/* Set up a direct monitor handler and a signal handler. */
#pragma exception_handler(exp_handler, com, 0,   
   _C2_MH_ESCAPE | _C2_MH_NOTIFY | _C2_MH_STATUS)
signal(SIGIO, sig_handler);
/* Open the file for the fourth read. */
if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{  
    printf("Open for the fourth read fails\n");  
    exit(5);  
}

/* Read until end-of-file. */
/* The EOF exception is generated in function read_file. Since */
/* there is no direct monitor handler for the read_file function, */
/* the signal handler is called. */
/* The direct monitor handler in main() is not called because the */
/* exception was mapped to SIGIO and the signal handler gets called */
/* at function read_file. */
    printf("The fourth read starts\n");
    read_file(fp);
    _Rclose(fp);
    printf("The fourth read finishes\n");
/* Disable the direct monitor handler. */
#pragma disable_handler
}

#include <signal.h>
#include <stdio.h>
/* HANDLERS (created with activation group *CALLER) */
void my_handler(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    return;
}
void main_handler(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    printf("In main_handler\n");
}

図 8-14. T1520XH2 — 直接モニター・ハンドラーを提供するサービス・プログラムの例: 次の図に、サービス・プログラム HANDLERS のソースを示します。

関数でなくラベルをハンドラーとして使用する例: 以下の例では、関数でなくラベルをハンドラーとして使用する直接モニター・ハンドラーについて説明します。
### ILE 条件ハンドラーの使用

ILE 条件ハンドラーにより、1つ以上の条件ハンドラーを実行時に登録できます。ILE 条件ハンドラーを登録するには、ILE 条件ハンドラー登録 (CEEHDLR) パイド可能 API を使用します。この API を使用する際には、ソース・コードに <lecond.h> ヘッダー・ファイルを組み込みます。ILE 条件ハンドラーは、ILE 条件ハンドラー登録抹消 (CEEHDLU) パイド可能 API を呼び出すことで、登録抹消できます。

条件ハンドラーは、ILE 条件ハンドラー登録 (CEEHDLR) パイド可能 API によって実行時に登録される例外ハンドラーです。条件処理プログラムは、例外を処理するか、バーコレートするか、あるいはプロモートするために使用されます。例外は、ILE 条件の形で条件処理プログラムに提示されます。
ILE C 条件ハンドラーを使用するタイミング： 複数の ILE 言語にわたる条件処理の（または例外処理の有効範囲を呼び出しスタック項目にするための）一貫したメカニズムを必要とする場合は、ILE バインド可能 API CEEHDLRを使用してください。活動化グループを有効範囲とするシグナル・ハンドラーとは異なり、CEEHDLR はそれを呼び出す関数を有効範囲とします。

ILE 条件ハンドラーは、ILE 条件を使用してシステム間の整合性を高めます。ILE 条件は、HLL におけるエラー条件に関する、システムから独立した表現です。

条件ハンドラーを使用する ILE ソースの例： 次の例は、プログラム MYPGM のソースを示しています。

```c
#include <leond.h>
#include <stdio.h>
void my_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new);
void main_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new);
void fred(void)
{
    _HDLR_ENTRY hdlr=my_handler;
    Char *p = NULL;
    CEEHDLR(&hdlr, NULL,NULL);
    *p = 'x'; /* exception */
    CEEHDLU(&hdlr,NULL);
}
int main(void)
{
    _HDLR_ENTRY hdlr=main_handler;
    CEEHDLR(&hdlr,NULL,NULL);
    fred();
}
```

図 8-16. T1520XH5 — ILE 条件ハンドラーを使用するための ILE C ソース — main()

例 MYPGM では、次のようになります。
- プロシージャー main() により、条件 handler main_handler が登録されます。
- プロシージャー main() が関数 fred() を呼び出し、この関数により条件ハンドラー my_handler が登録されます。
- 関数 fred() で例外が発生し、その結果 my_handler が制御を行い、そして次に main_handler が制御を行います。
main() 関数は制御境界です。この例外は未処理とみなされるため、関数 fred() に機能チェックが送信されます。ハンドラー my_handler および main_handlerが再度、今度は機能チェックのために呼び出されます。これらのハンドラーはいずれも機能チェックを処理しないため、プログラムは異常終了し、main() 関数の呼び出し元に対して CEE9901 メッセージが送信されます。

ILE 条件ハンドラーを提供するサービス・プログラムの例： 以下の例は、サービス・プログラムHANDLERS のソースを示しています。
例外処理の例: 次の例は、条件ハンドラーを使用して例外を処理する方法を示しています。この例では次のようにになります。

- ILE 条件ハンドラー cond_hdlr が、CEEHDLR を使用して main() 関数に登録されます。
- 次に、MCH1211（ゼロ除算）例外が発生します。ハンドラー cond_hdlr が呼び出され、例外を処理する必要があることを示します。
- この後、main() 関数で制御が再開されます。

以下のステップにより、図 8-17 のソースが作成され、実行されます。

1. プログラム T1520IC6 を作成するには、次のように入力します。
   ```
   CRTBNDC PGM(MYLIB/T1520IC6) SRCFILE(QCPPLE/QACSRC)
   ```
2. プログラム T1520IC6 を実行するには、次のように入力します。
   ```
   CALL PGM(MYLIB/T1520IC6)
   ```

出力は次のとおりです。

```
condition was raised: Facility_ID = MCH, MsgNo = 0x1211
The condition was handled.
```

実行キーを押して端末セッションを終了してください。
/* This program uses the ILE bindable API CEEHDLR to handle a condition. */
#include <stdio.h>
#include <stdlib.h>
#include <leawi.h>
/* A condition handler registered by a call to CEEHDLR in main(). */
void cond_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    *rc = CEE_HDLR_RESUME; /* handle the condition */
    printf("condition was raised: Facility_ID = %.3s, MsgNo = 0x%4.4x\n", 
             cond->Facility_ID, cond->MsgNo);
}
int main(void)
{
    _HDLR_ENTRY hdlr = cond_hdlr;
    _FEEDBACK fc;
    int x,y;
    int zero = 0;
    /* Register the condition handler. */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    { /* Failed to register the condition handler
        exit(88);
    }
    /* Cause a divide by zero condition. */
    x = y / zero;
    /* The code resumes here after the condition has been handled. */
    printf("The condition was handled.\n");
}

/* This program uses the ILE bindable API CEEHDLR to handle a condition. */
#include <stdio.h>
#include <stdlib.h>
#include <leawi.h>
/* A condition handler registered by a call to CEEHDLR in main(). */
void cond_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    *rc = CEE_HDLR_RESUME; /* handle the condition */
    printf("condition was raised: Facility_ID = %.3s, MsgNo = 0x%4.4x\n", 
             cond->Facility_ID, cond->MsgNo);
}
int main(void)
{
    _HDLR_ENTRY hdlr = cond_hdlr;
    _FEEDBACK fc;
    int x,y;
    int zero = 0;
    /* Register the condition handler. */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    { /* Failed to register the condition handler
        exit(88);
    }
    /* Cause a divide by zero condition. */
    x = y / zero;
    /* The code resumes here after the condition has been handled. */
    printf("The condition was handled.\n");
}

/* This program uses the ILE bindable API CEEHDLR to handle a condition. */
#include <stdio.h>
#include <stdlib.h>
#include <leawi.h>
/* A condition handler registered by a call to CEEHDLR in main(). */
void cond_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    *rc = CEE_HDLR_RESUME; /* handle the condition */
    printf("condition was raised: Facility_ID = %.3s, MsgNo = 0x%4.4x\n", 
             cond->Facility_ID, cond->MsgNo);
}
int main(void)
{
    _HDLR_ENTRY hdlr = cond_hdlr;
    _FEEDBACK fc;
    int x,y;
    int zero = 0;
    /* Register the condition handler. */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    { /* Failed to register the condition handler
        exit(88);
    }
    /* Cause a divide by zero condition. */
    x = y / zero;
    /* The code resumes here after the condition has been handled. */
    printf("The condition was handled.\n");
}

/* This program uses the ILE bindable API CEEHDLR to handle a condition. */
#include <stdio.h>
#include <stdlib.h>
#include <leawi.h>
/* A condition handler registered by a call to CEEHDLR in main(). */
void cond_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    *rc = CEE_HDLR_RESUME; /* handle the condition */
    printf("condition was raised: Facility_ID = %.3s, MsgNo = 0x%4.4x\n", 
             cond->Facility_ID, cond->MsgNo);
}
int main(void)
{
    _HDLR_ENTRY hdlr = cond_hdlr;
    _FEEDBACK fc;
    int x,y;
    int zero = 0;
    /* Register the condition handler. */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    { /* Failed to register the condition handler
        exit(88);
    }
    /* Cause a divide by zero condition. */
    x = y / zero;
    /* The code resumes here after the condition has been handled. */
    printf("The condition was handled.\n");
}

/* This program uses the ILE bindable API CEEHDLR to handle a condition. */
#include <stdio.h>
#include <stdlib.h>
#include <leawi.h>
/* A condition handler registered by a call to CEEHDLR in main(). */
void cond_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    *rc = CEE_HDLR_RESUME; /* handle the condition */
    printf("condition was raised: Facility_ID = %.3s, MsgNo = 0x%4.4x\n", 
             cond->Facility_ID, cond->MsgNo);
}
int main(void)
{
    _HDLR_ENTRY hdlr = cond_hdlr;
    _FEEDBACK fc;
    int x,y;
    int zero = 0;
    /* Register the condition handler. */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    { /* Failed to register the condition handler
        exit(88);
    }
    /* Cause a divide by zero condition. */
    x = y / zero;
    /* The code resumes here after the condition has been handled. */
    printf("The condition was handled.\n");
}

/* This program uses the ILE bindable API CEEHDLR to handle a condition. */
#include <stdio.h>
#include <stdlib.h>
#include <leawi.h>
/* A condition handler registered by a call to CEEHDLR in main(). */
void cond_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    *rc = CEE_HDLR_RESUME; /* handle the condition */
    printf("condition was raised: Facility_ID = %.3s, MsgNo = 0x%4.4x\n", 
             cond->Facility_ID, cond->MsgNo);
}
int main(void)
{
    _HDLR_ENTRY hdlr = cond_hdlr;
    _FEEDBACK fc;
    int x,y;
    int zero = 0;
    /* Register the condition handler. */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    { /* Failed to register the condition handler
        exit(88);
    }
    /* Cause a divide by zero condition. */
    x = y / zero;
    /* The code resumes here after the condition has been handled. */
    printf("The condition was handled.\n");
}
in fred_hdlr, percolate exception.
in main_hdlr: Facility_ID = MCH, MsgNo = 0x1211
Resume here because resume cursor not moved and main_hdlr handled the exception.
A condition was percolated from fred() to main() and was then handled.
実行キーを押して端末セッションを終了してください。

T1520IC7 — 条件処理のためにメッセージをバーコレートするための ILE C ソース
/* This program uses the ILE bindable API CEEHDLR to enable handlers */
/* that percolate and handle a condition. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <leawi.h>
/* A condition handler registered by a call to CEEHDLR in fred(). */
void fred_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    if (!memcmp(cond->Facility_ID, "MCH", 3) && cond->MsgNo == 0x1211)
    {
        *rc = CEE_HDLR_PERC; /* ... let it percolate to main ... */
        printf("in fred_hdlr, percolate exception.\n");
    }
    else
    {
        *rc = CEE_HDLR_RESUME; /* ... otherwise handle it. */
        printf("in fred_hdlr, handle exception.\n");
    }
}
/* A condition handler registered by a call to CEEHDLR in main(). */
void main_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    printf("in main_hdlr: Facility_ID = %.3s, MsgNo = 0x%4.4x\n",
            cond->Facility_ID, cond->MsgNo);
    **(_INT4 **)token = 1; /* Update the user's token. */
    *rc = CEE_HDLR_RESUME; /* Handle the condition */
}
int fred(void)
{
    _HDLR_ENTRY hdlr = fred_hdlr;
    _FEEDBACK fc;
    int x,y, zero = 0;
    /* Register the handler without a token. */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(88);
    }
    /* Cause a divide by zero condition. */
    x = y / zero;
    printf("Resume here because resume cursor not moved and main_hdlr";
           " handled the exception\n");
}
int main(void)
{
    _HDLR_ENTRY hdlr = main_hdlr;
    _FEEDBACK fc;
    volatile _INT4 token=0, *tokenp = &token;
    /* Register the handler with a token of type _INT4. */
    CEEHDLR(&hdlr, (_POINTER *)&tokenp, &fc);
    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(99);
    }
例外のプロモート例: 次の例では、

- `main()` で、バインド可能な API `CEEHDLR` が `main_hdlr` を登録します。
- `fred()` で、バインド可能な API `CEEHDLR` が `fred_hdlr` を登録します。
- MCH1211（ゼロ除算）例外が発生し、ハンドラー `fred_hdlr` が呼び出されます。
- ハンドラー `fred_hdlr` は、バインド可能な API `CEEMRCR` を使用して、再開カーソルを `main()` 関数内の再開ポイントに移動します。
- ハンドラー `fred_hdlr` により CEE9902 の条件トークンが作成され、結果コードはプロモートを行うように設定されます。
- ハンドラー `fred_hdlr` が戻り、元の MCH1211 は CEE9902 にプロモートされます。
- ハンドラー `main_hdlr` が CEE9902 例外により呼び出され、結果コードは条件を処理するよう設定されます。
- ハンドラー `main_hdlr` が戻り、CEE9902 が処理されます。
- `main()` での `fred()` の呼び出しに続き、ステートメントで制御が再開します。

プログラム T1520IC8 のソース・コードは、『T1520IC8 — メッセージをプロモートして条件を処理するための ILE C ソース』に示されています。

1. プログラム T1520IC8 を作成するには、次のように入力します。
   
   ```
   CRTBNDC PGM(MYLIB/T1520IC8) SRCFILE(QCPPLE/QACSRC)
   ```

2. プログラム T1520IC8 を実行するには、次のように入力します。
   
   ```
   CALL PGM(MYLIB/T1520IC8)
   ```

出力は以下の状況情報メッセージになります。

```
in fred_hdlr: moving resumes. Facility_ID = MCH, MsgNo = 0x1211
promoting condition....
A condition was promoted from MCH1211 to CEE9902 by fred() and was handled by the condition handler enabled in main().
```

実行キーを押して端末セッションを終了してください。

---

T1520IC8 — メッセージをプロモートして条件を処理するための ILE C ソース

/* This program uses the ILE bindable API CEEHDLR to promote a */
/* divide by zero condition to a CEE9902. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <leawi.h>
/* A condition handler registered by a call to CEEHDLR in fred(). */
void fred_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{  
  _INT4 type=1;
  CEEMRCR(&type,NULL);
  /* Move the resume cursor to the resume point */
  /* in main(). */
  /* If its a divide by zero error */
  printf("in fred_hdlr: moving resumes. ");
printf("Facility_ID = %.3s, MsgNo = 0x%4.4x\n",
    cond->Facility_ID, cond->MsgNo);
if (!memcmp(cond->Facility_ID, "MCH", 3) && cond->MsgNo == 0x1211)
{
    *rc = CEE_HDLR_PROM; /*... Promote the condition to unexpected error.*/
    *new = *cond;
    memcpy(new->Facility_ID, "CEE", 3);
    new->MsgNo = 9902;
    printf("promoting condition....\n");
} else
{
    *rc = CEE_HDLR_PERC; /*...Otherwise,Percolate to the next handler. */
    printf("percolating condition....\n");
}
/* A condition handler registered by a call to CEEHDLR in main(). */
void main_hdlr(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new)
{
    if (!memcmp(cond->Facility_ID,"CEE",3) && cond->MsgNo == 9902;)
        **(_INT4 **)token = 1; /* Got the promoted CEE9902. */
    else
        **(_INT4 **)token = 2; /* It is not a CEE9902. */
    *rc = CEE_HDLR_RESUME; /* Handle the condition. */
}
int fred(void)
{
    _HDLR_ENTRY hdlr = fred_hdlr;
    _FEEDBACK fc;
    int x,y, zero = 0;
    /* Register the handler without a token. */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(88);
    }
    /* Cause a divide by zero condition. */
    x = y / zero;
    /* This is not the resume point because of the call to CEEMRCR in */
    /* fred_hdlr. */
    {
        printf("This is not the resume point: should not get here\n");
    }
}
int main(void)
{
    _HDLR_ENTRY hdlr = main_hdlr;
    _FEEDBACK fc;
    volatile _INT4 token=0, *tokenp = &token;
    /* Register the handler with a token of type _INT4. */
    CEEHDLR(&hdlr, (_POINTER *)&tokenp, &fc);
    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(99);
    }
    fred();
    /* See if the condition handler for main() received the promoted */
    /* condition. */
    if (*tokenp == 1)
        printf("A condition was promoted from MCH1211 to CEE9902 by "
            "fred() and was handled by the condition handler enabled "
            "in main().\n");
}
C/C++ シグナル・ハンドラーの使用

HLL 周有ハンドラーは、エラー処理のために定義された言語機能です。ILE C/C++ signal() 関数を使用して、例外メッセージを処理できます。

IBM i 例外、ILE C/C++ 実行時環境によって C および C++ シグナルにマップされます。シグナルハンドラーは、シグナルの処置方針を決定します。呼び出し元の活動化グループとは異なる活動化グループでシグナルハンドラーを登録することはできません。シグナルハンドラーが、処理されているシグナルの発生元とは異なる活動化グループ内にある場合、動作は未定義になります。

シグナル・ハンドラーの使用タイミング

複数のプラットフォーム間での移植可能コードでは、例外の処理に signal() 関数のみを使用します。ILE 条件ハンドラーを使用する場合は、ILE 対応言語間で例外を処理するための、一貫性のあるメカニズムが必要な場合のみにしてください。ILE 対応プラットフォーム間での移植性が懸念事項である場合、ILE 条件ハンドラーと signal() 関数を使用できます。それ以外の場合、3つすべてのタイプのハンドラーを使用できます。""C/C++ シグナルと ILE 例外ハンドラーの同時使用"" を参照してください。

注：

ストリーム入出力関数は、SIGIO シグナルをトラップします。このシグナルは、任意のタイプのソケットで通常のデータ、OOB データ、エラー状態など、どのようなことも発生すれば送信されます。SIGIO に登録されたシグナルハンドラーは、ストリーム・ファイルの処理時に生成された例外に対しては呼び出されません。

signal() 関数を使用すると常に、例外が暗黙的に処理されます。ただし、シグナル処置が SIG_DFL の場合は除き、その場合は、例外がパーコレートされます。直接モニター・ハンドラーを使用した場合、例外を暗黙的に処理する制御処置 (_CTLA_HANDLE、CTLA_HANDLE_NO_MSG、_CTLA_IGNORE、または _CTLA_IGNORE_NO_MSG) を指定するか、(制御処置 _CTLA_INVOKE を指定した場合) QMHCHGEM または ILE 状態処理 API を使用してハンドラー関数内で例外を明示的に処理する必要があります。

注：通常、直接モニターが最速のハンドラーです。

HLL 周有のハンドラー (ILE C ではシグナル・ハンドラー) は、グローバルでです。これは、signal() 関数が呼び出された活動化グループ内のすべての関数呼び出しに対して有効になります。ILE では、条件ハンドラーおよび直接モニター・ハンドラーは、それを有効化した関数に対して、またはその関数で無効にされるまで有効とされます。

以下の例では、シグナル関数が戻ったときにシグナル・ハンドラーの状態を変更しない場合、シグナル・ハンドラーの状態を明示的に管理する必要があることを示します。

```
#include <signal.h>
void f(void)
{
    void (*old_state)(int);
    /* Save old state of signal action */
    old_state = signal(SIGALL,handlr);
    /* Other code in your application */
    /* Reset state of signal */
    signal(SIGALL,old_state);
}
```

図 8-19. シグナル・ハンドラーの状態を管理するための ILE C ソース
ILE 条件ハンドラーおよび直接モニター・ハンドラーはグローバル・ハンドラーではないため、この要件はありません。

シグナルの発信
シグナルは、暗黙的または明示的に発信されます。明示的にシグナルを発信する場合は、raise() 関数を使用します。例外が発生すると、IBM i によって暗黙的にシグナルが発信されます。例えば、存在しないプログラムを呼び出した場合、該当するプログラム・オブジェクトが見つからないことを示す暗黙的シグナルが発信されます。

シグナル処理の関数プロタイプ
ヘッダー・ファイル <signal.h> には、シグナル処理に関する関数プロタイプがいくつか含まれています。

ご使用のプログラムでのシグナル処理に、以下の関数を使用することができます。
- raise()
- signal()
- _GetExcData()

_getExcData() 関数は ILE C/C++ 拡張機能で、これにより、シグナルに関連付けられた例外メッセージに関する情報を入手することができます。また、この関数はその例外メッセージに関する情報を含む構造体を返します。_getExcData() 関数は、#pragma exception_handler ディレクティブに渡されたものと同じ構造体を返します。

signal() 関数は、シグナルが発送された際に実行されるアクションを指定します。<signal.h> ヘッダー・ファイルには、マクロとして定義されたシグナルが 10 個あります。また、マクロ SIGALL にはシグナルのセマンティクスがありますが、これには独自の特性があります。10 個のシグナルを、以下に示します。

SIGABRT
プログラムの異常終了。

SIGFPE
ゼロ除算などの算術演算エラー。

SIGILL
許可されていない命令。

SIGINT
対話式アプリケーション・シグナルの受信などのシステム中断。

SIGIO
レコード・ファイル・エラー状態。

SIGOTHER
他のどのシグナルにもマップされない、その他のすべての *ESCAPE および *STATUS メッセージ。

SIGSEGV
記憶域へのアクセスが無効です。

SIGTERM
プログラムに終了要求が送信されました。

SIGUSR1
ユーザー定義のシグナル・ハンドラー用に予約済みです。
SIGUSR2
ユーザー定義のシグナル・ハンドラー用に予約済みです。

SIG_IGN と SIG_DFL も、<signal.h> ヘッダー・ファイルに含まれるシグナル・アクションです。

SIG_IGN
シグナルの無視。

SIG_DFL
シグナルに対するデフォルト・アクション

SIGALL は ILE C/C++ 拡張機能で、ユーザーはこれを使用して、アクションが SIG_DFL であるすべてのシグナルに対する独自のデフォルト処理関数を登録することができます。例のセクションに示したとおり、このデフォルト処理関数は、signal() 関数を使用して SIGALL で登録することができます。機能チェックはシグナルではないため、シグナル関数によるモニターの対象にはできません。raise() 関数によって SIGALL をシグナル通知することはできません。

ILE C/C++ ランタイム環境によるシグナルの処理方法
シグナルを受信すると、Integrated Language Environment C/C++ ランタイム環境は、以下の 3 つのうちいずれかの方法でそのシグナルを処理します。

- 関数の値が SIG_IGN である場合は、例外はランタイム環境によって処理され、シグナル・ハンドラーは呼び出されないため、シグナルは無視されます。シグナルにマップされたメッセージが *ESCAPE または *NOTIFY メッセージである場合、そのメッセージはジョブ・ログに入れられます。
- 関数の値が関数へのポインターである場合、そのポインターによってアドレス指定された関数が呼び出されます。
- 関数の値が SIG_DFL である場合、システムは (ここに記載した 3 つの方法のうちいずれかを選択して) SIGALL に対して登録された値を使用します。SIGALL に対する関数の値が SIG_DFL である場合は、例外はパーソレートされます。

注: 関数の値は、signal() 関数の呼び出しにおける関数引数です。

シグナル・アクションのリセット
例外が発生するかシグナルが発信されると、シグナル処理関数が呼び出されます。signal() 関数を使用してハンドラーが定義されます。sig パラメーターに割り当てる値は、funct パラメーターで参照される関数と関連付けられます。

C プログラムでシグナル・ハンドラーが呼び出されると、それに対応するシグナル・アクションが SIG_DFL に設定されます。同じシグナルを再度処理する場合は、このシグナル・アクションをリセットする必要があります。リセットしない場合、それ以降の例外に対してはデフォルトのアクションが実行されます。シグナルの処理は、signal() を呼び出すことでハンドラーの中からも外からもリセットすることができます。

例:
次の図は、シグナル・ハンドラーをリセットするソース・コードを示しています。
signal ( SIGINT, &myhandler );
raise ( SIGINT );   /* signal is handled by myhandler.  */
...
raise ( SIGINT );   /* signal is handled by SIG_DFL. */
...
signal ( SIGINT, &myhandler ); /* reset signal handler to myhandler. */
raise ( SIGINT ); /* signal is handled by myhandler. */

図8-20 シグナル・ハンドラーのリセット

図8-20では、次のようにになります。
1. 省略時値は SIG_IGN 、別のハンドラー、または同じハンドラーにリセットできます。シグナル・ハンドラーは、再帰的に呼び出すことが可能です。複数のシグナル・ハンドラー呼び出しは、いったんスタックされると、他の呼び出しであるかのように動作します。例えば、前の呼び出し元に対するアクション・シグナルが選択された場合、(その呼び出し前のシグナル・ハンドラーであっても) 前の呼び出し元には返されず、前の呼び出し元へ戻ります。
2. signal() 関数は、指定されたシグナルに対する前のシグナル・ハンドラーのアドレスを返し、新しいシグナル・ハンドラーのアドレスを設定します。

シグナル・ハンドラーのスタック
以下の図に示すとおり、ユーザーは、signal() によって返される値を使用して自分でシグナル・ハンドラーをスタックすることができます。

void (*)(*func1)();
void (*)(*func2)();
func1 = signal ( SIGINT, &handler2 ); /*func1 contains the address of */
    /*func1 contains the address of */
    /*a previous signal handler or */
    /*SIG_DFL if no handler has been */
    /*defined. */
func2 = signal ( SIGINT, func1); /*func2 contains the address of */
    /*handler2. */

図8-21 シグナル・ハンドラーのスタック

例: シグナル・ハンドラーのセットアップ
次の例は、シグナル・ハンドラーのセットアップ方法を示しています。この例では、シグナル・ハンドラーがセットアップされていなければ、SIGIO に対するデフォルトのアクションは SIG_IGN であることを示しています。例外は無視されます。シグナル・ハンドラーが SIGIO に対してセットアップされている場合、そのシグナル・ハンドラーが呼び出されます。

手順:
1. プログラム T1520SIG を作成するには、次のように入力します。
   CRTBNDC PGM(MYLIB/T1520SIG) SRCFILE(QCPPLE/QACSRC)
   8-28 ページの『シグナル・ハンドラーをセットアップするソース・コードのサンプル』はそのソース・コードを示したものです。
2. プログラム T1520SIG を実行するには、次のように入力します。
   CALL PGM(MYLIB/T1520SIG)
   出力は一連の画面に表示されます。
The first read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
The first read finishes
The second read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In SIGIO handler
Exception message ID is CPF5001
Signal raised is 9
The second read finishes
The third read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In SIGALL handler
Exception message ID is CPF5001
Signal raised is 9
The third read finishes

#include <stdio.h>
#include <signal.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#define FILE_NAME "QTEMP/MY_FILE"
#define RCD_LEN 80
#define NUM_RCD 5
/* The signal handler for SIGIO. */
static void handler_SIGIO(int sig)
{
    printf("In SIGIO handler\n");
    printf("Exception message ID is %7.7s\n", _EXCP_MSGID);
    printf("Signal raised is %d\n", sig);
}
/* The signal handler for SIGALL. */
static void handler_SIGALL(int sig)
{
    _INTRPT_Hndlr_Parms_T data;
    _GetExcData(&data);
    printf("In SIGALL handler\n");
    printf("Exception message ID is %7.7s\n", data.Msg_Id);
    printf("Signal raised is %d\n", sig);
}
int main(void)
{
    _RFILE *fp;
    Int  i;
    char  buf[RCD_LEN];
    char  cmd[100];
    /* Create a file. */
sprintf(cmd, "CRTPF FILE(%s) RCDLEN(%d)", FILE_NAME, RCD_LEN);
system(cmd);

/* Open the file for write. */
if ( (fp = _Ropen(FILE_NAME, "wr")) == NULL )
{
    printf("Open for write fails\n");
    exit(1);
}

/* Write some data into the file. */
memset(buf, '1', RCD_LEN);
for ( i = 0; i < NUM_RCD; i++ )
{
    _Rwrite(fp, buf, RCD_LEN);
}
_Rclose(fp);

/* Open the file for the first read. */
if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{
    printf("Open for the first read fails\n");
    exit(2);
}

/* Open the file for the first read. */
if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{
    printf("Open for the first read fails\n");
    exit(2);
}

/* Read until end-of-file. */
/* Since there is no signal handler set up and the default */
/* action for SIGIO is SIG_IGN, the EOF exception is ignored. */
i = 1;
printf("The first read starts\n");
while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
{
    printf("Read record %d\n", i++);
}
_Rclose(fp);
printf("The first read finishes\n");
/* Set up a signal handler for SIGIO. */
signal(SIGIO, handler_SIGIO);
/* Open the file for the second read. */
if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{
    printf("Open for the second read fails\n");
    exit(3);
}

/* Set up a signal handler for SIGIO. */
signal(SIGIO, handler_SIGIO);
/* Open the file for the second read. */
if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{
    printf("Open for the second read fails\n");
    exit(3);
}

/* Read until end of file. */
/* Since a signal handler is set up for SIGIO, the signal */
/* handler is called when the EOF exception is generated. */
i = 1;
printf("The second read starts\n");
while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
{
    printf("Read record %d\n", i++);
}
_Rclose(fp);
printf("The second read finishes\n");
/* Set up a signal handler for SIGIO. */
signal(SIGIO, handler_SIGIO);
/* Open the file for the third read. */
if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{
    printf("Open for the third read fails\n");
    exit(4);
}

/* Read until end of file. */
/* Since there is no signal handler for SIGIO but there is a */
/* signal handler for SIGALL, the signal handler for SIGALL */
/* is called when the EOF exception is generated. But */
/* the signal ID passed to the SIGALL signal handler is still */
/* equal to SIGIO. */
i = 1;
printf("The third read starts\n");
C/C++ シグナルと ILE 例外ハンドラーの同時使用

言語固有のエラー処理 (C シグナル) と追加の ILE 例外ハンドラー・タイプの両方を使用する場合、例外ハンドラーの優先順位が重要になります。

優先順位

例外が発生した呼び出しスタック項目では、システムは以下の優先順位でハンドラーを呼び出します。
1. 直接モニター
2. ILE 条件ハンドラー
3. signal()
直接モニター・ハンドラーとシグナル・ハンドラーの併用の例

以下の例では、#pragma exception_handler ディレクリプと signal() 関数を併用する方法を示しています。また、この例では、SIGIO を使用した例外処理の方法も示しています。EOF メッセージは SIGIO にマップされます。SIGIO の省略時値は SIG_IGN です。また、HLL 固有のハンドラーと直接モニター・ハンドラーの両方が定義済みである場合は、直接モニター・ハンドラーが先に呼び出されることも示しています。

この例では、8-13 ページの『直接モニター・ハンドラーの使用方法を説明するソースの例』に示したソースを使用しています。

1. プログラム T1520ICA を作成するには、次のように入力します。

   CRTBNDC PGM(MYLIB/T1520ICA) SRCFILE(QCPPEL/QACSRC)

2. プログラム T1520ICA を実行するには、次のように入力します。

   CALL PGM(MYLIB/T1520ICA)

出力の最初の画面は、次のようにになります。

```
The first read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
The first read finishes
The second read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In direct monitor handler
Exception message ID is CPF5001
The second read finishes
The third read starts
Read record 1
```

出力の 2 番目の画面は、次のようにになります。

```
Read record 2
Read record 3
Read record 4
Read record 5
In direct monitor handler
Exception message ID is CPF5001
In signal handler
Exception message ID is CPF5001
The third read finishes
The fourth read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In signal handler
Exception message ID is CPF5001
The fourth read finishes
実行キーを押して端末セッションを終了してください。
```
ネストされた例外の処理
例外はネストされることがあります。ネストされた例外とは、別の例外の処理中に起こる例外のことで、これが起こった時には、最初の例外の処理が一時的に中断されます。例外処理は、最後に生成された例外から再開されます。

注: ネストされた例外によってプログラムが終了した場合、最初の例外の例外ハンドラーが完了しないことがあります。

例:
以下の例では、ネストされた例外を示します。

```c
#include <signal.h>
void hdlr_hdlr(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    /* Handle exception 2 using QMCHGEM. */
}
void main_hdlr(_INTRPT_Hndlr_Parms_T * _ptr128 parms)
{
    #pragma exception_handler(hdlr_hdlr,0,0,_C2_MH_ESCAPE)
    /* Generate exception 2. */
    /* Handle exception 1 using QMCHGEM. */
}
int main(void)
{
    #pragma exception_handler(main_hdlr,0,0,_C2_MH_ESCAPE)
    /* Generate exception 1. */
}
```

図8-23 例外をネストするための ILE C ソース
この例では、main()関数は、main_hdlr が制御を取るようにする例外を生成します。ハンドラー main_hdlr は別の例外を生成し、これによって hdlr_hdlr が制御を取るようにします。ハンドラー hdlr_hdlr が、その例外を処理します。制御は main_hdlr で再開し、元の例外が処理されます。

この例で示しているように、例外ハンドラー内で例外を受け取ることができます。例外の再帰を防止するために、例外ハンドラー呼び出しスタック項目は、例外ハンドラーブロックに値が乗って関数を終了するように機能します。そのため、例外ハンドラー内で例外をモニターすることをお勧めします。

取り消し処理プログラムの使用

取扱ハンドラーは、C++でスタックのアンワイド時にデストラクターを呼び出すために使用されます。オブジェクトが正しく破棄されるようにするために、C++ try-catch-throw フィーチャーを使用することをお勧めします。

注: try-catch-throw フィーチャーの使用については、「ILE C/C++ 解説書」を参照してください。

取り消しハンドラーは、関数内のコード本体のまわりで有効にすることができます。取り消しハンドラーは、有効になっていると、呼び出しスタック項目の一時停止ポイントがコード内 (#pragma cancel_handler および #pragma disable_handler ディレクティブ内) にあり、呼び出しスタック項目が取り消された場合のみ制御を取ります。

#pragma cancel_handler ディレクティブは、呼び出しスタック項目 (呼び出しスタック項目内の一時停止ポイント) 内で取り消しハンドラーを静的に登録する手段を提供します。呼び出しスタック項目終了ユーザ
一出口プロシージャー登録 (CEERTX) および呼び出しスタック項目終了ユーザー出口プロシージャー登録
抹消 (CEETUTX) ILE バインド可能 API は、ユーザー定義ルーチンを自動的に登録する方法を提供します。
このユーザー定義ルーチンは、このルーチンの登録対象である呼び出しスタック項目が取り消されたとき
に実行されるものです。

取り消しハンドラーは、呼び出しスタック項目が通常の戻る以外の何らかの理由によって終了されたとき
に、クリーンアップ処置および回復処置のための制御を受け取るようにするという重要な機能を提供しま
す。

例:

以下の単純な例で、ILE 取り消しハンドラーメカニズムの使用を示します。この機能により、アプリケ
ーションは、特定の関数呼び出しを取り消したときに、エラーのレポートおよびログイン、ファイルのク
ローズなどの操作を実行するために呼び出されるユーザー提供関数を使用できます。取り消しを発生させる
通常の方法は、exit() 関数または abort() 関数の使用、以前の呼び出しにジャンプするための longjmp()
関数の使用、および未処理例外から生成された CEE9901 機能チェックの使用です。

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <except.h>

/* The following function is called a "cancel handler". It is 
* registered for a particular invocation (function) with the 
* #pragma cancel_handler directive. The variable identified 
* on this directive as the "communications area" can be accessed 
* using the 'Com_Area' member of the _CNL_HndlrParms_T structure. */

void CancelHandlerForReport( _CNL_HndlrParms_T *cancel_info ) {
    printf("In Cancel Handler for function 'Report' ...
    /* Changing the value in the communications area will update the */
    /* 'return_code' variable in the invocation being cancelled */
    /* (in function 'Report' in this example). Note that the */
    /* ILE C compiler will issue a warning for the following */
    /* statement since it uses a non-ISO C compliant technique. */
    /* However, this will not affect the expected runtime behavior. */
    /* Set "return_code" in Report to an arbitrary number. */
    *(volatile unsigned *)cancel_info->Com_Area ) = 500;
    printf("Communication Area now has the value: %d \n",
        *(volatile unsigned *)cancel_info->Com_Area );
    printf("Leaving Cancel Handler for function 'Report'...
    \
    */
}

/* The following function is also called a cancel handler but has */
/* been registered for the 'main' function. That is, when the */
/* 'main' function is cancelled, this function will automatically */
/* be called by the system. */

void CancelHandlerForMain( _CNL_HndlrParms_T *cancel_info ) {
    printf("In Cancel Handler for function 'main' ...
    /* Changing the value in the communications area will update the */

/* 'return_code' variable in the invocation being cancelled */
/* (in function 'main' in this example). Note that the */
/* ILE C compiler will issue a warning for the following */
/* statement since it uses a non-ISO C compliant technique. */
/* However, this will not affect the expected runtime behavior. */
/* Set "return_code" in main to an arbitrary number. */
*(volatile unsigned *)cancel_info->Com_Area = 999;
printf("Communication Area now has the value: %d \n",
   *(volatile unsigned *)cancel_info->Com_Area);
printf("Leaving Cancel Handler for function 'main'... \n");
}

/*-------------------------------------------------------------------*/
/* The following is simple function that registers another function */
/* (named 'CancelHandlerForReport' in this example) as its "cancel */
/* handler". When 'exit()' is used from this function, then this */
/* invocation and all prior invocations are cancelled by the system */
/* and any registered cancel handlers functions are automatically */
/* called. */
/* *-------------------------------------------------------------------*/
void Report( void ) {
  volatile unsigned return_code; /* communications area */
  #pragma cancel_handler( CancelHandlerForReport, return_code )
  printf("in function Report()...about to call 'exit'...\n");
  /* Using the exit function will cause this function invocation */
  /* and all function invocations within this program to be */
  /* cancelled. If any of the functions being cancelled have */
  /* cancel handlers enabled, then those cancel handler functions */
  /* will be called by the system after each cancellation. */
  exit( 99 ); /* exit with an arbitrary value */
  printf("in function Report() just after calling 'exit'... \n");
  #pragma disable_handler
}

/*-------------------------------------------------------------------*/
/* In the 'main()' function a cancel handler is registered so that */
/* the function 'CancelHandlerForMain()' is called if 'main()' is */
/* cancelled. */
/* *-------------------------------------------------------------------*/
int main( void ) {
  volatile unsigned return_code; /* communications area */
  #pragma cancel_handler( CancelHandlerForMain, return_code )
  return_code = 0; /* initialize return code which will */
  /* eventually be set in the cancel handler */
  printf("In main() about to call Report()...\n");
  Report();
  printf("...back from calling Report(). \n");
  printf("return_code = %d \n", return_code);
  #pragma disable_handler
}

/*-------------------------------------------------------------------*/
/* This program will result in the following screen output: */
/* */
/* In main() about to call Report()... */
/* in function Report()...about to call 'exit'... */
/* In Cancel Handler for function 'Report'... */
/* Communication Area now has the value: 500 */
/* Leaving Cancel Handler for function 'Report'... */
/* In Cancel Handler for function 'main'... */
/* Communication Area now has the value: 999 */
/* Leaving Cancel Handler for function 'main'... */
/* *-------------------------------------------------------------------*/
/* */

Example: Error checking and exception handling various methods of use

The example below demonstrates error handling using some of the methods described.

- Major/Minor tracing and code checking

IBM i: ILE C/C++ Programmer's Guide

Page 8-34
手順

1. 図8-24 に示されている DDS ソースを使用して表示装置ファイル T1520DDJ を作成するには、以下を入力します。
   CRTSPF FILE(MYLIB/T1520DDJ) SRCFILE(QCPPLE/QADDSSRC)

2. 以下に示しているソースを使用してプログラム T1520EHD を作成するには、以下を入力します。
   CRTBNDC PGM(MYLIB/T1520EHD) SRCFILE(QCPPLE/QACSRC)

   注: プログラム T1520EHD は、signal() 信号を使用して SIGIO シグナルを出します。このシグナルは、任意のタイプのソケットで通常のデータ、OOB データ、エラー状態など、どのようなことも発生すれば送信されます。

3. ライブラリー MYLIB をライブラリー・リストに追加する必要がある場合は、以下を入力します。
   ADDLIB L1B(MYLIB)

4. プログラム T1520EHD を実行するには、以下を入力します。
   CALL PGM(MYLIB/T1520EHD)

出力は次のとおりです。

<table>
<thead>
<tr>
<th>PHONE BOOK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: Smith, John</td>
</tr>
<tr>
<td>Address: 2711 Westsyde Rd.</td>
</tr>
<tr>
<td>Phone #: 721-9729</td>
</tr>
</tbody>
</table>

<ENTER> : Saves changes
f3 : Exits with changes saved
f5 : Brings back original field values

ソース・コードのサンプル

このセクションの図では、「手順」で使用されているソース・コードを示します。

A
A DSPSIZ(24 80 *DS3)
A INDARA
A R PHONE
A CF03(03 'EXIT')
A CF05(05 'REFRESH')
A 7 28 Name:
A 9 25 Address:
A 11 25 Phone #:
A 1 35 PHONE BOOK'
A 16 19 <ENTER> : Saves changes'
A 17 21 f3 : Exits with changes saved'
A 18 21 f5 : Brings back original fiel-
A d values'
A 05 21 32 Screen refreshed'
A 05 DSPATR(HI)

図8-24. T1520DDJ — 電話帳表示のための DDS ソース
/* This program illustrates how to: */
/* - check the major/minor return code */
/* - check the errno global variable */
/* - get error information from the */
/* _EXCP_MSGID global variable */
/* - use the signal function. */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stddef.h>
#include <errno.h>
#include <signal.h>
#include <recio.h>

#define IND_ON '1'
#define IND_OFF '0'
#define HELP 0
#define EXIT 2
#define REFRESH 4
#define FALSE 0
#define TRUE 1

typedef struct PHONE_LIST_T{
char name[11];
char address[20];
char phone[8];
}PHONE_LIST_T;

void error_check(void);
/* The error checking routine. */
void error_check(void)
{
if (errno == EIOERROR || errno == EIORECERR)
    printf("id:%7.7s\n", _EXCP_MSGID);
if (strncmp(_Maj_Min_rc.major_rc,"00",2) ||
    strncmp(_Maj_Min_rc.major_rc,"00",2))
    printf("Major : %2.2s	Minor : %2.2s\n",
        _Maj_Min_rc.major_rc,_Maj_Min_rc.minor_rc);
    errno = 0;
}
/* The signal handler routine. */
void sighd(int sig)
{
signal(SIGIO,&sighd);
}
/* MAIN LINE */
int main(void)
{
    FILE *dsfp;
    PHONE_LIST_T phone_in_rec,
    phone_out_rec = { "Smith, John",
                     "2711 Westsysde Rd. ",
                     "721-9729" };

    /* MAIN LINE */

7-25. T1520EHD — 例外を処理するための ILE C ソース
プログラムでのポインターの使用

ISO C では、ポインター・タイプは、関数タイプ、データ・オブジェクト・タイプ、または不完全タイプから派生されます。IBM i では、ポインター・タイプは、他の IBM i エンティティー（システム・オブジェクト（例えば、プログラム）、コード・ラベル、プロセス・オブジェクトなど）からも派生できます。これらのポインター・タイプは通常、IBM i ポインターと呼ばれ、ILE C マシン・インターフェース・ライブラリーおよび ILE C/C++ 例外処理構造体および関数で広く使用されます。

注: 「ILE C/C++ for AS/400® MI Library Reference」には、ILE C マシン・インターフェース・ライブラリー関数に関する情報が含まれています。

このトピックでは以下について説明します。

- IBM i ポインター・タイプ
- オープン・ポインターの使用
- 他のポインターの使用
- ポインター変数の宣言
- ポインターのキャスト

IBM i ポインター・タイプ

IBM i ポインター・タイプは、以下のとおりです。
オープン
他の任意のポインタータイプを保持できるポインター。

スペース
データ・オブジェクトへの汎用ポインター。

機能
*PGM オブジェクトへのシステム・ポインターまたはバインド済み ILE プロシージャーへのプロシージャー・ポインター。

システム
キー、索引、ライブラリ、*PGM オブジェクトなどのシステム・オブジェクトへのポインター。

ラベル
プロシージャーまたは関数の実行可能コード内の固定位置へのポインター。

呼び出し
ILE の下のプロシージャー（関数）呼び出し、あるいは EPM または OPM の下のプログラム呼び出しのプロセス・オブジェクトへのポインター。

中断
制御が中断されているプロシージャー内の位置へのポインター。

これらのポインター・タイプ、およびデータ・オブジェクトおよび不完全タイプへのポインターは、互いにデータ・タイプ互換ではありません。例えば、データ・オブジェクトへのポインターとして宣言された変数に、関数ポインターまたはシステム・ポインターの値を代入できません。システム・ポインターは、呼び出しポインターまたはデータ・オブジェクトへのポインターと等価であるか比較できません。上記は、オープン・ポインターには当てはまりません。

注:
- ラベル・ポインターは、setjmp マクロでのみ使用されます。
- オープン・ポインターは、疑似ポインター・タイプです。任意の他のポインター・タイプを含めることができますが、単独ではポインター・タイプではありません。

オープン・ポインターの使用
オープン・ポインターでは最適化を行えません。これらの使用は、どうしても必要な場合に限ってください。ILE C/C++ プログラムでオープン・ポインターを使用する前に、以下の特徴と制約を考慮してください。

- オープン・ポインターは void ポインターにマップされます。
- オープン(void) ポインターは、どんなタイプのポインターでも保持できます。これは、システム上のすべてのポインター・タイプとデータ・タイプの互換性があります。オープン・ポインターを他のポインター・タイプにキャストするとき、または他のポインター・タイプをオープン・ポインターにキャストするときに、コンパイル時エラーは発生しません。

注: コンテキストに不適切な値がポインターに含まれる（例えば、ポインター算術的にシステム・ポインターがある）場合、実行時例外を受け取る可能性があります。

- オープン・ポインターは、どのタイプのポインターにも割り当てることができます。

注: オープン・ポインターに保持されたポインターのタイプが、割り当てのターゲットとデータ・タイプの互換性がない場合、実行時例外を受け取る可能性があります。

- オープン・ポインターは、どのポインター・タイプに対しても等しいかどうか（==、!＝）を比較することができます。

8-38 IBM i: ILE C/C++ プログラマーの手引き
オープン・ポインター以外のポインターの使用

ILE C/C++ プログラムでポインターを使用する前に、以下の特性と制約を考慮してください。

- 等価演算 (==、!=) の場合
  - NULL ポインターは、任意のタイプのポインターに対して割り当ておよび等価比較 (==, !=) できます。
  - ポインターは、同じタイプのポインターまたはオープン・ポインターに対して割り当ておよび等価比較 (==, !=) できます。これら以外の場合は、コンパイル時エラーが発生します。
  - 条件式 if (iptr) は、式 if (ptr == NULL) に相当します。

- 関係演算 (<, >, <=, >=) の場合
  - NULL ポインターを任意のポインター・タイプと一緒に使用することはできません。
  - データ・オブジェクトを指すポインター、またはデータ・オブジェクトを指すポインターを含むオープン・ポインターのみが使用可能です。その他の場合は、コンパイル時エラーまたは実行時にエラーが発生する可能性があります。

- 算術演算 (+, -, ++, --) では、データ・オブジェクトを指すポインターのみが使用可能です。その他の場合は、コンパイル時エラーが発生します。

ポインター変数の宣言

ILE C/C++ を使用して、以下を宣言できます。

- データへのポインター
- 関数ポインター (バインドされた ILE プロシージャーへのポインター)

C および C++ で IBM i ポインター変数を宣言する

*PGM オブジェクト (プログラム) を指すポインターは、以下のいずれかの方法で宣言することができます。

  - #pragma linkage ディレクティブとの OS リンケージまたは外部 OS リンケージを含むよう指定された型定義を指すポインターを宣言する。
  - システム・ポインター (_SYSPTR) を宣言する。

ILE C <pointer.h> ヘッダー・ファイルにより提供された型定義 (typedef) を使用して、他の IBM i ポインター・タイプの変数を宣言することも可能です。

8-40 ページの図 8-27 は、IBM i C ポインターの宣言を示しています。8-40 ページの図 8-28 は、IBM i C++ ポインターの宣言を示しています。
ILE C でバインド済みプロシージャーへの関数ポインタを宣言する

関数ポインタとは、ILE プログラム・オブジェクト内のバインド済みプロシージャー（関数）、またはシステム内の OS リンケージ・プログラム・オブジェクト（システム・ポインタ）のいずれかを指すポインタです。

図 8-29 では、バインド済みプロシージャー（同じ ILE プログラム・オブジェクト内で定義された関数）を指すポインタを宣言する方法を示しています。

```c
#include <pointer.h> /* The pointer header file. */
_SYSPTR sysp; /* A system pointer. */
_SPCPTR spcp; /* A space pointer. */
_INVPTR invp; /* An invocation pointer. */
_OPENPTR opnp; /* An open pointer. */
_LBLPTR lbp; /* A label pointer. */
void (*fp) (int); /* A function pointer. */
#pragma datamodel (p128)
#pragma linkage (OS_FN_T, OS)
#pragma datamodel(pop)
typedef void (OS_FN_T) (int); /* Type of an OS-linkage function. */
OS_FN_T * os_fn_p; /* An OS-linkage function pointer. */
int * ip; /* A pointer to a data object. */
#include <pointer.h> /* The pointer header file. */
_SYSPTR sysp; /* A system pointer. */
_SPCPTR spcp; /* A space pointer. */
_INVPTR invp; /* An invocation pointer. */
_OPENPTR opnp; /* An open pointer. */
_LBLPTR lbp; /* A label pointer. */
void (*fp) (int); /* A function pointer. */
#pragma datamodel (p128)
#pragma linkage (OS_FN_T, OS)
#pragma datamodel(pop)
typedef void (OS_FN_T) (int); /* Type of an OS-linkage function. */
extern "OS" typedef void (OS_FN_T) (int);
#pragma datamodel(pop)
int * ip; /* A pointer to a data object. */
```

図 8-27. ポインター変数を宣言するための ILE C ソース

図 8-28. ポインター変数を宣言するための ILE C++ ソース

ILE C および ILE C++ で OS リンケージ付き関数ポインタを宣言する

OS リンケージ関数（プログラム）へのポインタおよびシステム・ポインタ（SYSPTR）は、データ型の互換性があります。システム・ポインタを使用してプログラムのアドレスを保持ることができ、そのプログラムをシステム・ポインタを通じて呼び出すことができます。
注: プログラムでないシステム・オブジェクトのアドレスを含むシステム・ポインターを使用して呼び出しを行うと、未定義の動作が発生します。

ILE C コンバイラーにシステム・ポインター・タイプと IBM i ポインター・タイプを関連付けるよう強制するには、以下のタスクを両方実行してください。

- システム・ポインター・タイプを void (void *) へのポインターとして定義します。
- #pragma ポインター・ディレクティブをヘッダー・ファイルに定義します。

図 8-30 は、ILE C で IBM i プログラムへのポインターを OS リンケージ付きの関数ポインターとして宣言する方法を示しています。図 8-31 は、ILE C++ で IBM i プログラムへのポインターを OS リンケージ付きの関数ポインターとして宣言する方法を示しています。

注: #pragma リンケージ OS ディレクティブがコードから除外された場合、ILE C コンバイラーは os_fct_ptr が void を返すバインド済み C 関数へのポインターであると見なし、os_fct_ptr と rslvsp() が返すシステム・ポインターとの間の非互換ポインター・タイプに対するコンパイル・エラーを出します。

```c
#include <miptrnam.h>
#include <stdio.h>
#pragma datamodel(p128)
typedef void (OS_fct_t) (void);
#pragma linkage(OS_fct_t, OS)
#pragma datamodel(pop)
int main (void)
{
    OS_fct_t *os_fct_ptr; char pgm_name[10];
    printf("Enter the program name: \n");
    scanf("%s", pgm_name);
    /* Dynamic assignment of a system pointer to program "MYPGM" */
    /* in *LIBL. The rslvsp MI library function will resolve to */
    /* this program at runtime and return a system pointer to */
    /* the program object. */
    os_fct_ptr = rslvsp(_Program, pgm_name, "*LIBL", _AUTH_OBJ_MGMT);
    os_fct_ptr(); /* OS-linkage *PGM call using a */
                   /* pointer. */
}
```

図 8-30. IBM i プログラムへのポインターを関数ポインターとして宣言するための ILE C ソース

```c
#include <miptrnam.h>
#include <stdio.h>
extern "OS" typedef void (OS_fct_t) (void);
int main (void)
{
    OS_fct_t *os_fct_ptr;
    char pgm_name[10];
    printf("Enter the program name: \n");
    scanf("%s", pgm_name);
    /* Dynamic assignment of a system pointer to program "MYPGM" */
    /* in *LIBL. The rslvsp MI library function will resolve to */
    /* this program at runtime and return a system pointer to */
    /* the program object. */
    os_fct_ptr = rslvsp(_Program, pgm_name, "*LIBL", _AUTH_OBJ_MGMT);
    os_fct_ptr(); /* OS-linkage *PGM call using a pointer */
}
```

図 8-31. IBM i プログラムへのポインターを関数ポインターとして宣言するための ILE C++ ソース
ポインターをキャストする

C 言語において、キャストとは、データ・オブジェクトを別のデータ・タイプとして一時的に表示するための構造のことです。

ポインターをキャストする際、特に非データ・オブジェクト・ポインターの場合、以下の特性と制約を考慮してください。

- ポインターを、同じ IBM i ポインター・タイプの別のポインターにキャストすることができます。
  注: ILE C コンパイラが式の中にタイプの不一致を検出した場合、コンパイル時エラーが発生します。
  - オープン (void) ポインターは、任意のタイプのポインターを保持することができます。オープン・ポインターを他のポインター・タイプにキャストしても、また他のポインター・タイプをオープン・ポインターにキャストしても、コンパイル時エラーにはなりません。
  注: コンテキストに適していない値がポインターに含まれている場合、実行時例外を受け取る可能性があります。

- 有効なデータ・オブジェクト・ポインターを符号付き整数型または符号なし整数型に変換する際、戻り値はポインターのオフセットになります。ポインターがヌル値の場合、この変換で返される値はゼロ (0) です。
  注: 変換元が NULL ポインターであるのか、オフセット 0 の有効なポインターであるのかを判別することはできません。

- 有効な関数 (procedure) ポインター、システム・ポインター、呼び出しポインター、ラベル・ポインター、または中断ポインターを符号付きまたは符号なし整数型に変換する際、その結果は常にゼロです。

- 有効なスペース・アドレスを含むオープン・ポインターを変換する際、戻り値はそのアドレスに含まれるオフセットになります。

- 整数をポインターに変換することはできますが、その結果のポインター値を逆参照することはできません。そのようなポインターの右側の 4 バイトには元の整数値が含まれており、ポインターを整数に変換し直すことでこの値をリカバリすることができます。
  注: この点は、ILE C の以前のバージョンで示されていた方から変更になりました。以前のバージョンでは、整数からポインターへの変換を行うと、その結果は必ず NULL ポインター値になってしまいました。

例:

8-43 ベージの図 8-32 は IBM i ポインターのキャストを示しています。
例：IBM i ポイントを動的プログラム呼び出しの引数として別の ILE C プログラムに渡す

以下の例では、IBM i ポインタを動的プログラム (OS リンケージ) 呼び出しの引数として別の ILE C プログラムに渡す方法を説明しています。

この例は、2 つの ILE C プログラムから成っています。プログラム T1520DL8 は、いくつかのタイプの IBM i ポインタを引数としてプログラム T1520DL9 に渡します。プログラム T1520DL9 はその引数を受け取り、それらをチェックして正常に渡されたことを確認します。

手順
1. MYLIB ライブラリーを作成するには、次のように入力します。

   ADDLIB LIB(MYLIB)

2. 8-44 ページの図 8-33 に示されたソースを使用してプログラム T1520DL8 を作成するには、次のように入力します。

   CRTBNDC PGM(MYLIB/T1520DL8) SRCFILE(QCPPLE/QACSRC)

3. 8-45 ページの図 8-34 に示されたソースを使用してプログラム T1520DL9 を作成するには、次のように入力します。

   CRTBNDC PGM(MYLIB/T1520DL9) SRCFILE(QCPPLE/QACSRC)

4. プログラム T1520DL8 を実行するには、次のように入力します。

   CALL PGM(MYLIB/T1520DL8)

出力は以下のようにになります。
Pointers passed correctly.
実行キーを押して画面セッションを終了してください。

ソース・コードのサンプル

```c
/* This program passes several types of pointers as arguments */
/* to another ILE C program T1520DL9. */
#include <stdio.h>
#include <pointer.h>
#pragma(p128)
typedef struct {
    _SPCPTR spcptr; /* A space pointer. */
    _SYSPTR sysptr; /* A system pointer. */
    void (*fnptr)(); /* A function pointer. */
} PtrStructure;
#pragma linkage (T1520DL9, OS)
#pragma datamodel(pop)
void T1520DL9 (PtrStructure *, _SPCPTR, _SYSPTR, void (*)());
void function1(void) /* A function definition. */
{
    printf("Hello!\n");
}
int main(void)
{
    int i = 4;
    PtrStructure ptr_struct;
    /* Make assignments to the fields of ptr_struct. */
    ptr_struct.spcptr = (_SPCPTR)&i; /* A space pointer. */
    ptr_struct.sysptr = (_SYSPTR)T1520DL9; /* A system pointer. */
    ptr_struct.fnptr = &function1; /* A function pointer. */
    /* Call T1520DL9, passing the address of ptr_struct and other */
    /* valid pointer arguments. */
    T1520DL9(&ptr_struct, (_SPCPTR)&i, (_SYSPTR)T1520DL9, &function1);
}
```

図8-33. T1520DL8 — IBM i ポインターを使用する ILE C ソース
/* This program receives the arguments from T1520DL8 and checks them */
/* to make sure they were passed correctly. */
#include <stdio.h>
#include <pointer.h>
type define struct {
  _SPCPTR spcptr; /* A space pointer. */
  _SYSPTR sysptr; /* A system pointer. */
  void (*fnptr)(); /* A function pointer. */
} PtrStructure;
int main( int argc, char **argv)
{
  _OPENPTR openptr; /* An open pointer. */
  _SPCPTR spcptr; /* A space pointer. */
  _SYSPTR sysptr; /* A system pointer. */
  void (*fnptr)(); /* A function pointer. */
  PtrStructure *ptr_struct_ptr;
  int error_count = 0;
  /* Receive the structure pointer passed into a local variable. */
  ptr_struct_ptr = (PtrStructure *)argv[1];
  /* Receive the pointers passed into an open pointer, 
   * then assign them to pointers of their own type. */
  openptr = (_OPENPTR)argv[2];
  spcptr = openptr; /* A space pointer. */
  openptr = (_OPENPTR)argv[3];
  sysptr = openptr; /* A system pointer. */
  openptr = (_OPENPTR)argv[4];
  fnptr = openptr; /* A function pointer. */
  /* Test the pointers passed with the pointers in ptr_struct_ptr. */
  if (spcptr != ptr_struct_ptr->spcptr)
    ++error_count;
  if (sysptr != ptr_struct_ptr->sysptr)
    ++error_count;
  if (fnptr != ptr_struct_ptr->fnptr)
    ++error_count;
  if (error_count > 0)
    printf("Pointers not passed correctly.\n");
  else
    printf("Pointers passed correctly.\n");
  return;
}

図 8-34. T1520DL9 — IBM i ポインターを使用する ILE C ソース

**ILE C/C++ 呼び出し規則の使用**

ILE C/C++ コンパイラーを使用してコンパイルされるプログラムに適用される ILE 呼び出し規則について説明します。複数言語アプリケーションに適用される呼び出し規則については、8-75 ページの『複数言語アプリケーションの処理』を参照してください。

このトピックでは以下について説明します。
- プログラムおよびプロシージャーの呼び出し
- 残っているプログラムおよびプロシージャー
- ライブラリー修飾があるプログラムの呼び出し (バインド可能 API の使用)
- ILE C からの ILE C++ プログラムの呼び出し
・**ILE C からの C++ クラスへのアクセス**

注: 用語パラメーター と引数 は、同じ意味で使用しています。

**プログラムおよびプロシージャーの呼び出し**

ILE では、プログラム処理はプロシージャー・レベルで行われます。ILE プログラムは 1 つ以上のモジュールから成り、モジュールは 1 つ以上のプロシージャーから成ります。

ILE では、プログラム (*PGM) または ILE プロシージャーのいずれも呼び出すことができます。呼び出し側プログラムは、呼び出し元は、呼び出しステートメントのターゲットがプログラムであるか、ILE プロシージャーであるかを識別しなければなりません。

C/C++ ILE 規則は、呼び出し側プログラムおよび呼び出し側 ILE プロシージャーによって異なります。

ILE C および C++ モジュールに含めることができる main() プロシージャーは 1 つのみですが、従属プロシージャー（関数）は多数含めることができます。この他の特定の ILE 言語では、1 つのプロシージャーのみが許可されます。

プログラムおよびプロシージャーの呼び出しについて詳しくは、『ILE 概念』を参照してください。

**動的プログラム呼び出しの使用**

動的プログラム呼び出しを使用して、OPM、EPM、または ILE プログラムを呼び出すことができます。

OPM および EPM プログラムとは異なり、ILE プログラムは、動的プログラム呼び出しの使用に制限されています。ILE プログラムでは、他のプロシージャーを呼び出すために、静的プロシージャー呼び出しまたはプロシージャー・ポインタ呼び出しも使用することができます。

動的プログラム呼び出しはプロシージャー呼び出しの特別な形式です。すなわち、これはプログラム呼び出しに対する呼び出しです。プログラム呼び出しのプロシージャーに対する呼び出しです。プログラム呼び出しのプロシージャーは、プログラム作成時に指定され、プログラムが呼び出される時に制御を受け取るプロシージャーです。つまり、プログラム呼び出しのプロシージャーを呼び出すのは、別のプログラムの main() 関数を呼び出すのと同じことです。

動的プログラム呼び出しが実行されると、呼び出されたプログラム名は、呼び出しプログラムが呼び出されたプログラムに初めて制御を渡す直前に、実行時のアドレスに解決されます。

動的プログラム呼び出しには以下に対する呼び出しが含まれます。

- **ILE プログラム**

注: ILE プログラムが呼び出される時に、プログラム呼び出しのプロシージャーはプログラム・パラメーターを受け取り、プログラムの初期制御権を与えられます。また、プログラム内のすべてのプロシージャーがプロシージャー呼び出し用に使用可能です。

- **EPM プログラム**

- **OPM プログラム**

- **バインド不可能 API**

**ILE 呼び出しスタックを使用してプログラム・フローを制御する仕組み**

呼び出しスタックとは、呼び出しスタック項目のリストであり、後入れ先出し (LIFO) 順序に従います。呼び出しスタック項目は、プログラムまたはプロシージャーの呼び出しです。1 つのジョブにつき 1 つの呼び出しスタックがあります。
ILE プログラムが呼び出されると、最初にプログラム入力プロシージャーが呼び出しスタックに追加されます。プログラム入力プロシージャーの呼び出し後に、スタックにプッシュされるプログラム内のメイン入口点（C または C++ の場合は、main()) に制御が渡されます。

図 8-35 に、2 つのモジュール（プログラム入力プロシージャーおよび関連ユーザープロシージャーが含まれた C++ モジュールと、通常のプロシージャーが含まれた C モジュール）から成る ILE プログラムを呼び出す OPM プログラムで構成されるプログラムの呼び出しスタックを示します。最新の項目は、スタックの最下部にあります。

図 8-35. 呼び出しスタック上のプログラム呼び出しとプロシージャー呼び出し

注: ユーザープロシージャー（UEP）の呼び出しは自動的に行われるので、動的プログラム呼び出しでは、プログラム入力プロシージャーと UEP の呼び出しが同時に行われます。呼び出しスタックに関する後の図では、動的プログラム呼び出しの 2 つのステップが結合されています。

呼び出しスタックについて詳しくは、「ILE 概念」を参照してください。

プログラムおよびプロシージャーの名前変更

以下の理由のため、プログラムまたはプロシージャー（関数）を名前変更することが必要な場合があります。

- ILE プロシージャーの名前を分かりやすいものにして、保守のために識別しやすくする。例えば、ILE プロシージャー名 QRZ1233 を chmod に名前変更できます。
- 正しくない文字が含まれているプログラムの名前を変更する。例えば、A-B は、C++ 言語では無効な名前です。

#pragma map ディレクティブを使用して、プログラムで内部 ID を IBM i 準拠の名前（プログラム名の場合 10 文字以下、ILE プロシージャー名の場合 1 文字以上）にマップすることができます。

注: 構文および説明については、「ILE C/C++ コンパイラー参照」を参照してください。

#pragma map ディレクティブは、単一コンパイル単位内で、ソース・ファイル内の任意の場所に配置できます。名前付けオブジェクト、関数、または演算子の宣言または定義の前に配置できます。ディレクティブ内に現れる ID は、プロトタイプ引数リストで使用されるすべてのタイプ名を含めて、実際の発生点と無関係に、ディレクティブがファイル・スコープで現れた場合と同様に解決されます。
例については、図 8-36 を参照してください。このコードでは、func() という名前の 2 つの関数があります。1 つは通常の関数（プロトタイプ化された int func(int)）で、もう 1 つは、クラス・メンバー関数 void func(void) です。混乱しないようにするために、#pragma map ディレクティブを使用して、funcname1 および funcname2 に名前変更しています。

```cpp
int func(int);

class X
{
    public:
        void func(void);
        #pragma map(func, "funcname1")  // maps ::func
        #pragma map(X::func, "funcname2")  // maps X::func
    };
```

図 8-36. #pragma map ディレクティブを使用して関数を名前変更する例:

注: マッピングは、パラメーター・タイプおよびスコープに基づくことができます。例えば、void func(int), void func(char), #pragma map(func(int),"intFunc"), #pragma map(func(char), "charFunc") , これは、*PRV オプションを使用した場合、機能しません。

ライブラリー修飾のあるプログラムの呼び出し

ライブラリー修飾とともにバインド可能 API を使用して、ライブラリー修飾のあるプログラムを呼び出すことができます。

例:

プログラム T2123API は、DSM ILE バインド可能 API 呼び出しを使用して、ウィンドウを作成し、入力された内容をエコー出力します。「System API Reference」に、ILE バインド可能 API に関する情報が含まれています。DSM API のプロトタイプは、<qsnsess.h> ヘッダー・ファイルにあります。API ごとに「extern "OS nowiden" return type API(arg_list);」が指定されています。ここで、return type は void または API によって返されるタイプで、arg_list は API が取るパラメーターのリストです。これにより、すべての値の引数が、暗黙的に値によって渡されます。

```cpp
// This program uses Dynamic Screen Manager API calls to
// create a window and echo whatever is entered. This is an
// example of bound API calls. Note the use of extern linkage
// in the <qsnsess.h> header file. OS, nowiden ensures that a
// pointer to an unwidened copy of the argument is passed to the
// API.
// Use BNODIR(QSNAPI) on the CRTPGM command to build this
// example.

#include <stddef.h>
#include <string.h>
#include <iostream.h>
#include <qsnapi.h>

#define BOTLINE " Echo lines until: PF3 - exit"

// DSM Session Descriptor Structure.

typedef struct{
    Qsn_Ssn_Desc_T sess_desc;
    char buffer[300];
}storage_t;

void F3Exit(const Qsn_Ssn_T *Ssn, const Qsn_Inp_Buf_T *Buf, char *action)
{
    *action = '1';
}
```

8-48 IBM i: ILE C/C++ プログラマーの手引き
```c
int main(void)
{
    int i;
    storage_t storage;
    // Declarators for declaring windows. Types are from the <qsnsess.h>
    // header file.
    Qsn_Inp_Buf_T input_buffer = 0;
    Q_Bin4 input_buffer_size = 50;
    char char_buffer[100];
    Q_Bin4 char_buffer_size;
    Qsn_Ssn_T session1;
    Qsn_Ssn_Desc_T *sess_desc = (Qsn_Ssn_Desc_T *) &storage;
    Qsn_Win_Desc_T win_desc;
    Q_Bin4 win_desc_length = sizeof(Qsn_Win_Desc_T);
    char *botline = BOTLINE;
    Q_Bin4 botline_len = sizeof(BOTLINE) - 1;
    Q_Bin4 sess_desc_length = sizeof(Qsn_Ssn_Desc_T) +
                          botline_len;
    Q_Bin4 bytes_read;

    // Initialize Session Descriptor DSM API.
    QsnInzSsnD(sess_desc, sess_desc_length, NULL);

    // Initialize Window Descriptor DSM API.
    QsnInzWinD(&win_desc, win_desc_length, NULL);

    sess_desc->cmd_key_desc_line_1_offset = sizeof(Qsn_Ssn_Desc_T);
    sess_desc->cmd_key_desc_line_1_len = botline_len;
    memcpy(storage.buffer, botline, botline_len);
    sess_desc->cmd_key_desc_line_2_offset = sizeof(Qsn_Ssn_Desc_T) +
                                         botline_len;

    // Set up the session type.
    sess_desc->EBCDIC_dsp_cc = '1';
    sess_desc->scl_line_dsp = '1';
    sess_desc->num_input_line_rows = 1;
    sess_desc->wrap = '1';

    // Set up the window size.
    win_desc.top_row = 3;
    win_desc.left_col = 3;
    win_desc.num_rows = 13;
    win_desc.num_cols = 45;

    // Create a window session.
    sess_desc->cmd_key_action[2] = F3Exit;
    session1 = QsnCrtSsn(sess_desc, sess_desc_length,
                        NULL, 0,
                        '1',
                        &win_desc, win_desc_length,
                        NULL, 0,
                        NULL, NULL);

    if(input_buffer == 0)
    {
    
    
    }
```
ILE C からの C++ プログラムおよびプロシージャーの呼び出し

ILE C から他の ILE 言語を呼び出せることにより、C++ プログラムおよび関数も呼び出すことができます。必ず次のことを行う必要があります。

・ 呼び出したい C++ 関数をすべて外部として宣言する。
・ 呼び出しのリンクージョクを指定する。

リンクージョクの指定

#pragma linkage および #pragma argument ディレクティブを使用してリンクージョクを指定します。これらのディレクティブの使用について詳しくは、8-77 ページの表 8-6 および 8-99 ページの「ILE プログラムからの ILE C プロシージャーへのアクセス」を参照してください。

リンクージョクを指定する際は、以下の C++ の特性を考慮してください。

・ C++ プログラムは標準 OS リンクージョクの呼び出し規則を使用します。#pragma linkage ディレクティブを使用して、関数呼び出し時に外部プログラム呼び出しのフラグを立てます。
・ C++ 関数を呼び出す際は、渡されるパラメーターのタイプ、ポインターと値のどちらを使用するか、およびパラメーターを拡大するかどうかについて、送信元と受信先の両方が同意していることを確認してください。例えば、呼び出す対象の関数が extern "C nowiden" として宣言されている場合、ILE C の関数宣言で #pragma argument(func, nowiden) ディレクティブを使用する必要があります。
• extern "C" または extern "C Nowiden" を使用して C++ コード内で関数を明示的に宣言することで、
その C++ 関数を外部として宣言することができます。図 8-37 に示すように、C モジュールおよび C++
モジュールの両方で使用されるヘッダー・ファイル内の関数宣言に、#ifdef ステートメントを追加する
ことができます。

```c
#ifdef __cplusplus
extern "C" {
  #endif
  function declarations
  #ifdef __cplusplus
  }
  #endif
#endif
```

図 8-37. 外部関数として宣言された ILE C++ 関数

注: これらのステートメントは C リンケージを使用して宣言されます。

例: C++ オブジェクトを使用する ILE C プログラム

このプログラムは、C で作成されたソース・コードから C++ クラスのデータ・メンバーやアクセスする
方法を示しています。

プログラム構造: プログラムは以下のファイルで構成されています。

• 以下を含む C++ ソース・ファイル hourclas.cpp
  - 1 つの基底クラス HourMin ならびに 2 つの派生クラス HourMinSec1 および HourMinSec2 の定義
  - extern "C" リンケージを持つ以下の 3 つの関数プロトタイプ
    - extern "C" void CSetHour(HourMin *)
    - extern "C" void CSetSec(HourMin *)
    - extern "C" void CSafeSetHour(HourMin *)
  - extern "C" リンケージ、extern "C" void CXXSetHour(HourMin * x) を持つ関数の定義
  - プログラム・ロジックを含む main() 関数

注: これらの関数は C++ ソース・ファイル hourclas.cpp の定義を参照してください。

• 以下を含む C ソース・ファイル hour.c
  - ファイル hourclas.cpp の C++ クラス HourMin にマップされる構造体 CHourMin
  - hourclas.cpp で宣言された、extern "C" リンケージを持つ 3 つの関数の定義
  - 関数 CSafeSetHour() の定義

C ソース・ファイル hour.c が使用する C++ ソース・ファイル hourclas.cpp の定義

```c
#include <iostream.h>

class HourMin { // base class
  protected:
    int h;
    int m;
  public:
    void set_hour(int hour) { h = hour % 24; } // keep it in range
    int get_hour() { return h; }
    void set_min(int min) { m = min % 60; } // keep it in range
    int get_min() { return m; }
    HourMin(): h(0), m(0) {}
```

IBM i 機能を使用した処理 8-51
```cpp
void display() { cout << h << ':' << m << endl; }
}

// derived from class HourMin
class HourMinSec1 : public HourMin {
    public:
        int s;
        void set_sec(int sec) { s = sec % 60; } // keep it in range
        int get_sec() { return s; }
        HourMinSec1() { s = 0; }
        void display() { cout << h << ':' << m << ':' << s << endl; }
    }

// has an HourMin contained inside
class HourMinSec2 {
    private:
        HourMin a;
        int s;

    public:
        void set_sec(int sec) { s = sec % 60; } // keep it in range
        int get_sec() { return s; }
        HourMinSec2() { s = 0; }
        void display() { cout << a.get_hour() << ':' << a.get_min() << ':' << s << endl; }
    }

extern "C" void CSetHour(HourMin *
); // defined in C
extern "C" void CSetSec(HourMin *
); // defined in C
extern "C" void CSafeSetHour(HourMin *
); // defined in C

// wrapper function to be called from C code */
extern "C" void CXXSetHour(HourMin * x) {
    x->set_hour(99); // much like the C version but the C++
    // member functions provide some protection
    // expect 99 % 24, or 3 to be the result
}

// other wrappers may be written to access other member functions
// or operators ...

main() {
    HourMin hm;
    hm.set_hour(18); // supper time;
    CSetHour(&hm); // pass address of object to C function
    hm.display(); // hour is out of range

    HourMinSec1 hms1;
    CSetSec((HourMin *) &hms1)
    hms1.display();

    HourMinSec2 hms2;
    CSetSec(&hms2);
    hms2.display();

    CSafeSetHour(&hm); // pass address to a safer C function
    hm.display(); // hour is not out of range
}
```
/* C code  hour.c */

struct CHourMin {
    int hour;
    int min;
};

void CSetHour(void * v) {
    struct CHourMin * p;
    p = (struct CHourMin *) v; // force it to the type we want
    p->hour = 99; // with power comes responsibility (oops!)
}

struct CHourMinSec {
    struct CHourMin hourMin;
    int sec;
};

// handles both HourMinSec1, and HourMinSec2 classes

void CSetSec(void *v) {
    struct CHourMinSec * p;
    p = (struct CHourMinSec *) v; // force it to the type we want
    p->sec = 45;
}

void CSafeSetHour(void *v) {
    struct CHourMin * p;
    p = (struct CHourMin *) v; // force it to the type we want
    // ... do things with p, but be careful
    // ... use a C++ wrapper function to access C++ function members
    CXXSetHour(p); // almost the same as p->hour = 99
}

/* C++ ソース・ファイル hourclas.cpp の定義を使用する C ソース・ファイル hour.c */

プログラム・フロー: 8-31 ページの「C ソース・ファイル hour.c が使用する C++ ソース・ファイル hourclas.cpp の定義」で示すことわり、main() C++ 関数は以下を行います。

1. 基底クラス HourMin のオブジェクト hm をインスタンス化する
2. 基底クラス内の h 変数（時間）に値を割り当てる
3. 基底クラスのアドレスを、C ソース・ファイル hour.c で定義された関数 CSetHour() に渡す。これにより、基底クラス内の h に新しい値が割り当てられる
4. 基底クラス内の h の値を表示する
5. 派生クラス HourMinSec1 のオブジェクト hms1 をインスタンス化する
6. このオブジェクト・クラスのアドレスを、C ソース・ファイル hour.c で定義された関数 CSetSec() に渡す。これにより、オブジェクト内の s に値が割り当てられる
7. オブジェクト内の s の値を表示する
8. クラス HourMin を含むクラス HourMinSec2 のオブジェクト hms2 をインスタンス化する
9. この新しいオブジェクトのアドレスを、C ソース・ファイル hour.c で定義された関数 CSetSec() に渡す。これにより、オブジェクト内の s に値が割り当てられる
10. オブジェクト内の s の値を表示する
11. 基底クラス・オブジェクトのアドレスを、C ソース・ファイル hour() で定義された関数 SafeSetHour() に渡す。この関数は、そのアドレスを C++ ソース・ファイル hourclas.cpp の C++ ソース・ファイル hourclas.cpp の定義を参照して定義された関数 CXXSetHour() に戻す

プログラム出力: プログラム出力は次のようにになります。

99:00
0 -0:45
0 -0:45
3 -0:45
実行キーを押して終了セッションを終了してください。

### ILE C から C++ クラスにアクセスする

既存の C++ クラスに他の言語 (ILE C など) からアクセスすることができますが、これらのクラスのデータ・メンバーを表示して操作するには、独自の関数を作成する必要があります。

クラス MyClass の共有 C/C++ ヘッダーは、例えば以下のようなになります。

```c
/* myclass.h */
#ifdef __cplusplus
class MyClass {
public:
    MyClass() {
        n = new int[100];
    }
    ~MyClass() {
        delete [] n;
    }
    int &operator[] (int i) {
        return n[i];
    }
private:
    int *n;
};
#else
struct MyClass;
MyClass *createMyClass();
void destroyMyClass(MyClass*);
int *MyClassIndex(int);
#endif
```

図8.39 共有 C/C++ ヘッダー・ファイルの例

### C 構造体への C++ クラスのマップ

仮想関数のない C++ クラスは、対応する C 構造体にマップできませんが、両データ・タイプには、基本的な違いが存在します。C++ クラスには、データ・メンバー、およびこのデータ・メンバーに対してアクセスおよび操作を実行するためのデータ・メンバー関数が含まれています。対応する C 構造体には、C++ クラスに含まれているメンバー関数は含まれておらず、データ・メンバーのみが含まれています。

8-55 ページの図 8-40 に C++ クラス Class1 を示し、8-55 ページの図 8-41 に対応する C 構造体を示します。
class Class1
{
  public:
    int m1;
    int m2;
    int m3;
    f1();
    f2();
    f3();
};

例8-40. 仮想関数のない C++ クラスの例

struct Class1
{
  int m1;
  int m2;
  int m3;
};

例8-41. 仮想関数のない C++ クラスに対応する C 構造体の例

C プログラムから C++ クラスにアクセスするには、クラス・データ・メンバーを直接検査および操作するための独自の関数を作成する必要があります。

注：C++ クラスのデータ・メンバーは public、protected、または private にすることができますが、対応する C 構造体の変数は常に、パブリックにアクセス可能です。C++ 言語に組み込まれた安全機能を除去することがでできます。

メンバー関数の形式で C++ 演算子の独自定義を指定した場合、このクラスで C++ 演算子を使用できます。

他の言語でアクセスする独自の C++ クラスを作成する際には、以下の点に留意してください。
・ 静的データ・メンバーは、他の言語に渡される C++ オブジェクトの一部でないため、クラスで使用しないでください。
・ 仮想関数は、クラスと C 構造体間のデータ・メンバーの位置合わせが異なるためにデータ・メンバーにアクセスできないので、クラスで使用しないでください。

注：他の言語で作成されたプログラムがクラスのすべてのデータ・メンバーにパブリックにアクセスできるようにすることで、データの劣化化が破損する可能性があります。

例：C++ オブジェクトを使用する ILE C プログラム
このプログラムは、C で作成されたソースコードから C++ クラスのデータ・メンバーにアクセスする方法を示しています。

プログラム・ファイルと構造体: プログラムは以下のファイルで構成されています。
・ 以下を含む C++ ソース・ファイル hourclas.cpp
  - 1 つの基底クラス HourMin ならびに 2 つの派生クラス HourMinSec1 および HourMinSec2 の定義
  - extern "C" リンケージを持つ以下の 3 つの関数プロトタイプ
    - extern "C" void CSetHour(HourMin *)
    - extern "C" void CSetSec(HourMin *)
    - extern "C" void CSafeSetHour(HourMin *)
- extern "C" リンケージ、extern "C" void CXXSetHour(HourMin * x) を持つ関数の定義
- プログラム・ロジックを含む main() 関数

注: 『C ソース・ファイル hour.c が使用するクラスを含む C++ ソース・ファイル hourclas.cpp』を参照してください。

- 以下を含む C ソース・ファイル hour.c
  - ファイル hourclas.cpp の C++ クラス HourMin にマップされる構造体 CHourMin
  - hourclas.cpp で宣言された、extern "C" リンケージを持つ 3 つの関数の定義
  - 関数 CSafeSetHour() の定義

注: 8-57 ページの『C ソース・ファイル hourclas.cpp で定義された C++ クラスを使用する C ソース・ファイル hour.c』を参照してください。

プログラムの説明: 『C ソース・ファイル hour.c が使用するクラスを含む C++ ソース・ファイル hourclas.cpp』で示すとおり、hourclas.cpp プログラムの main() 関数は以下を行います。

• 基底クラス HourMin のオブジェクト hm をインスタンス化する
• 基底クラス内の h 変数（時間）に値を割り当てる
• 基底クラスのアドレスを、C ソース・ファイル hour.c で定義された関数 CSetHour() に渡す。これにより、基底クラス内の h に新しい値が割り当てられる

注: hour.c のソース・コードは 8-57 ページの『C ソース・ファイル hourclas.cpp で定義された C++ クラスを使用する C ソース・ファイル hour.c』に示しています。

• 基底クラス内の h の値を表示する
• 派生クラス HourMinSec1 のオブジェクト hms1 をインスタンス化する
• このオブジェクト・クラスのアドレスを、C ソース・ファイル hour.c で定義された関数 CSetSec() に渡す。これにより、オブジェクト内の s に値が割り当てられる
• オブジェクト内の s の値を表示する
• クラス HourMin を含むクラス HourMinSec2 のオブジェクト hms2 をインスタンス化する
• この新しいオブジェクトのアドレスを、C ソース・ファイル hour.c で定義された関数 CSetSec() に渡す。これにより、オブジェクト内の s に値が割り当てられる
• オブジェクト内の s の値を表示する
• 基底クラス・オブジェクトのアドレスを関数 SafeSetHour() に渡す。

注: 関数 SafeSetHour() は C ソース・ファイル hour() に定義されています。この関数は、そのアドレスを hourclas.cpp に定義された関数 CXXSetHour() に戻します。

C ソース・ファイル hour.c が使用するクラスを含む C++ ソース・ファイル hourclas.cpp

#include <iostream.h>

class HourMin {  // base class
protected:
    int h;
    int m;

public:
    void set_hour(int hour) { h = hour % 24; }  // keep it in range
    int get_hour() { return h; }
    void set_min(int min) { m = min % 60; }  // keep it in range
    int get_min() { return m; }
    HourMin();: h(0), m(0) {}
void display() { cout << h << ':' << m << endl; }

// derived from class HourMin
class HourMinSec1 : public HourMin {
  public:
    int s;
    void set_sec(int sec) { s = sec % 60; } // keep it in range
    int get_sec() { return s; }
  HourMinSec1() { s = 0; }
  void display() { cout << h << ':' << m << ':' << s << endl; }
}; // has an HourMin contained inside

class HourMinSec2 {
  private:
    HourMin a;
    int s;
  public:
    void set_sec(int sec) { s = sec % 60; } // keep it in range
    int get_sec() { return s; }
    HourMinSec2() { s = 0; }
    void display() {
      cout << a.get_hour() << ':' << a.get_min() << ':' << s << endl; }
};

extern "C" void CSetHour(HourMin *); // defined in C
extern "C" void CSetSec(HourMin *); // defined in C
extern "C" void CSafeSetHour(HourMin *); // defined in C

// wrapper function to be called from C code */
extern "C" void CXXSetHour(HourMin * x) {
  x->set_hour(99); // much like the C version but the C++
  // member functions provide some protection
  // expect 99 % 24, or 3 to be the result
}

// other wrappers may be written to access other member functions
// or operators ...
void CSetHour(void * v) {
    struct CHourMin * p;
    p = (struct CHourMin *) v; // force it to the type we want
    p->hour = 99; // with power comes responsibility (oops!)
}

struct CHourMinSec {
    struct CHourMin hourMin;
    int sec;
};

// handles both HourMinSec1, and HourMinSec2 classes
void CSetSec(void *v) {
    struct CHourMinSec * p;
    p = (struct CHourMinSec *) v; // force it to the type we want
    p->sec = 45;
}

void CSafeSetHour(void *v) {
    struct CHourMin * p;
    p = (struct CHourMin *) v; // force it to the type we want
    // ... do things with p, but be careful
    // ...
    // use a C++ wrapper function to access C++ function members
    CXXSetHour(p); // almost the same as p->hour = 99
}

Program Output: プログラム出力は次のようにになります。

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>99:0</td>
</tr>
<tr>
<td>0 -0 :45</td>
</tr>
<tr>
<td>0 -0 :45</td>
</tr>
<tr>
<td>3 -0</td>
</tr>
</tbody>
</table>

実行キーを押して端末セッションを終了してください。

別のプラットフォームから ILE へのプログラムの移植
新しいコードを作成する場合は、できる限り ISO のガイドラインに従い、プラットフォーム固有の拡張は避けてください。一般的に、コードは移植可能でなければなりません。

このセクションでは、以下について説明します。
- **C から ILE C++ へのコード移植の制限**
- **ILE C++ オブジェクトの呼び出しの変更**
- **BCD マクロを使用してコード化 10 進数オブジェクトを ILE C++ にポータルする**
- **C と C++ の両方で機能するヘッダー・ファイル**
- **より厳密な C++ 型検査の処理**
- **unsigned char 変数としての unsigned char ポインターの宣言**
- **C++ での文字配列の初期化**
- **ストリング・リテラルへのアクセス権限の指定**
- **単一活動化グループへの有効範囲設定によるキャッシュされない C++ 例外の回避**

**ILE C または C++ へのコード移植の制限**
このセクションでは、ILE C および C++ へのコード移植の制限について説明します。

8-58 IBM i: ILE C/C++ プログラマーハの手引き
ファイルの組み込み

> C++ ILE C++ では、以下のようにします。
> - 組み込みファイル名は、有効なワークステーション・ファイル名でなければなりません (例えば、
  "file_name" や <file_name>。
- 組み込みファイルは *LIBL 値および *CURLIB 値を参照できません。

> C ILE C 組み込み名では、これらの値を使用できます。例えば、("*LIBL/ABC",
*LIBL/ABC/*All)...

プラットフォーム固有の拡張機能

_far16 および _pascal16 などのプラットフォーム固有の拡張機能は、移植できないプラットフォーム固有のポインタです。

共用体のメンバー

> C++ コンストラクターのあるクラスのオブジェクトは union のメンバーにはないため、ILE C++
の _DecimalT クラス・テンプレートを union のメンバーとして使用することはできません。

構造体のメンバー

> C++ ILE C++ では、_DecimalT クラス・テンプレートが structのメンバーである場合、その struct
を初期化指定子リストで初期化することはできません。

> C ILE C の構造体を次に示します。

typedef struct {
  char s1;
  decimal(5,3) s2;
}s_type;

s_type s ={"+", 12.345d};

図 8-42. ILE C++ に移植できない ILE C 構造体定義の例

> C++ ILE C++ では、次の図に示すようにコードを書き換える必要があります。

struct s_type {
  char s1;
  decimal(5,3) s2;
  s_type (char c, decimal(5,3) d): s1(c), s2(d) {}
};
s_type s {'+', _D("12.345")};

10 進定数

> C++ 接尾部 D または d を使用して定義された 10 進定数は、C++ の _DecimalT クラス・テンプレート
ではサポートされていません。その代わりに、バック 10 進定数を表すのに __D() で囲んだストリング
・リテラルを使用します。ILE C で定義された 10 進定数 123.456D は、ILE C++ での
__D("123.456") と同等です。

10 進定数および case ステートメント

__D マクロは、_ConvertDecimal コンストラクターを頻繁に使用する必要があるコードを単純化するために
使用されます。__D マクロは _ConvertDecimal コンストラクターと同等であるので、__D マクロは、case
ステートメントと一緒に使用することはできません。有効な case ステートメントとは、整数定数式を使用するものです。以下の図に示したコードは、コンパイラ・エラーとなります。

decimal(4,3) op;
switch int(op) {
  case int(D("1.3")):
    ..... break;
}

図 8-43 両立しない 10 進定数と case ステートメントを使用したコードの例

注: コンパイラは、case ステートメントに case 式が整数定数式ではないことを示すフラグを立つます。

**IFS 内のライブラリ QSYS.LIB**

IFS (Integrated File System) 内のライブラリ QSYS.LIB は、ストリーム・ファイルで情報を保管および操作するための共通インターフェースを提供します。C ストリーム入出力関数および C++ ストリーム入出力クラスは、統合ファイル・システムを使用して実装されます。統合ファイル・システムには、7 つのファイル・システムがあります。ライブラリ (QSYS.LIB) サイズ指数・システムは、IBM i ファイル・システムにマップされます。

- 論理ファイルがサポートされない
- サポートされる物理ファイルのタイプは、単一のフィールドが含まれたプログラム記述ファイルと、単一のテキスト・フィールドが含まれたソース物理ファイルのみである
- バイト範囲ロックはサポートされない (「システム API 解説書」)
- データベース・ファイル・メンバーが開かれているジョブがある場合、隨時そのファイルへの書き込み権限が付与されるジョブは 1 つだけであり、他のジョブには読み取りアクセスのみが許可される

**ILE C++ コンパイラを使用してコンパイルされたプログラムのストリーム入出力は、デフォルトで統合ファイル・システムを使用します。ILE C コンパイラの場合、デフォルトで C ストリーム入出力になります。データベースまたは DDM ファイルを使用しているプログラムがある場合、SYSIFCOPT(*NOIFSIO) コンパイラ・オプションの使用が最良の選択になります。これにより、統合ファイル・システムではなく、IBM i ファイル・システムを使用して既存のプログラムが確実にコンパイルされます。統合ファイル・システム内の制限されたデータベースまたは DDM ファイルを使用しているプログラムをコンパイルした場合、実行時にエラーになります。**

**テラベースに関する考慮事項**

8-176 ページの「テラベース環境でコードを移植する際のバイナリ互換性に関する考慮事項」を参照してください。

**ILE C++ オブジェクトの呼び出しの変更**

C++ オブジェクトを呼び出す ILE C ソース・コードは、ILE C++ の下に実行されるように変更する必要があります。

以下に例を示します。

- #pragma linkage ディレクティブまたは #pragma argument ディレクティブの代わりに、関数定義または宣言での extern リンケージ指定を使用する必要があります。
- #pragma map ディレクティブには、いくつかのセマンティックな違いがあります。
・#pragma argument ディレクティブと extern リンケージ指定が関数定義を処理する方法には次のような違いがあります。関数呼び出しを処理する際には、両方で同じコードが生成されますが、#pragma argument ディレクティブは、関数定義内のパラメーターに影響しません。extern リンケージ指定は、関数定義内のパラメーターに影響します。

ヘッダー・ファイルにおける相違点

ILE C では、packed decimal データ・タイプを使用する前に必ず、ヘッダー・ファイル<br><decimal.h> をソースに含める必要があります。

ILE C++ では、代わりに <bcd.h> を含める必要があります。

リンケージ指定における相違点

このセクションの図では、ILE C と ILE C++ でのリンケージ指定における相違点を示します。図 8-44、図 8-45 および図 8-46 に示したソース・コードでは同じことが実行されます。

Module1.c
extern void foo (int *i, char **s)
{
  *s = *i ? "Not Zero" : "Zero";
}
図 8-44. extern リンケージ指定を使用する ILE C ソース・コードの例

Module2.c
extern void foo (int, char *);
#pragma argument (foo, VREF)
int main()
{
  char *s;
  foo (1, s);
}
図 8-45. #pragma argument リンケージ指定を使用する ILE C ソース・コードの例

Module3.C
extern "VREF" void foo (int i, char *s)
{
  s = i ? "Not Zero" : "Zero";
}
int main()
{
  char *s;
  foo (1, s);
}
図 8-46. extern リンケージ指定を使用する ILE C++ ソース・コードの例
関数定義における相違点

このコードは、#pragma linkage ディレクティブを置き換えるために、関数定義を持つ extern "OS" がどのように使用されるかを示します。詳細については、『ILE C/C++ 呼び出し規則の使用』を参照してください。

<table>
<thead>
<tr>
<th>使用しないもの：</th>
<th>使用するもの：</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma datamodel (p128) typedef void (FUNC)(int); #pragma linkage (FUNC, OS) #pragma datamodel(pop)</td>
<td>extern &quot;OS&quot; typedef void (FUNC)(int);</td>
</tr>
<tr>
<td>typedef void (FUNC)(int) extern &quot;OS&quot; FUNC; //error</td>
<td></td>
</tr>
</tbody>
</table>

BCD マクロを使用してコード化 10 進数オブジェクトを ILE C++ に移植する

ILE バック 10 進データ型と互換性のある 2 進化 10 進数オブジェクトを作成可能にするために、IBM i 用の 2 進化 10 進数クラス・ライブラリーが提供されています。

以下の図に示すマクロは、_DecimalT クラス・テンプレートが ILE C との互換性を維持するために使用するマクロです。

#define decimal _Decimal
#define digitsof __digitsof
#define precisionof __precisionof
#define _Decimal(n,p) _DecimalT<n,p>
#define __digitsof(DecName) (DecName).DigitsOf()
#define __precisionof(DecName) (DecName).PrecisionOf()

図 8-47. ILE C から ILE C++ ヘコードを移植する BCD マクロ

例

以下の図は、ILE C と ILE C++ でバック 10 進数データ・タイプをコーディングするためのソースを示しています。

// ILE C program
#include <decimal.h>

void main()
{
    int dig, prec;
    decimal(9,3) d93;
    dig = digitsof(d93);
    prec = precisionof(d93);
}

図 8-48. コードをバック 10 進数データ・タイプに移植するための ILE C ソース・コード
// C++ program
#include <bcd.h>

void main()
{
    int dig, prec;
    _DecimalT<9,3> d93;
    dig = d93.DigitsOf();
    prec = d93.PrecisionOf();
}

図8-49. コードをバック 10 進数データ・タイプに移植するための ILE C++ ソース・コード

以下の図は、<bcd.h> で定義されたマクロを使用すると、最初のプログラムに示したのと同じ ILE C を使用できるということを示しています。

// C++ program using the macro
#include <decimal.h>

void main()
{
    int dig, prec;
    decimal(9,3) d93;
    dig = digitsof(d93);
    prec = precisionof(d93);
}

図8-50. コードを ILE C++ に移植するための BCD マクロの使用例

注: <decimal.h> に <bcd.h> ヘッダー・ファイルが含まれているので、<decimal.h> ヘッダー・ファイルが指定されています。

クラス・テンプレート・インスタンス化の ILE C 構文へのマッピング

クラス・テンプレート・インスタンス化を所要の ILE C 構文にマップするために、C++ では、以下の図に示したマクロを使用します。

#define decimal _Decimal
#define _Decimal(n,p) _DecimalT<n,p>

図8-51. C++ クラス・テンプレート・インスタンス化を ILE C 構文にマップする BCD マクロ

注: decimal(n,p) 指定子を使用する ILE C コードは、何ら変更なしに C++ に移植できます。

ILE C によってサポートされる _DecimalT<n,p> 指定子は、C++ コンパイラによってサポートされていません。_DecimalT<n,p> 指定子を使用するには、型指定子の場所にゼロを明示的に挿入する必要があります。例えば、decimal(10) を decimal(10,0) に変更する必要があります。

乗算および除算の追加精度の処理

_DecimalT クラス・テンプレートは、乗算および除算の演算の内部結果としてそれぞれ最大 62 桁および 93 桁を使用できるようにします。これは、両方の演算に最大 31 桁が使用される ILE C packed decimal データ・タイプとは異なります。

注: この内部結果は、中間結果とは異なります。内部結果は、操作時の一時結果を保管するようになってい

IBM i 機能を使用した処理 8-63
オブジェクトの桁数の決定

ILE C では、packed decimal データ・タイプで __digitsof 演算子を使用すると、結果は整数定数になります。__digitsof 演算子は、packed decimal データ・タイプまたはパック 10 進数式に適用できます。__digitsof 演算子は、packed decimal データ・タイプで n という総桁数を戻します。

packed decimal データ・タイプでの桁数 n を決定するには、図 8-52 に示した例に従ってください。

# include <decimal.h>

int n,n1;
decimal (5, 2) x;

n = __digitsof(x);  /* the result is n=5 */
n1 = __digitsof(1234.567d); /* the result is n1=7 */

図 8-52. パック 10 進数データ・タイプでの桁数を決定するコードの例

ILE C++ では、_DecimalT クラス・テンプレートでメンバ関数 DigitsOf() を使用すると、結果は整数定数になります。メンバ関数 DigitsOf() は、_DecimalT クラス・テンプレート・オブジェクトに適用できます。メンバ関数 DigitsOf() は、_DecimalT クラス・テンプレート・オブジェクトでの総桁数 n を戻します。

_DecimalT クラス・テンプレート・オブジェクトでの桁数 n を決定するには、図 8-53 に示した例に従ってください。

# include <bcd.h>

int n,n1;
_DecimalT <5, 2> x;

n = x.DigitsOf();  // the result is n=5

図 8-53. _DecimalT クラス・テンプレート・オブジェクトでの桁数を決定するコードの例

内部パック 10 進数データ・オブジェクトの桁数の決定

ILE C では、packed decimal データ・タイプで __precisionof 演算子を使用すると、結果は整数定数になります。__precisionof 演算子は、packed decimal データ・タイプまたはパック 10 進数式に適用できます。__precisionof 演算子は、packed decimal データ・タイプの小数桁数 p を示します。

packed decimal データの小数桁数 p を決定するには、図 8-54 に示した例に従ってください。

# include <decimal.h>

int p,p1;
decimal (5, 2) x;

p=__precisionof(x);  /* The result is p=2 */
p1=__precisionof(123.456d);  /* The result is p1=3 */

図 8-54. 内部パック 10 進数データ・オブジェクトの小数桁数を決定するコードの例
「C++」ILE C++では、_DecimalT クラス・テンプレートでメンバ関数 PrecisionOf()を使用すると、結果は整数定数になります。メンバ関数 PrecisionOf()は、_DecimalT クラス・テンプレート・オブジェクトに適用できます。メンバ関数 PrecisionOf()は、_DecimalT クラス・テンプレート・オブジェクトの小数桁数 p を示します。

_DecimalT クラス・テンプレート・オブジェクトの小数桁数 p を決定するには、以下に示した例に従ってください。

```c++
#include <bcd.h>

int p,p1;
_DecimalT <5, 2> x;
p=x.PrecisionOf(); // The result is p=2
```

図8-55 内部 _DecimalT クラス・オブジェクトの小数桁数を決定するコードの例

フォーマット設定された C 入出力関数の値のフォーマット設定

fprintf(), sprintf(), vprintf(), vsprintf()、および vsprintf()の関数の動作は、printf()関数と同じです。fscanf()およびsscanf()の関数の動作は、scanf()関数と同じです。

出力のフォーマットを制御するには、printf()関数の flags、width、および precision のフィールドを使用します。

scanf()関数のフォーマットを制御するには、scanf()関数の * および width のフィールドを使用します。これらのフィールドについては、「ILE C/C++ ランタイム・ライブラリ関数」を参照してください。

印刷関数フラグ: 表8-2 では、D(n,p) 变換のフラグ文字とその意味について説明します。

<table>
<thead>
<tr>
<th>フラグ</th>
<th>意味</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>小数部分がない場合にも、結果に常に小数点文字が含まれます。</td>
</tr>
<tr>
<td>0</td>
<td>フィールド幅を満たすために先行ゼロ（符号または基底の前にある）が使用されます。スペース埋め込みは行われません。</td>
</tr>
<tr>
<td>-</td>
<td>結果は、常にフィールド内で左寄せされます。</td>
</tr>
<tr>
<td>+</td>
<td>結果は、常に正負号または負符号で始まります。</td>
</tr>
<tr>
<td>スペース</td>
<td>符号付き変換の結果が符号なしの場合や、符号付き変換の結果が文字なしの場合に、結果の先頭に常にスペース文字が付けられます。</td>
</tr>
</tbody>
</table>

印刷関数のフィールド幅: printf()関数のオプションの最小フィールド幅は、_DecimalT クラス・テンプレート・オブジェクトの整数部分、小数部分、またはその両方の部分に表示される最小桁数を示します。文字列がフィールド幅に満たない場合は、フィールドにスペースが埋め込まれます。フィールド幅は * とすることができます。nが * の場合、nの値は、パラメーター・リストの対応する位置から派生します。

印刷関数のフィールド精度: printf()関数のオプションの精度は、_DecimalT クラス・テンプレート・オブジェクトの小数部分に表示される桁数を示します。デフォルトの精度は p です。精度が p を超える場合、追加のゼロが埋め込まれます。精度が p に満たない場合、丸めが実行されます。フィールドの精度は * とすることができます。pが * の場合、pの値は、パラメーター・リストの対応する位置から派生します。
変換指定子： printf() 関数の変換指定子は以下のとおりです。

D(n,p) _DecimalT クラス・テンプレート・オブジェクトは、[ - ] ddd.ddd 形式で変換されます。この場合、小数点文字の後の桁数は指定の精度と等しくなります。精度が欠落している場合、0 が採用されます。精度がゼロで # フラグが指定されていない場合、小数点文字は表示されません。小数点文字が表示される場合、その前に少なくとも 1 桁が表示されます。値は、適切な桁数に切り捨てられます。（n,p）記述子は、_DecimalT クラス・テンプレート・オブジェクトの特性を記述するために使用されます。n と p の両方を 10 進数数の形式にする必要があります。p が 11 以上の場合は、デフォルト値のゼロが使用されます。指定子が先に述べた以外の形式である場合、振る舞いは未定義のものになります。

印刷される数値の n および p が変換指定子 %D(n,p) の n および p と一致しない場合、振る舞いは未定義のものになります。_DecimalT クラス・テンプレート・オブジェクト式の結果のクラスのサイズが不明である場合は、必ず引数リストに単項演算子__digitsof (式) および __precisionof (式) を使用して D(*,*) の * を置き換えます。

scanf() 関数の変換指定子は以下のとおりです。

D(n,p) 10 進数に一致し、想定されるサブジェクト・シーケンスの形式は、オプションの正負号または負符号にオプションで小数点を含む空ではない数字のシーケンスが続くという形式です。サブジェクト・シーケンスは、最初の非空白ライド文字から始まる、入カストリングの最長の初期サブシーケンスとして定義されています。これが想定される形式です。入カストリングが空であるか、すべて空白文字からなっているか、あるいは最初の非空白文字が符号、数字、または 10 進小数点文字以外である場合には、サブジェクト・シーケンスには文字は含まれません。

条件演算子の ILE C または C++ への移植

ILE C では、2 番目のオペランドと 3 番目のオペランドの両方が算術型である場合、通常の算術変換が実行されて、共通型になります。

C++ では、2 番目と 3 番目の式の両方が同じクラスである場合、2 番目または 3 番目の式をキャストして、同じクラスにする必要があります。

次の図は、2 番目と 3 番目の式が同じクラスでないために条件式が失敗する例を示しています。

```
#include <bcd.h>

main()
{
    int var_1;
    decimal(4,2) op_1_1 = _D("12.34");
    decimal(10,2) op_1_2 = _D("123.45");
    var_1 = (op_1_1 < op_1_2) ? (op_1_1 + 3) : op_1_2;
}
```

図 8-56. クラスの違いが原因で失敗する条件式の例

_DecimalT クラス・テンプレートで条件演算子を使用するには、以下のいずれかを行ってください。

・ 2 番目の式に明示的キャストを使用し、3 番目の式と同じタイプにする。
・ 同じタイプの変数を使用する。

式間のクラスの相違を解決する明示的キャストの例: 以下の図は、3 番目の式と同じクラスを持つようにする 2 番目の式の明示的キャストを示しています。
#include <bcd.h>

main()
{
    int var_1;
    decimal(4,2) op_1_1 = __D("12.34");
    decimal(10,2) op_1_2 = __D("123.45");
    var_1 = (op_1_1 < op_1_2) ? (_DecimalT<10,2>)__D(op_1_1 + 3) : op_1_2;
}

図8-57 式間のクラスの相違を解決する明示的キャストの例

一貫性のある変数型の使用例: 以下の図は、同じ型の変数の使用法を示しています。

#include <bcd.h>

main()
{
    int var_1;
    decimal(10,2) op_1_1 = __D("12.34");
    decimal(10,2) op_1_2 = __D("123.45");
    var_1 = (op_1_1 < op_1_2) ? op_1_1 : op_1_2;
}

図8-58 一貫性のある変数型の使用例

注: (op_1_1 + 3) は _Decimal<13,2> という結果になるため、2 番目の式からは + 3 が除去されています。

ILE C のバック 10 進数データ・タイプを _DecimalT クラス・テンプレートに移植

クラス・テンプレート _DecimalT では、_DecimalT からクラス・テンプレート・インスタンス化を取得するためにコンストラクターも代入演算子も多重定義されていません。このため、1 つの _DecimalT クラス・テンプレート・オブジェクトから別の _DecimalT クラス・テンプレート・オブジェクトへの変換を含む明示的な型キャストを直接実行することはできません。代わりに、マクロ __D を使用して明示的な型キャストを必要とする式を取り込む必要があります。以下の図に示したこのプログラムは、ILE C で作成されています。

#include <decimal.h>

void main ()
{
    decimal(4,0) d40 = 123D;
    decimal(6,0) d60 = d40;
    d60 = d40;
    decimal(8,0) d80 = (decimal(7,0))1;
    decimal(9,0) d90;
    d60 = (decimal(7,0))12D;
    d60 = (decimal(4,0)d80;
    d60 = (decimal(4,0))(d80 + 1);
    d60 = (decimal(4,0))(d80 + (float)4.500);
}

図8-59 バック 10 進数データ・タイプを使用する ILE C コード

このソースは、以下の図で示されているように ILE C++ で再作成する必要があります。
```c
#include
int main ( void )
{
    _DecimalT<4,0> d40 = __D("123"); // OK
    _DecimalT<6,0> d60 = __D(d40); // Because no constructor exists that can convert
d40 to d60. macro __D is needed to convert d40 into
// an intermediate type first.

    _DecimalT<8,0> d80 = (_DecimalT<8,0>); // OK
    // Type casting an int, not a decimal(n,p)
    d60 = d40; // OK. This is different from
    // the second statement in
    // which the constructor was
called. In this case, the
    // assignment operator is called
    // and the compiler converts
d40 into the intermediate
    // type automatically.

    _DecimalT<9,0> d90;
    d60 = (_DecimalT<7,0>); // "12"
    d60 = (_DecimalT<4,0>); // needed to convert d60 into
    // _DecimalT<n,p>. macro __D is
    // needed to convert
    // them to an intermediate
    // type first.
}
```

図8-60. C パック10進数データ・タイプの代わりに_DecimalT クラス・テンプレートを使用する ILE C++ コード

バック構造の使用における相違点

_Packed_キーワードは、埋め込みを無視して、構造をできるだけ圧縮するようにコンパイラーに指示します。

IEL Cでは、このキーワードは、構造定義および型定義で使用できます。

IEL C++では、同じキーワードを型定義でしか使用できません。

<table>
<thead>
<tr>
<th></th>
<th>ILE C</th>
<th>ILE C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef _Packed struct {</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>)ps_t;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_Packed struct {</td>
<td>ok</td>
<td>エラー</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>)ps_v;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

したがって、_Packed_キーワードは、ヘッダー・ファイルにある型定義でのみ使用されるようにする必要があります。
ILE C/C++ コンパイラでは、#pragma pack ディレクティブは C プログラムのみに適用されます。ILE C の #pragma pack ディレクティブは、Windows の #pragma pack ディレクティブと互換性はありません。

エラー検査における相違点
このセクションでは、以下に関するエラー検査について説明します。

- 無効な 10 進数形式
- 数学演算子

無効な 10 進数形式: ILE C では、パック 10 進数は、ネイティブ・データ・タイプとして実装されます。これにより、無効な 10 進数形式などのエラーは、コンパイル時に検出されるようになります。

C++ では、同様のエラーの検出は、以下の例に示すように実行時まで抑えられます。

```c
#define _DEBUG 1
#include <bcd.h>

int main ( void )
{
    _DecimalT<10,20> b = __D("ABC"); // Runtime exception is raised
}
```

and

```c
#define _DEBUG 1
#include <bcd.h>

int main ( void )
{
    _DecimalT<33,2> a; // Max. dig. allow is 31. Again,
    // runtime exception is raised
}
```

注: 一部のエラーはコンパイル時に検出できます。例えば、n<1, (_Decimal<-33,2>) です。

数学演算子: ILE C には、パック 10 進数オペランドの符号または数字のコードに関する追加のエラー検査が備わっています。有効な符号は 16 進数 A から F です。有効な数字の範囲は 16 進数 0 から 9 です。10 進数オペランドがエラー状態である場合、ILE C はエラー・メッセージを生成します。この追加の検査は _Decimal1T クラス・テンプレートにはありません。

C++ 以下のコードの場合、結果として、ILE C ではエラー・メッセージが生成されますが、ILE C++ では生成されません。

```c
#include <decimal.h>

int main ( void )
{
    decimal(10,2) a, b;
    int c;
    c = a > b; // a and b are not valid packed decimals because
    // a and b are not initialized
}
```

C と C++ の両方で機能するヘッダー・ファイル
C ヘッダー・ファイルは通常、C++ では使用できません。構造体、共用体、タイプ定義、および変数は、問題なく使用できる可能性があります。関数プロトタイプおよび pragma で注意を払う必要があります。
二重関数プロトタイプの使用

ヘッダー・ファイルを ILE C コンパイラおよび ILE C++ コンパイラで使用できるようにするには、以下の例に示すように "OS" リンケージ・タイプを持つすべての関数に二重プロトタイプが必要です。

```c
#ifdef __cplusplus
    extern "linkage-type" //linkage type "OS"
#else
    #pragma linkage(function_name,linkage_type)
#endif
void function_name(...);
```

図 8-61．ヘッダー・ファイルを ILE C と ILE C++ の両方で使用できるようにする二重プロトタイプの単一セットの例

二重プロトタイプを持つ関数のリストがある場合、以下の例に示した構文を使用できます。

```c
#ifdef __cplusplus
    extern "linkage-type" { //linkage type "OS"
#else
    #pragma linkage(function_name1, linkage_type)
    ...
    #pragma linkage(function_nameN, linkage_type)
#endif
void function_name1(...);
    ...
void function_nameN(...);
#ifdef __cplusplus
}
#endif
```

図 8-62．ヘッダー・ファイルを ILE C と ILE C++ の両方で使用できるようにする二重プロトタイプの複数セットの例

ILE C プログラムで C++ リンケージ関数にアクセスできるようにする

> C++

ヘッダー・ファイルを以下の例に示すように構造内で折り返します。
### 注:
1. リンケージ指定 `extern "C"` は、プロトタイプ化されたすべての関数が C リンケージを持つことをコンパイラに指示します。C++ リンケージ関数は、C++ 内部名を使用して C から呼び出すことができません。他の言語からの関数呼び出しについては、8-45 ページの「ILE C/C++ 呼び出し規則の使用」を参照してください。
2. マクロ `__ILEC400__` で `__cplusplus` を置き換えることができますが、他の実装環境に移植可能であるので `__cplusplus` をお勧めします。

### QSYSINC ヘッダー・ファイルの組み込み

`<file/header.h>` を使用する必要があります。例えば、QSYSINC/MIH/SYSEPT を組み込むには、`#include <mih/sysept.h>` を使用できます。

### より厳密な C++ 型検査の処理

C++ では、C より厳密に型検査が行われます。

この部では、以下の方法について説明します。

- `#pragma enum` ディレクティブを使用して、整数データ型サイズの問題を解決する
- さまざまなシナリオの非互換のポインター・タイプを解決する

### 整数データ・タイプのサイズ問題の解決

`<mih/sysept.h>` では、列挙サイズは、`#pragma enum` ディレクティブが使用されない限り、常に整数のサイズです。

enum タイプのサイズに関する問題を解決するには、以下の図に示すように `#pragma enum` ディレクティブを使用します。
非互換ポインター型の解決

<table>
<thead>
<tr>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO C では、void を指すポインターは、他の任意の型のポインターに割り当てることができます。ポインターを示的にキャストする必要はありません。</td>
<td>C++ では、void ポインターを、他の void ポインターのみに割り当てることができます。void ポインターを戻すC メモリー間数 (malloc(), calloc(), realloc() など) を使用する場合、コードのコンパイル前に、各 void ポインターを適切なポインター型にキャストする必要があります。注：malloc() や free() の代わりに、new や delete という演算子を使用できます。</td>
</tr>
<tr>
<td>C コンパイラは、memcmp() を使用して定数 char 配列を揮発性 char 配列と比較するソース・コードをコンパイルします。</td>
<td>memcmp() を使用して定数 char 配列を揮発性 char 配列と比較するソース・コードをコンパイルしようとすると、C++ コンパイラはエラー・メッセージ (例えば、volatile unsigned char は不能を変換されるキャスト) を生成します。コードのコンパイル前に void ポインターを適切なポインター型にキャストしない限り、volatile ポインターが想定される場合に constant ポインターを使用できません。注：malloc() や free() の代わりに、new や delete という演算子を使用できます。</td>
</tr>
</tbody>
</table>

注: 互換性について詳しくは、「ILE C/C++ 解説書」を参照してください。

未定義名エラーを回避するための名前マングリングの無効化

すべての C++ 関数名は、関数の多重定義を有効にするためにマングルされます。ILE C/C++ 関数をマングル名 (例えば、LocateSpaces__FPc) にバインドすると、未定義名エラーを受け取ります。

ILE C では、サービス・プログラム関係は、LocateSpaces__FPc == LocateSpaces または LocateSpace__FPc == LocateSpace です。ILE C コードを移植し、関数名マングリングを無効にする場合は、関数名について extern "C" を使用します。

C++ 関数プロトタイプとのタイプ不一致の解

ILE C++ では、以下が必要です。
- 完全なプロトタイプ宣言

8-72 IBM i: ILE C/C++ プログラマーの手引き
・関数のすべての宣言は、関数の固有の定義に一致している必要があります
・ポインター宣言で定義されているタイプは、関数プロトタイプで定義されているタイプに一致している必要があります

注: ISO C にはこのような制限はありません。

関数プロトタイプ不一致の例

#include <signal.h>
void (*sig_handler)(int);  // 1
typedef void (*SIG_T)();  // function pointer typedef of type void (*) ()
extern "C" void (*signal (int, void(*) (int))) (int);  // function pointer
  // prototype with return
  // type void(*) (int)
SIG_T oldsig = signal (SIGALL, sig_handler);  // function pointer definition
  // of type SIG_T

図 8-65. 型不一致の例

注:
1. 関数ポインター・プロトタイプ ("void(*) ()") に定義された型と関数プロトタイプ (void (*) (int)) の型定義の間の型不一致のため、この例では以下のエラーが生成されます。
   CZP0257(30) An object or reference of type "void(*) ()" cannot be initialized with an expression of type "extern "C" void (*) (int)"
2. int パラメーターは、型定義 SIG_T に存在しません。

関数プロトタイプ不一致の処理
型不一致を処理するには、以下のことを行うことができます。
・キャスト式を使用してポインターを別の型の変数に割り当てる。
・デフォルトの char 型を設定するように C++ ソースをコンパイルするときに、DFTCHAR コンパイラーオプションを使用する。

注: これらのオプションについて詳しくは、「ILE C/C++ コンパイラ参調」を参照してください。

unsigned char 変数としての unsigned char ポインターの宣言

> C++

QXX 関数は、unsigned char ポインターを返します。ILE C では、signed char を unsigned char ポインターに代入できます。 C++ では、これは無効です。

以下の図に示すように、unsigned char ポインターは、ソース・コードで unsigned char 変数として宣言する必要があります。
文字配列の初期化

C++では、文字配列を初期化すると、末尾の'¥0'(char型のゼロ)がストリングの初期化指定子に付加されます。存在する配列エレメントの数を超える初期化指定子のある文字配列は初期化できません。

ISO Cにおいて、このタイプの情報では、末尾の'¥0'のスペースを省略することができます。

例えば、次の初期化はC++では無効です。

```c
```

暗黙的後書き'¥0'(char型のゼロ)のスペースがないので、この初期化はエラーになります。例えば、次の初期化はC++では有効です。

```c
char v[4] = "asd"; //valid in C++
```

注: 互換性について詳しくは、「ILE C/C++解説書」を参照してください。

ストリング・リテラルへのアクセス権限の指定

ストリングを読み取り/書き込みメモリーに入れるには、ファイル内のCまたはC++コードの前に

```c
#pragma strings
```

ディレクティブを入れる必要があります。

READONLYオプションを使用すると、コンパイラがストリングを読み取り専用メモリーに入れることを指定します。

WRITEABLEオプションを使用すると、コンパイラがストリングを書き込み可能メモリーに入れる必要があることを指定します。ストリングはデフォルトでは書き込み可能です。

Cストリングはデフォルトでは読み取り/書き込みです。

C++ストリングはデフォルトでは読み取り専用です。

注: このpragmaは、モジュールの作成またはバインド・プログラムの作成のコマンドの*STRONLYオプションをオーバーライドします。

```c
#pragma strings
```

ディレクティブの構文については、「ILE C/C++コンパイラ参照」を参照してください。
単一活動化グループへの有効範囲設定によるキャッチされない例外の回避

C++  ILE で、set_terminate() 関数の影響の有効範囲は活動化グループに設定されます。これが ILE 以外のプラットフォームから移植されたコードのコンパイルにどう影響するかについて、次の図に例を示します。

// File main.c
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <terminat.h>

void a();
void my_terminate();

int main() { 2
    set_terminate(my_terminate);
    try {
        a();
    }
    catch(...) cout << "failed" << endl;
}

// File term.c 1
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
void my_terminate() {
    cout << "failed" << endl;
}
void a() { throw 7; } 3

図 8-67. ILE に移植され、キャッチされない例外になるコードの例

図 8-67 において:
1. my_terminate() と a() はどちらも、デフォルトで単一の活動化グループ（例えば、「B」）で実行されるサービス・プログラムにあります。
2. main() プロシージャーは、デフォルトで、別の活動化グループ（例えば、「A」）で実行されます。
3. このシナリオで、set_terminate が実行される活動化グループは my_terminate() を呼び出す前に終了されるため、デフォルトの活動化処理がオーバーライドされない限り、 a() によってスローされた例外は main() までバーチャル化できません。
結果として、エラー・メッセージ CEE9901 が main() に送られます。

別のプラットフォームから ILE にコードを移植した場合は、次の場合数が同じ活動化グループで実行されるようにする必要があります。
• 例外ハンドラー (この例では、my_terminate())
• 例外をスローする関数 (この例では a())

複数言語アプリケーションの処理

このセクションでは、ILE 呼び出し規則およびそれが複数言語アプリケーションにどのように適用されるのかについて説明します。 ILE C/C++ コンパイラでコンパイルされるプログラムに適用される呼び出し規則については、8-45 ページの『ILE C/C++ 呼び出し規則の使用』を参照してください。
動的プログラム呼び出しを使用して、OPM、EPM、または ILE プログラムを呼び出すことができ、動的プログラム呼び出しは、プログラム・オブジェクト (*PGM) に対する呼び出しです。OPM および EPM プログラムとは異なり、ILE プログラムは、動的プログラム呼び出しの使用に制限されていません。ILE プログラムでは、他のプロシージャー呼び出するために、静的プロシージャー呼び出しまたはプロシージャー・ポインター呼び出しも使用することができます。

このトピックでは以下について説明します。
- 言語間のプロシージャー呼び出し
- プログラム (*PGM) を呼び出すための ILE 規則
- ILE C/C++ からのプログラムの呼び出し
- 制御言語プログラムから ILE C++ プログラムへの引数の受け渡し
- ILE プログラムからの ILE C プロシージャーへのアクセス
- ILE C++ 動的プログラム呼び出しでのリンクスケップ指定の使用

注: 用語はパラメーター と 引数 は、同じ意味で使用しています。

### 言語間のプロシージャー呼び出し

ILE C では、異なる ILE 高水準言語 (HLL) で作成されたプロシージャー間で引数を渡すことができます。

呼び出し側関数では、引数のサイズおよびタイプが、呼び出し先関数によって期待されるものになるようにする必要があります。

注: 言語間の静的プロシージャー呼び出しでは、以下の場合、操作記述子を使用して、文字ストリングの表現の違いを解決できます。
- 静的呼び出しが、C または C++ 以外の言語で作成されたプロシージャーに対するものである。
- このデータ・タイプの値が引数として渡される。

ILE C では、ILE COBOL や ILE RPG などの言語のバイナド機械プロシージャーへの呼び出しを単純化するために、`pragma argument` ディレクティブが用意されています。`pragma argument` ディレクティブにより、標準 C メカニズム以外のメカニズムで引数を渡すことができます。

デフォルトでは、ILE C プロシージャーは、引数を値によって渡し、受け取ります。これは、データ・オブジェクトへのポインターや引数リストに直接入れられることを意味します。値によって引数を渡すことで、整数および浮動小数点の値を拡張できます。

ILE RPG は、引数を参照によって渡し、受け取ります。これは、データ・オブジェクトへのポインターが引数リストに入れて入れられることを意味します。呼び出し先プロシージャーによって行われる引数の変更は、呼び出し元プロシージャーに反映されます。また、ILE RPG は、値によって引数を渡すことができます。

ILE COBOL は、引数を参照による方法と内容による方法のどちらでも渡すことができます。後者は、データ・オブジェクトの値が一時ロケーションにコピーされることを意味します。コピーのアドレス（ポインター）が、引数リストに配置されます。また、ILE COBOL は、値によって引数を渡すことができます。

注: EPM C および Pascal のプロシージャーまたは関数は、ILE C プロシージャーを呼び出すことができません。OPM プログラムは、ILE プロシージャー (ILE C プロシージャーを含む) を呼び出すことができません。

8-77 ページの表 8-5 に、プロシージャー呼び出しでメソッドを渡すデフォルト引数を示します。

8-76 IBM i: ILE C/C++ プログラマーの手引き
### 表8-5. ILE プロシージャーの引数の受け渡し

<table>
<thead>
<tr>
<th>ILE HLL</th>
<th>引数の受け渡し</th>
<th>引数の受け取り</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILE C のデフォルト</td>
<td>値による</td>
<td>値による</td>
</tr>
<tr>
<td>#pragma argument OS を使用した ILE C</td>
<td>参照による</td>
<td>参照による</td>
</tr>
<tr>
<td>#pragma linkage OS ディレクティブを使用した ILE C</td>
<td>外部プログラムの呼び出し時に OS リンケージを使用</td>
<td>参照による</td>
</tr>
<tr>
<td>ILE COBOL のデフォルト</td>
<td>参照によって</td>
<td>参照による</td>
</tr>
<tr>
<td>ILE CL</td>
<td>参照による</td>
<td>参照による</td>
</tr>
<tr>
<td>ILE RPG のデフォルト</td>
<td>参照による</td>
<td>参照による</td>
</tr>
</tbody>
</table>

### 任意のプログラム呼び出し (*PGM) 用の ILE 規則

プログラムを呼び出す ILE C/C++ プログラム (*PGM) がある場合、ILE C/C++ ソース内の ILE C/C++ 用 OS 呼び出し規則を使用して、PGMNAME がバインド済み ILE プロシージャーでなく外部プログラムであることをコンパイラーに指示します。

注: ILE C 用の OS 呼び出し規則は #pragma linkage (PGMNAME, OS) ディレクティブです。ILE C++ 用の OS 呼び出し規則は extern "OS" です。

このセクションでは、任意の外部プログラムを呼び出す ILE C/C++ プログラム用の動的プログラム呼び出し規則を説明する例を紹介します。[表8-6]に示すように、ILE C は、EPM 入り口点を除くすべての外部プログラムを呼び出す際、同じ規則を使用します。

### 表8-6. 動的プログラム呼び出し規則

<table>
<thead>
<tr>
<th>处置</th>
<th>プログラム呼び出し規則</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILE 呼び出し *PGM (*PGM は以下のとおり)</td>
<td>#pragma linkage (PGMNAME, OS)</td>
</tr>
<tr>
<td>• ILE C</td>
<td>以下に例を示します。</td>
</tr>
<tr>
<td>• OPM COBOL</td>
<td>void PGMNAME(void);</td>
</tr>
<tr>
<td>• OPM RPG</td>
<td>#pragma linkage (PGMNAME, OS)</td>
</tr>
<tr>
<td>• OPM CL</td>
<td>/* Other code */</td>
</tr>
<tr>
<td>• OPM BASIC</td>
<td>/* Dynamic call to program PGMNAME */</td>
</tr>
<tr>
<td>• OPM PL/I</td>
<td>PGMNAME();</td>
</tr>
<tr>
<td>• EPM C</td>
<td></td>
</tr>
<tr>
<td>• EPM PASCAL</td>
<td></td>
</tr>
<tr>
<td>• EPM FORTRAN;</td>
<td></td>
</tr>
<tr>
<td>• ILE COBOL</td>
<td></td>
</tr>
<tr>
<td>• ILE RPG</td>
<td></td>
</tr>
<tr>
<td>• ILE CL</td>
<td></td>
</tr>
<tr>
<td>• C++</td>
<td></td>
</tr>
</tbody>
</table>
表8-6 動的プログラム呼び出し規則 (続き)

<table>
<thead>
<tr>
<th>処置</th>
<th>プログラム呼び出し規則</th>
</tr>
</thead>
</table>
| EPM 入口点を呼び出す ILE C | `#pragma linkage (QPXXCALL, OS)` 例えば、  

```c
#include <xxenv.h>
/* The xxenv.h header file holds */
/* the prototype for QPXXCALL */
/* The #pragma linkage (QPXXCALL, OS) */
/* is in this header file. */
/* Other code. */
/* Dynamic call to program QPXXCALL. */
/* Dynamic call to EPM entry point using QPXXCALL: */
/* the name of the entry point is entname, envid */
/* names the user-controlled environment, the */
/* program and library name is given by envpgm, */
/* parm1 and parm2 are arguments passed to entname. */
QPXXCALL(entname, envid, &envpgm, parm1, parm1);
```

再帰的呼び出しと非再帰的呼び出しの混合

1つの言語で再帰的呼び出しを使用して、別の言語で非再帰的呼び出しを使用するときには、十分に注意する必要があります。以下に例を示します。

- 活動中の ILE C++ プロシージャー (すなわち呼び出しスタック上にあるもの) は、再帰的に (すなわち制御権をその呼び出し元に戻す前に) 呼び出すことができます。
- ILE COBOL プロシージャーは、非再帰的呼び出して呼び出す必要があります。言い換えれば、呼び出しスタックにある ILE COBOL プロシージャーは、制御権をその呼び出し元に戻すか、呼び出しスタックから除かれられるまで呼び出することができます。

既に活動中の別の ILE COBOL プロシージャーを呼び出す可能性のある ILE COBOL プロシージャーの呼び出しに、ILE C++ プロシージャーは使用しないでください。

図8-68 は、こうした呼び出しが機能しないことを示しています。プロシージャー A が ILE C++ プロシージャーであり、プロシージャー B と C が ILE COBOL プロシージャーであり、これらのプロシージャーが同じプログラム内にあると想定してください。プロシージャー A がプロシージャー B を呼び出す場合、プロシージャー B はプロシージャー A にもプロシージャー B も呼び出すことができます。プロシージャー B がプロシージャー A に制御権を戻し、しかるプロシージャー A がその後でプロシージャー C を呼び出した場合、プロシージャー C はプロシージャー B を呼び出すことができますが、プロシージャー A も C も呼び出すことができません。

![呼び出しスタック](_calls.png)

図8-68. ILE C++ プロシージャーは活動中の ILE COBOL プロシージャーを呼び出すことができない

同様に既に呼び出しスタックにある OPM COBOL プログラムを呼び出すことはできません。
ILE プログラムから非 EPM プログラムへの引数の引き渡し

ILE エントリー・ポインタ以外のプログラムに引数を引き渡す際には、以下の規則に従います。

- ILE C/C++ プログラムが呼び出すプログラム名は、大文字でなければなりません。 #pragma map ディレクティブを使用すると、10 文字を超える内部 ID を、プログラム内の IBM i (10 文字以下) に拡張するオブジェクト名にマップすることができます。
- プログラム呼び出しの戻りコードは、整数を戻すようにプログラムを宣言することで検索できます。以下に例を示します。

```c
int PGMNAME(void);
#pragma linkage(PGMNAME, OS)
```

注：ILE ソース内で宣言された関数は、int か void のいずれかを戻す必要があります。その他の型は許可されません。

図 8-69 戻された関数結果を検索するための #pragma linkage(PGMNAME, OS) ディレクティブの使用例

呼び出しで戻される値は、プログラム呼び出しの戻りコードです。呼び出し対象のプログラムが ILE プログラムである場合、この戻りコードは、ヘッダー・ファイル <milib.h> に定義された
_LANGUAGE_RETURN_CODE マクロを使用してもアクセスすることができます。呼び出し対象のプログラムが EPM または OPM プログラムである場合、ジョブ属性検索 (RTVJOBA) コマンドを使用してこの戻りコードにアクセスできます。

- #pragma linkage (PGMNAME, OS) ディレクティブを使用する場合、すべての引数（ポインタと集約を除く）は、コンパイラによって一時変数にコピーされます。一時変数を指すポインタは、呼び出されたプログラムに渡されます。非ポインタ引数は、値参照によって渡されます。
- 値参照（時に値によって間接的に参照されます）は、以下のようなパラメーター引き渡しメカニズムのことをです。- 非ポインタ値が一時変数にコピーされ、この一時変数のアドレスが渡されます。呼び出されたプログラムで変数も行われた変更は、呼び出し側プログラムに反映されません。
- 引数を参照で渡す場合、(&) 演算子のアドレスを使用する必要があります。

EPM エントリー・ポインタに引数を渡す際には、以下の規則に従います。

- EPM デフォルト・エントリー・ポインタを呼び出す ILE C プログラムがある場合、ILE C ソース内で
  #pragma linkage (PGMNAME, OS) ディレクティブを使用して、PGMNAME が外部プログラムであり、バインドされた ILE プロシージャーでないことをコンパイラに通知します。
- EPM のデフォルト以外のエントリー・ポインタを呼び出す ILE C プログラムがある場合、EPM API
  QPXXCALL を使用する必要があります。QPXXCALL は、EPM デフォルト・エントリー・ポインタを
  呼び出す場合にも使用できます。QPXXCALL は OPM プログラムであるため、ILE C ソース中に
  #pragma linkage (QPXXCALL, OS) ディレクティブを使用する必要があります。

ILE プログラムから EPM プログラムへの引数の引き渡し

一般的に、ILE プログラムは、EPM プログラムから関数チェックを受け取りません。ILE プログラムは、raise() 関数からの明示的シグナルをモニターしないからです。EPM 環境では、raise() 関数によって生成される *ESCAPE メッセージの結果として診断メッセージが生成されます。
ILE プログラムが (*ESCAPE メッセージの結果として) 暗黙的にシグナルを出す EPM C プログラムを呼び出すときには、ILE プログラムは、*ESCAPE メッセージによって出された暗黙シグナルをモニターして処理できます。

**C++ 動的プログラム呼び出しにおけるリンクージ指定の使用**

C++ プログラムから OPM、ILE、または EPM プログラムを呼び出すことができます。OPM、ILE、または EPM プログラムからもまた、C++ プログラムを呼び出すことができます。

---

**C++**

C++ が提供するリンクージ指定により、動的プログラム呼び出しおよびそれらの間のデータ共有が可能になります。構文図および追加情報については、「ILE C/C++ 解説書」を参照してください。

有効なストリング・リテラル: “string-literal” は、特定の関数に関連したリンクージを指定するために使用されます。リンクージ指定を使用するストリング・リテラルには、大小文字の区別はありません。プログラムを呼び出すためのリンクージ指定に有効なストリング・リテラルは以下のとおりです。

"OS"  OS リンクージ呼び出し

"OS widen"

拡大されたパラメーターを使用しない OS リンクージ呼び出し。詳細については 8-82 ページの「関数の外部 (OS) リンクージの指定」を参照してください。

リンクージ指定:

---

**C++**

C++ プログラムで ILE、OPM、または EPM プログラム (*PGM) を呼び出した場合、C++ ソース内で extern "OS" リンクージ指定を使用し、呼び出されるプログラムがバインドされた ILE プロシージャーではなく、外部プログラムであることをコンバイラーに指示します。例えば、C++ プログラムで OPM COBOL プログラム (*PGM) を呼び出した場合、C++ ソース内のこの extern "OS" リンクージ指定は、COBOL_PGM がバインドされた ILE プロシージャーではなく、外部プログラムであることをコンバイラーに指示します。

extern "OS" void COBOL_PGM(void);

ILE、OPM、または EPM プログラムで C++ プログラムを呼び出した場合、ILE、OPM、または EPM 言語固有の CALL ステートメントを使用します。

**ILE C/C++ からの任意の ILE プログラムの呼び出し**

**ILE C++ から別の高水準言語へのパラメーターの受け渡し**

プログラム間（またはプロシージャー間）でデータを共有するには、両者のプログラムが使用できるパラメーターを呼び出し先のプログラムまたはプロシージャーに受け渡す必要があります。C++ では、リンクージ指定を使用して、外部呼び出しで使用するパラメーター受け渡し規則をコンバイラーに指示します。

---

**C++**

C++ から別の高水準言語 (HLL) にパラメーターを受け渡す場合は、以下を考慮してください。

・ HLL のパラメーター受け渡しスタイル

各 HLL ごとに、独自のパラメーター受け渡し方法があります。パラメーターは、以下のものとして受け渡すことができます。
- 値へのポインター
- 値のコピーへのポインター
- 値自体

C++ は、3 つのすべての方法でパラメーターを受け渡します。これらのスタイルについて詳しくは、8-85 ページの『デフォルト・パラメーター受け渡しスタイルの使用』を参照してください。

- 言語間のデータの互換性

HLL ごとに、サポートされるデータ表現方法が異なります。呼び出し元と呼び出し先のプログラムまたはプロシージャー共通のデータ・タイプを持つパラメーターのみを受け渡してください。プログラムに受け渡されるデータの正確な形式が分からない場合は、受け渡されるパラメーター形式に関する追加情報を提供するために操作記述子を受け渡すことを、プロシージャーのユーザーに示すことができます。詳しくは、8-86 ページの『操作記述子を使用した不明データ・タイプのパラメーターの受け渡し』を参照してください。

異なるリンケージ指定の使用：表 8-7 は、パラメーター受け渡しで使用するリンケージ・タイプによる影響を示しています。

### 表 8-7: 各種リンケージ指定による影響

<table>
<thead>
<tr>
<th>リンケージ</th>
<th>名前のマングル</th>
<th>パラメーターの受け渡し</th>
<th>パラメーターの拡張</th>
<th>コメント</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>C++</em></td>
<td>いいえ</td>
<td>C++</td>
<td>いいえ</td>
<td>これはデフォルトです。</td>
</tr>
<tr>
<td><em>C</em></td>
<td>いいえ</td>
<td>C++</td>
<td>いいえ</td>
<td>ILE C で作成された関数 (プロシージャー) の呼び出しに使用されます。</td>
</tr>
<tr>
<td><em>C nowiden</em></td>
<td>いいえ</td>
<td>C++</td>
<td>いいえ</td>
<td>ILE C で作成された関数 (プロシージャー) の呼び出しに使用されます。</td>
</tr>
<tr>
<td><em>OS</em></td>
<td>いいえ</td>
<td>OS</td>
<td>いいえ</td>
<td>OPM/EPM/ILE 言語で作成された外部プログラムの呼び出しに使用されます。</td>
</tr>
<tr>
<td><em>OS nowiden</em></td>
<td>いいえ</td>
<td>OS</td>
<td>いいえ</td>
<td>OPM/EPM/ILE 言語で作成された外部プログラムの呼び出しに使用されます。</td>
</tr>
<tr>
<td><em>RPG</em></td>
<td>いいえ</td>
<td>OS</td>
<td>いいえ</td>
<td>ILE RPG で作成されたプロシージャーの呼び出しに使用されます。アドレス・タイプ (ポインターまたは参照) を持つパラメーターは、直接的または値で受け渡されます。その他のパラメーターはすべて、間接的に値で受け渡されます。</td>
</tr>
<tr>
<td><em>COBOL</em></td>
<td>いいえ</td>
<td>OS</td>
<td>いいえ</td>
<td>ILE COBOL で作成されたプロシージャーの呼び出しに使用されます。アドレス・タイプ (ポインターまたは参照) を持つパラメーターは、直接的または値で受け渡されます。その他のパラメーターはすべて、間接的に値で受け渡されます。</td>
</tr>
<tr>
<td><em>CL</em></td>
<td>いいえ</td>
<td>OS</td>
<td>いいえ</td>
<td>ILE CL で作成されたプロシージャーの呼び出しに使用されます。アドレス・タイプ (ポインターまたは参照) を持つパラメーターは、直接的または値で受け渡されます。その他のパラメーターはすべて、間接的に値で受け渡されます。</td>
</tr>
<tr>
<td>リンケージ</td>
<td>名前のマングル</td>
<td>パラメーターの受け渡し</td>
<td>パラメーターの拡張</td>
<td>コメント</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
<td>----------------------</td>
<td>------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>&quot;ILE&quot;</td>
<td>いい</td>
<td>OS</td>
<td>はい</td>
<td>ILE 言語で作成されたプロシージャーの呼び出しに使用されます。 RPG、COBOL、および CL の指定と同じです。関数が作成される際の特定の言語がプログラマーに分からない場合、このリンケージを使用します。 #pragma argument ディレクティブを使用する C コードがあって、このコードを C++に移植する予定の場合は、extern &quot;ILE&quot; のリンケージ指定を使用します。</td>
</tr>
<tr>
<td>&quot;ILE nowiden&quot;</td>
<td>いい</td>
<td>OS</td>
<td>いい</td>
<td>ILE 言語で作成されたプロシージャーの呼び出しに使用されます。</td>
</tr>
<tr>
<td>&quot;VREF&quot;</td>
<td>いい</td>
<td>OS</td>
<td>はい</td>
<td>ILE 言語で作成されたプロシージャーの呼び出しに使用されます。パラメーターは、間接的に値で受け渡されます。</td>
</tr>
<tr>
<td>&quot;VREF nowiden&quot;</td>
<td>いい</td>
<td>OS</td>
<td>いい</td>
<td>ILE 言語で作成されたプロシージャーの呼び出しに使用されます。パラメーターは、間接的に値で受け渡されます。</td>
</tr>
</tbody>
</table>

関数に ILE リンケージがあることの指定: 後にストリング・リテラル "RPG"、"COBOL"、または "CL" が続く extern キーワードは、関数に "ILE" リンケージがあることを指定するために使用されます。こうしたストリング・リテラルは、ILE C での #pragma argument ディレクティブと同じ働きをします。また "VREF" リンケージは、#pragma argument ディレクティブの VREF パラメーターと同じ働きをします。

注: #pragma argument ディレクティブについて詳しくは、「ILE C/C++ コンパイラー参照」を参照してください。

ILE、CL、COBOL、および RPG リンケージの指定: 関数に ILE、CL、COBOL、または RPG リンケージを指定すると、コンパイラーに以下を指示することになります。

- 値または非ポインタ引数として渡された引数が一時変数にコピーされ、この一時変数のアドレスが呼び出されたプロシージャーに渡される。
- ポインタ引数が呼び出されたプロシージャーに直接渡される。
- extern "ILE nowiden" を使用すると、プログラムおよびプロシージャーの間で引き渡しされるパラメーターおよび引数値は拡大化されない。他の ILE リンケージを指定すると、パラメーターが拡大化されます。
- 関数名はマングルされない。

VREF リンケージの指定: VREF リンケージを指定することは、ILE リンケージを指定することと同じです。ポインタ・パラメーターが一時変数に保管され、その一時変数のアドレスが実引数として渡されるという点が異なります。

関数の外部 (OS) リンケージの指定: extern 指定子と、その後に続くストリング・リテラル "OS" またはストリング・リテラル "OS nowiden" を使用して、外部プログラムを宣言します。これにより、通常の関数と同じ方法でこれらのプログラムを呼び出すことができます。

OS リンケージ関数が C++ プログラムから呼び出されると、コンパイラーはコードを生成し、次のタスクを順に実行します。

8-82 IBM i: ILE C/C++ プログラマーの手引き
1. `extern "OS"` が使用された場合は、ファームと戻り値が拡張されます。

   `extern "OS nowiden"` が使用された場合は、プログラム間で受け渡されるパラメータと戻り値は拡張されません。

2. 値で渡されたパラメータは一時変数にコピートされ、その一時変数のアドレスが呼び出し先プログラムに受け渡されます。

   構造に対して一時変数が作成された場合、一時変数の構造情報は、受け渡された `struct` パラメータと同じになります。

3. 参照で受け渡されたパラメータは、引き続き、参照で受け渡されます。

4. 配列およびポインタは、参照で受け渡されます。

5. 受け渡す引数が配列名またはポインタである場合、引数は直接受け渡され、一時変数は作成されません。配列またはポインタで参照されたデータは、呼び出し先プログラムで変更することができます。

6. 関数名はマングルされません。

C++ 動的プログラム呼び出しプログラム名は、大文字でなければなりません。`#pragma map` ディレクトリ使用すると、10文字を超える内部 ID を、プログラム内の IBM i (10文字以下) に拡張するオプジェクト名にマップすることができます。8-47ページの『プログラムおよびプロシージャーの名前変更』を参照してください。

動的プログラム呼び出しの戻りコードは、次のように整数を戻すプログラムを宣言することで取得可能です。

```cpp
extern "OS" int PGMNAME(void);
```

呼び出しで戻される値は、動的プログラム呼び出しの戻りコードです。呼び出されるプログラムが C++ プログラムである場合、この戻りコードには、ヘッダー・ファイル `<milib.h>` で定義された `_LANGUAGE_RETURN_CODE` マクロを使用してアクセス可能です。C++ プログラムは、 `_LANGUAGE_RETURN_CODE` で 4 バイトを戻します。呼び出されるプログラムが EPM または OPM プログラムである場合、この戻りコードには、IBM i ジョブ属性検索 (RTVJOB) コマンドを使用してアクセス可能です。

関数が OS リンケージ関数ポインタから呼び出されると、コンバイラーは、OS リンケージ関数を呼び出す際に同じコード・シーケンスを生成します。

非ポインタの引数が参照で受け渡され、呼び出し先プログラムで変数を行われた変更は、呼び出し元の C++ プログラムでは反映されません。

### 関数に C リンケージがあることの指定:
関数に C リンケージを指定すると、コンバイラーに以下を指示することになります。

- パラメータは、C++ 規則を使用して渡される。
- `extern "C"` で宣言された関数のパラメータは拡大化される。
- 関数名はマングルされない。

後に `string-literal "C"` または `string-literal "C nowiden"` が続く `extern` キーワードは、関数が "C++ リンケージではなく、"C" リンケージを持つように宣言されることを指定するために使用されます。

異なるリンケージ指定の使用 (C++ のみ):

```cpp
```
extern "literal" ステートメントを使用する際には、以下のことに注意してください。
• string-literal パラメーターには、大/小文字の区別があります。例えば、extern "OS NOWIDEN"、extern "OS nowiden"、および extern "os Nowiden" は大小文字が異なっていますが、すべて同様に処理されます。
• 関数の名前は、他の言語の命名規則に従う必要があります。例えば、OS リンケージ指定の場合、プログラム名やすべての IBM i オブジェクトの名前は、大文字にする必要があります。8-47 ベージの「プログラムおよびプロシージャーの名前変更」を参照してください。
• 関数の型定義は、リンケージ情報をせずに宣言することができます。宣言された型定義は、特定のリンケージの関数を宣言する場合に使用できます。型定義宣言は、中括弧 (()) で囲む必要があります。
• 関数は、異なるリンケージがある関数を指すポインターに直接割り当てることができます。この割り当てを可能にするには、型キャストを使用することができます。型キャストを使用すると、余分な制約なしに関数パラメーター不一致の問題を解決できます。8-87 ベージの「リンケージをオーバーライドせずに関数をオーバーライドする型キャスト」を参照してください。
• function ポインターをパラメーターとして使用する関数は、以下のコード・サンプルに示すように、
function ポインターのリンケージに基づいて過負荷になることはありません。
  // Using the typedef declarations above
  void foo (OS);
  void foo (CPP); // undefined behavior, foo already declared
• 非 C++ リンケージ指定で定義される関数は、そのリンケージに適切な規則に従ってパラメーターを受け入れます。以下のコード・サンプルに示すように、パラメーターを拡大する必要はありません。
  extern "C" void foo (char); // chars are widened in C
  // In another compilation unit we then have
  extern "C" void foo (char c) // this parameter is correctly widened
  {
    // implementation of foo (char);
  }
• extern "OS"、extern "OS nowiden"、または extern "builtin" リンケージのいずれかを持つ関数を定義しようとすると、以下のコード・サンプルに示すように、未定義の振る舞いが発生します。
  extern "OS" void FOOPGM (char); // declaration: OK
  extern "OS" void FOOPGM (char c) // definition: undefined behavior
  {
    // implementation of FOOPGM
  }
• function ポインターの宣言は以下のとおりです。
  extern "OS" { 
    typedef void (*fp) (char);
  }
  fp FOO;
注: 関数 FOO() は、型 fp の function ポインターとなるように宣言されます。
• 8-81 ベージの表 8-7 に示したすべてのリンケージ指定の規則の拡大は以下のとおりです。
  - int より小さいデータ・タイプは int に拡大されます。
  - float は double に拡大されます。
  - Address および Data ポインターは拡大されません。
  - struct は、引き渡される struct パラメーターと同じ構造および情報を持ちます。
• データ・オブジェクトは、extern リンケージ宣言内部で宣言できます。
extern "OS" {
    int a1;
}
extern "OS" int a2;

注: 変数 a1 は定義されていますが、変数 a2 は宣言されているだけです。

- 関数 F001()、F002()、および F003() は、すべて OS リンケージ関数として宣言されます。関数
  F001() および F002() は、宣言-リストの構文を使用して宣言されます。F003() は、単純宣言を使用して
  宣言されます。

extern "OS" {
    void FOO1 (char, char *);
    void FOO2 (int, int *);
}
extern "OS" int FOO3 (double, double *);

C++ リンケージ指定では、関数 ID はマングルされます。他のすべてのリンク指定では、すべての
関数 ID は、#pragma map ディレクティブで変更されない限り、エクスポートされた名前と同じです。

注: 構文および説明については、「ILE C/C++ コンパイラ参照」を参照してください。

デフォルト・パラメーター受け渡しスタイルの使用: ILE C/C++ と他の ILE 言語（特に、ILE C、ILE
RPG、または ILE COBOL）間でパラメーターを渡すには、データを参照で受け入れるように他方のプロシ
ージャーをセットアップする必要があります。

ILE C++ は、ILE HLL プログラムまたはプロシージャーを呼び出す場合と同じ呼び出しメカニズム
(extern リンケージ指定) を使用します。

ILE C++ は、以下の 3 つの受け渡し方法を使用して、パラメーターを渡し、受け取ります。

- 値によって、直接に
  そのデータ・オブジェクトの値が引数リストに直接入れられます。

- 値によって、間接に
  データ・オブジェクトの値は、一時的な場所にコピーされます。コピーのアドレス (ポインター)
  が、引数リストに配置されます。

- 参照による
  データ・オブジェクトへのポインターが引数リストに入れられます。呼び出し元プロシージャーに
  よって行われる引数の変更は、呼び出し元プロシージャーに反映されます。

他の ILE 言語では、異なるデータの受け渡し方法を使用していることがあります。表 8-8 を参照してください。

<table>
<thead>
<tr>
<th>ILE HLL</th>
<th>引数の受け渡し</th>
<th>引数の受け取り</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILE C</td>
<td>値によって直接に、または参照による</td>
<td>値によって直接に、または参照による</td>
</tr>
<tr>
<td>ILE C++</td>
<td>値によって直接に、または値によって間接的に、または参照による</td>
<td>値によって直接に、または参照による</td>
</tr>
<tr>
<td>ILE COBOL</td>
<td>参照による、または値によって間接的に</td>
<td>参照による、または値によって間接的に</td>
</tr>
<tr>
<td>ILE RPG</td>
<td>参照による</td>
<td>参照による</td>
</tr>
<tr>
<td>ILE CL</td>
<td>参照による</td>
<td>参照による</td>
</tr>
</tbody>
</table>

8-86 ページの表 8-9 に、ILE プロシージャーの一般的なパラメーター受け渡し方法を示します。
表8-9. ILE プロシージャーのデフォルト引数受け渡しスタイル

<table>
<thead>
<tr>
<th>ILE HLL</th>
<th>引数の受け渡し</th>
<th>引数の受け取り</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILE C</td>
<td>値によって、直接に</td>
<td>値によって、直接に</td>
</tr>
<tr>
<td>ILE C++</td>
<td>値によって直接に、または値によって間接的に</td>
<td>値によって直接に、または参照による</td>
</tr>
<tr>
<td>ILE COBOL</td>
<td>参照による、または値によって間接的に</td>
<td>参照による</td>
</tr>
<tr>
<td>ILE RPG</td>
<td>参照による</td>
<td>参照による</td>
</tr>
<tr>
<td>ILE CL</td>
<td>参照による</td>
<td>参照による</td>
</tr>
</tbody>
</table>

操作記述子を使用した不明データ・タイプのパラメーターの受け渡し： 呼び出し元プロシージャーがデータ・タイプを正確に認識していない場合でも、プロシージャーにパラメーターを渡すために、操作記述子を使用できます。操作記述子では、引数の形式について呼び出し元プロシージャーに記述情報を提供します。この情報によって、プロシージャーは受け渡されたパラメーターを正しく解釈することができます。操作記述子は、呼び出し元プロシージャーによって予期される場合だけ使用してください。

注：操作記述子について詳しくは、以下を参照してください。
・ ILE 概念
・ ILE C/C++ コンパイラー参照

C++ コンパイラーでは、ヌル終了ストリングを記述するための操作記述子をサポートします。 C++ での文字ストリングは、char string_name[n]、char * string_name、または string-literal で定義されます。

C++ では、ストリングは最初のヌル文字で終了され、そのヌル文字を含む連続した文字列として定義されます。別の言語では、ストリングは長さ指定字および文字列から構成されるものとして定義される場合があります。C++ 要素から別の言語で作成された数にストリングを受け渡す場合、操作記述子を引数と共に渡して、呼び出し先関数で、渡されるストリングの長さと型を判別できるようにするのが可能です。

操作記述子を使用するには、ソースで #pragma descriptor ディレクティブを指定して、引数に操作記述子がある関数を識別します。操作記述子は呼び出し元プロシージャーによって作成され、呼び出し先プロシージャーに渡されます。構文については、「ILE C/C++ コンパイラー参照」を参照してください。

次の例で、ILE C/C++ での操作記述子の使用法が示されています。示されている内容は、以下のとおりです。
・ ヘッダー・ファイル oper_desc.h で関数の #pragma descriptor ディレクティブを指定した func1() の #pragma descriptor。8-101ページの図8-85を参照。
・ func1() を呼び出す ILE C プログラム。8-102ページの図8-86を参照してください。
・ 操作記述子から情報を取得するための ILE API を含む func1() の ILE C ソース・コード。8-70ページの図8-70を参照してください。

例：操作記述子による関数の呼び出し：以下の図は、func1() を呼び出す ILE C プログラムを示しています。関数 func1() が呼び出されると、コンパイラーは、呼び出しに指定された 3 つの引数について操作記述子を生成します。
#include "oper_desc.h"
...
main()
{
    char a[5] = {'s', 't', 'u', 'v', '\0'};
    char *c;
    c = "EFGH";
    ...
    func1(a, "ABCD", c);
}

図8-70. 操作記述子による関数の呼び出しのための ILE C ソース

リンケージをオーバーライドせずに関数をオーバーライドする型キャスト
extern "OS" をオーバーライドせずに関数をオーバーライドするには、図8-71 に示したように型キャストを使用します。

extern "ILE"
{
    typedef void (*ILE)();
}
extern "C++"
{
    typedef void (*CPP)();
}
ILE pILE;
CPP pCPP = (CPP) pILE;

図8-71. リンケージをオーバーライドせずに関数をオーバーライドする型キャスト

制御言語プログラムから ILE C++ プログラムへの引数の受け渡し
コマンド行の CL 呼び出しが ILE C++ プログラムへ引数を渡す方法について、表8-10 で説明します。

表8-10. コマンド行の CL 呼び出しが ILE C++ プログラムへ渡される引数

<table>
<thead>
<tr>
<th>コマンド行の引数</th>
<th>Argv 配列</th>
<th>ILE C++ 引数</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv[0]</td>
<td>&quot;LIB/PGMNAME&quot;</td>
<td></td>
</tr>
<tr>
<td>argv[1..255]</td>
<td>通常のパラメーター</td>
<td></td>
</tr>
<tr>
<td>'123.4'</td>
<td>argv[1]</td>
<td>&quot;123.4&quot;</td>
</tr>
<tr>
<td>123.4</td>
<td>argv[2]</td>
<td>_D(&quot;00000123.40000&quot;)</td>
</tr>
<tr>
<td>'Hi'</td>
<td>argv[3]</td>
<td>&quot;Hi&quot;</td>
</tr>
<tr>
<td>Lo</td>
<td>argv[4]</td>
<td>&quot;Lo&quot;</td>
</tr>
<tr>
<td>'1'</td>
<td>argv[5]</td>
<td>&quot;1&quot;</td>
</tr>
</tbody>
</table>

CL 文字配列は、別のプログラムに受け渡されたとき、ヌル終了ではありません。
このような引数を制御言語プログラムから受け取る C++ プログラムでは、ストリングがヌル終了であることは見込めません。 QCAPEXC API を使用すれば、すべての引数をヌル終了にすることができます。

**ILE C++ プログラムへの CL 定数の引き渡し方法**
8-88ページの表8-11 は、コンパイル済みの制御言語プログラムから ILE C++ プログラムにどのように CL 定数が渡されるかを示しています。
表8.11 コンパイル済みの制御言語プログラムから ILE C++ プログラムに渡される CL 定数

<table>
<thead>
<tr>
<th>制御言語プログラム引数のコンパイル</th>
<th>Argv 配列</th>
<th>ILE C++ 引数</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>argv[0]</code></td>
<td>&quot;LIB/PGMNAME&quot;</td>
<td></td>
</tr>
<tr>
<td><code>argv[1..255]</code></td>
<td>通常のパラメーター</td>
<td></td>
</tr>
<tr>
<td><code>'123.4'</code></td>
<td><code>argv[1]</code></td>
<td>&quot;123.4&quot;</td>
</tr>
<tr>
<td><code>123.4</code></td>
<td><code>argv[2]</code></td>
<td>D(&quot;000000123.40000&quot;)</td>
</tr>
<tr>
<td><code>'Hi'</code></td>
<td><code>argv[3]</code></td>
<td>&quot;Hi&quot;</td>
</tr>
<tr>
<td><code>Lo</code></td>
<td><code>argv[4]</code></td>
<td>&quot;Lo&quot;</td>
</tr>
<tr>
<td><code>'1'</code></td>
<td><code>argv[5]</code></td>
<td>&quot;1&quot;</td>
</tr>
</tbody>
</table>

コマンド処理プログラム (CPP) は、[表8-11]に定義されているように CL 定数を渡します。コマンド作成 (CRTCMD) コマンドを使用して独自の CL コマンドを作成し、ILE C++ プログラムをコマンド処理プログラムとして定義できます。

### ILE C++ プログラムへの CL 変数の引き渡し方法

表8-12は、コンパイル済みの制御言語プログラムから ILE C++ プログラムにどのように CL 変数が渡されるかを示しています。すべての引数は、参照によって CL から C++ へ渡されます。

表8-12 コンパイル済みの制御言語プログラムから ILE C++ プログラムに渡される CL 変数

<table>
<thead>
<tr>
<th>CL 変数</th>
<th>C++ 引数</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL VAR(&amp;v) TYPE(*CHAR) LEN(10) VALUE('123.4')</td>
<td>&quot;123.4&quot;</td>
</tr>
<tr>
<td>DCL VAR(&amp;d) TYPE(*DEC) LEN(10 1) VALUE(123.4)</td>
<td>D(&quot;000000123.40000&quot;)</td>
</tr>
<tr>
<td>DCL VAR(&amp;h) TYPE(*CHAR) LEN(10) VALUE('Hi')</td>
<td>&quot;Hi&quot;</td>
</tr>
<tr>
<td>DCL VAR(&amp;i) TYPE(*LGL) LEN(1) VALUE('1')</td>
<td>&quot;1&quot;</td>
</tr>
</tbody>
</table>

CL 変数と数値リテラルは、ヌル終了ストリングを使用して ILE C++ プログラムに渡されません。文字リテラルおよび論理リテラルは、ヌル終了ストリングとして渡されますが、ブランクは埋め込まれません。パック 10 進数などの数値リテラルは 15.5 (8 バイト) として渡されます。

### CL の例: 複数言語 ILE アプリケーション

このプログラムでは、複数の ILE プログラミング言語を使用するプログラムを作成する際の標準的なステップを示します。

プログラムの説明: このプログラムは、8-119 ページの『ILE-OPM CL の例: ILE C++ プログラムからの OPM、COBOL、RPG プログラムへの呼び出し』で説明している小規模トランザクション処理プログラムの ILE バージョンです。

プログラム構造: プログラムは、以下のコンポーネントから構成されます。

- CL コマンド T2123CM3。ユーザー入力を受け取り、ILE 制御言語プログラムに受け渡します。
- ILE 制御言語プログラム T2123CL3。入力を処理し、ILE プログラムに受け渡します。
- ILE プログラム T2123ICB。C++ モジュール T2123ICB の main() 関数は、ILE COBOL モジュール T2123CB2 のプロシージャ CalcAndFormat を呼び出します。

8-88 IBM i: ILE C/C++ プログラマーの手引き
サービス・プログラム T2123SP3。C++ ソース・ファイル t2123icc.cpp から作成され、変数 TAXRATE をエクスポートします。

サービス・プログラム T2123SP4。ILE RPG モジュール・オブジェクト T2123RP2 から作成され、すべてのトランザクションの監査証跡をファイルに書き込みます。

外部記述ファイル T2123D02。監査証跡データを受け取ります。

図 8-72 は、ILE の構造を示しています。

図 8-72. ILE の構造

プログラムの活動化: ACTGRP パラメーター ACTGRP(*NEW) の場合、プログラム T2123CL3 および T2123ICB は、デフォルトで CRTPGM を使用して作成されます。制御言語プログラムが ILE C++ プログラムを呼び出すと、新規活動化グループが開始されます。
図 8-73. 基本オブジェクト構造

ACTGRP パラメーター ACTGRP(*CALLER) の場合、サービス・プログラムはデフォルトで CTRSRVPGM を使用して作成されます。それらは呼び出されると、呼び出し側プログラムの活動化グループ内で活動化されます。

図 8-73 は、この例で使用されている基本オブジェクト構造を示しています。

アプリケーション・モジュールおよびファイル: このプログラムには、外部記述ファイル、制御言語プログラム、コマンド・プロンプト、2 つの C++ ソース・ファイル、および ILE COBOL ソース・ファイルと ILE RPG ソース・ファイルが含まれています。

C++ プログラム T2123CB.CPP

ILE C++ プログラム T2123CB のソースは、8-122 ページの『C++ ソース・ファイル T2123IC5』に示したソースとほぼ同じです。言語間呼び出しに使用されるリンケージ指定に相違があります。8-91 ページの『C++ ソース・ファイル T2123ICB.CPP』を参照してください。

C++ ソース・ファイル T2123ICC

ILE C++ モジュール T2123ICC のソースには、変数 TAXRATE が ILE COBOL および ILE RPG プロシージャーによって使用されるように、このモジュールからエクスポートされることが示されています。8-93 ページの図 8-74 を参照してください。

注: TAXRATE の言語選択は C++ です。弱定義 (COBOL の EXTERNAL) は、サービス・プログラムから C や C++ などの強定義言語にエクスポートできませんが、C または C++ は COBOL にエクスポートできるからです。
制御言語プログラム T2123CL3
制御言語プログラム T2123CL3 は、CL 変数 item_name, price, quantity, および user_id を、参照によって ILE C++ プログラム T2123IC5 に渡します。

CL コマンド・プロンプト T2123CM3
C++ プログラム T2123ICB によって使用される品名、価格、および数量の入力のためのプロンプトをユーザーに出すために、CL コマンド・プロンプト T2123CM3 を使用します。

ILE COBOL モジュール T2123CB2
T2123CB2 内の ILE COBOL プロシージャーは、変数 price, quantity, および taxrate の値を指すポインタを受け取り、さらに formatted_cost と success_flag を指すポインタを受け取ります。CalcAndFormat() 関数は、合計コストを計算してフォーマットします。税金計算を行うために、パラメーターが ILE C++ プログラムから ILE COBOL プロシージャーに渡されます。8-94 ページの図 8-75 を参照してください。

ILE RPG モジュール T2123RP2
ILE RPG モジュール T2123RP2 には、プログラムの監査証跡を書き込む WriteAuditTrail() 関数が含まれています。8-95 ページの図 8-76 を参照してください。

サービス・プログラム T2123SP3
サービス・プログラム T2123SP3 は、C++ モジュール T2123ICC から作成されます。これは、変数 TAXRATE をエクスポートします。

サービス・プログラム T2123SP4
サービス・プログラム T2123SP4 は、ILE RPG モジュール T2123RP2 から作成されます。これはプロシージャー T2123RP2 をエクスポートします。

外部記述ファイル T2123DD2
ファイル T2123DD2 には、C++ プログラム T2123ICB の監査証跡が含まれます。DDS ソースは、監査ファイルのフィールドを定義します。監査ファイル T2123DD2 の DDS ソースについては、8-120 ページの「外部記述ファイル T2123DD2」を参照してください。

C++ ソース・ファイル T2123ICB.CPP:
// This program demonstrates the interlanguage call capability
// of an ILE C++ program. This program is called by a CL
// program that passes an item name, price, quantity and user ID.
// A COBOL procedure is called to calculate and format total
// cost. An RPG procedure is called to write an audit trail.

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <bcd.h>

// The #pragma map directive maps a function name to the bound
// procedure name so that the purpose of the procedure is clear.
// Tell the compiler that there are bound procedure calls and
// arguments are to be passed by value-reference.
extern "COBOL" void CalcAndFormat(_DecimalT<10,2>,
    short int, char[],
    char *);

#pragma map(CalcAndFormat,"T2123CB2")
extern "RPG" void WriteAuditTrail(char[],
    char[],
    _DecimalT<10,2>,
    short int, char[]);
#pragma map(WriteAuditTrail,"T2123RP2")

```c
int main(int argc, char *argv[])
{
    // Incoming arguments from a CL program have been verified by
    // the *CMD and null-terminated within the CL program.
    // Incoming arguments are passed by reference from a CL program.
    char *user_id;
    char *item_name;
    short int quantity;
    _DecimalT<10, 2> price;
    char formatted_cost[22];

    // Remove null terminator for RPG program. Item name is null
    // terminated for C++.
    char rpg_item_name[20];
    char null_formatted_cost[22];
    char success_flag = 'N';
    int i;

    //Incoming arguments are all pointers.
    item_name = argv[1];
    price = *((_DecimalT<10, 2> *) argv[2]);
    quantity = *((short *) argv[3]);
    user_id = argv[4];

    // Call the COBOL program to do the calculation, and return a
    // Y/N flag, and a formatted result.
    CalcAndFormat(price, quantity, formatted_cost, &success_flag);
    memcpy(null_formatted_cost,formatted_cost,sizeof(formatted_cost));

    // Null terminate the result.
    formatted_cost[21] = '\0';
    if (success_flag == 'Y')
    {
        for (i=0; i<20; i++)
        {
            // Remove null terminator for the RPG program.
            if (*(item_name+i) == '\0')
            {
                rpg_item_name[i] = ' ';  
            }
            else
            {
                rpg_item_name[i] = *(item_name+i);
            }
        }
    }
    // Call an RPG program to write audit records.
    WriteAuditTrail(user_id, rpg_item_name, price, quantity,
                    formatted_cost);
    cout <<"plus tax =" << quantity << item_name << null_formatted_cost
         <<endl <<endl;
    }
    else
    {
        cout <<"Calculation failed" <<endl;
    }
}
```

8-92 IBM i: ILE C/C++ プログラマーの手引き
C++ ソース・ファイル T2123ICC:

// Export the tax rate data.
#include <bcd.h>
const _DecimalT <2,2> TAXRATE = __D(".15");

图8-74. ILE COBOL および ILE RPG プロシージャーによって使用するための変数をエクスポートする C++ ソース・コード T2123ICC。:

注: 弱定義 (COBOL の EXTERNAL) は、サービス・プログラムから C や C++ などの強定義言語にエクスポートできませんが、C または C++ は COBOL にエクスポートできます。TAXRATE の言語選択は C++ です。

ILE COBOL プログラム T2123CB2:
IDENTIFICATION DIVISION.
PROGRAM-ID. T1520CB2 INITIAL.
******************************************************
* parameters: *
* incoming: PRICE, QUANTITY   *
* returns : TOTAL-COST (PRICE*QUANTITY*1.TAXRATE)   *
* SUCCESS-FLAG.   *
* TAXRATE : An imported value.   *
******************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-I.
OBJECT-COMPUTER. IBM-I.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 WS-TAXRATE PIC S9V99 COMP-3.
     VALUE 1.
  01 TAXRATE EXTERNAL PIC SV99 COMP-3.
LINKAGE SECTION.
  01 LS-PRICE PIC S9(8)V9(2) COMP-3.
  01 LS-QUANTITY PIC S9(4) COMP-4.
  01 LS-TOTAL-COST PIC $$,$$$,$$$,$$$,$$$99 DISPLAY.
  01 LS-OPERATION-SUCCESSFUL PIC X DISPLAY.
PROCEDURE DIVISION USING LS-PRICE
LS-QUANTITY
LS-TOTAL-COST
LS-OPERATION-SUCCESSFUL.

MAINLINE.
  MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
  PERFORM CALCULATE-COST.
  PERFORM FORMAT-COST.
  EXIT PROGRAM.

CALCULATE-COST.
  ADD TAXRATE TO WS-TAXRATE.
  COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY *
                               LS-PRICE *
                               WS-TAXRATE

  ON SIZE ERROR
    MOVE "N" TO LS-OPERATION-SUCCESSFUL
    END-COMPUTE.

FORMAT-COST.
  MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

I E R P G モジュール T 2 1 2 3 R P 2 :
ILE プログラムの呼び出し: T2123ICB は、メインプログラムとみなされます。これは、制御言語プログラム T2123CL3 の呼び出し時に作成される新規活動化グループ内で実行します。

プログラム T2123ICB のデータを入力するには、コマンド T2123CM2 を入力して、F4 (プロンプト) を押します。8-125 ページの『ILE-OPM プログラムの呼び出し』のサンプル・データを入力できます。

出力は、このプログラムの OPM バージョンと同じです。

物理ファイル T2123DD2 には、8-125 ページの『ILE-OPM プログラムの呼び出し』の OPM バージョンに示したものと同じデータが含まれています。

例: ILE C++ プログラムを呼び出すユーザー定義の制御言語プログラム

8-97 ページの図 8-80 は、main から戻り値を取得する方法を示しています。SQUARE という CL コマンドは ILE C++ プログラム SQITF を呼び出します。プログラム SQITF は SQ という別の ILE C++ プログラムを呼び出します。プログラム SQ はプログラム SQITF に値を返します。

CL コマンド・プロンプト SQUARE を使用して、ILE C++ プログラム SQITF のために二乗を求める数値を入力します。

CMD PROMPT('CALCULATE THE SQUARE')
PARM KWD(VALUE) TYPE(+INT4) RSTD(+NO) RANGE(1 + 9999) MIN(1) ALWUNPRT(+YES) PROMPT('Value' 1)

注: ILE C++ プログラムから整数値が戻ると、パフォーマンスに影響する可能性があります。

プログラミング・タスク:
1. 下に示したソースを使用して CL コマンド・プロンプト SQUARE を作成するには、以下を入力します。

CRTCMD CMD(MYLIB/SQUARE) PGM(MYLIB/SQITF) SRCFILE(MYLIB/QCMDSRC)

CMD PROMPT('CALCULATE THE SQUARE')
PARM KWD(VALUE) TYPE(+INT4) RSTD(+NO) RANGE(1 + 9999) MIN(1) ALWUNPRT(+YES) PROMPT('Value' 1)

図 8-77. SQUARE — 入力データを受け取るための CL コマンド・ソース

ILE C プログラム SQITF の値を入力するには、CL コマンド SQUARE を使用します。
2. 下に示したソースを使用してプログラム SQIFT を作成するには、以下を入力します。

```c
CRTBNDC PGM(MYLIB/SQIFT) SRCFILE(MYLIB/QCSRC)

/* This program SQIFT is called by the command SQUARE. This */
/* program then calls another ILE C program SQ to perform     */
/* calculations and return a value.                          */
#include <stdio.h>
#include <decimal.h>
#pragma linkage(SQ, OS)    /* Tell compiler this is external call, */
/* do not pass by value.                                     */
int SQ(int);            /* This program is called by another ILE C program called SQIFT. */
int main(int argc, char *argv[]) {
    int *x;
    int result;
    x = (int *) argv[1];
    result = SQ(*x);
    printf("The SQUARE of %d is %d\n", *x, result);
}
```

図 8.78. SQIFT — 値によって引数を渡すための ILE C ソース

3. 下に示したソースを使用してプログラム SQ を作成するには、以下を入力します。

```c
CRTBNDC PGM(MYLIB/SQ) SRCFILE(MYLIB/QCSRC) OUTPUT(*PRINT)

/* This program is called by another ILE C program called SQIFT. */
/* It performs the square calculations and returns a value to SQIFT. */
#include <stdio.h>
#include <decimal.h>
int main(int argc, char *argv[]) {
    int *vin;
    int vout;
    vin = (int *) argv[1];
    vout = (*vin)*(*vin);
    return(vout);
}
```

図 8.79. SQ — 計算を実行し、値を戻すための ILE C ソース

プログラム SQ は、整数値を計算して、その値を呼び出し側プログラム SQIFT に戻します。

4. プログラム SQIFT のデータを入力するには、以下を入力します。

```
SQUARE
```

次に、F4 (プロンプト) を押します。

5. 10 を入力して、実行キーを押します。出力は以下のようにになります。

```
The SQUARE of 10 is 100
実行キーを押して端末セッションを終了してください。
```

ソース・コード:

8-96 IBM i ILE C/C++ プログラマーの手引き
// このプログラムの機能は、命令 SQUARE によって呼び出されます。この
// プログラムは、再帰的に呼び出される ILE C++ サブルーチン SQ に数を
// 平方計算し、その結果を戻します。

#include <iostream.h>
extern "OS" int SQ(int); // これは外部呼出であることを示す。
                        // したがって、引数は値ではなくポインタを伝えるべき。

int main(int argc, char *argv[])
{
    int *x;
    int result;
    x = (int *) argv[1];
    result = SQ(*x);

    cout << "The SQUARE of " << x << "is " << result << endl;
}

ILE C++ プログラム SQ は、整数値を計算して、その値を呼び出し側プログラム SQITF に戻します。

// このプログラムは、ILE C++ サブルーチン SQITF によって呼び出される。
// これは、呼び出されたプログラム SQ が計算し、戻される結果を戻されます。

int main(int argc, char *argv[])
{
    return (*(int *) argv[1]) * (*(int *) argv[1]);
}

図 8-80. 指定された数の二乗を計算するユーザーディフェンス CL コマンド SQUARE

CL コマンド SQUARE を使用して、計算値を戻す： プログラム SQITF のデータを入力するには、以下のようにします。
1. コマンド SQUARE を入力して、F4 (プロンプト) を押します。
2. 10 を入力して、実行キーを押します。

出力は次のとおりです。

The SQUARE of 10 is 100
実行キーを押して終了セッションを終了してください。

例: ILE C++ プログラムにパラメーターを渡す制御言語プログラム

制御言語プログラム CLPROG1 は、パラメーター v, d, h, i, j を ILE C++ プログラム MYPROG1 に渡します。

注: ILE プログラムの実行時にパラメーターをそれに渡す場合、CL 呼び出し (CALL) コマンドの PARM
オプションを使用します。

パラメーターは、制御言語プログラム CLPROG1 内でヌル終了となっています。パラメーターは、参照によって渡されます。MYPROG1 の入力引数はすべてポインタです。

IBM i 機能を使用した処理 8-97
/* CLPROG1
PGM PARM(&V &D &H &I &J)
  DCL VAR(&V) TYPE(*CHAR) LEN(10)
  DCL VAR(&VOUT) TYPE(*CHAR) LEN(11)
  DCL VAR(&D) TYPE(*DEC) LEN(10 1)
  DCL VAR(&H) TYPE(*CHAR) LEN(10)
  DCL VAR(&HOUT) TYPE(*CHAR) LEN(11)
  DCL VAR(&I) TYPE(*CHAR) LEN(10)
  DCL VAR(&IOUT) TYPE(*CHAR) LEN(11)
  DCL VAR(&J) TYPE(*LGL) LEN(1)
  DCL VAR(&JOUT) TYPE(*LGL) LEN(2)
  DCL VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
  /* ADD NULL TERMINATOR FOR THE C++ PROGRAM */
  CHGVAR VAR(&VOUT) VALUE(&V TCAT &NULL)
  CHGVAR VAR(&HOUT) VALUE(&V TCAT &NULL)
  CHGVAR VAR(&IOUT) VALUE(&V TCAT &NULL)
  CHGVAR VAR(&JOUT) VALUE(&V TCAT &NULL)
  CALL PGM(MYPROG1) PARM(&VOUT &D &HOUT &IOUT &JOUT)
ENDPGM

図8-81. ILE C++ プログラムに引数を渡す制御言語プログラムの例：

制御言語プログラム CLPROG1 は、必要な値を入力するように求めるプロンプトをユーザーに出す CL コマンド・プロンプト MYCMD1 からその入力値を受け取ります。MYCMD1 のソース・コードは以下のとおりです。

CMD PROMPT('ENTER VALUES')
PARM KWD(V) TYPE(*CHAR) LEN(10) +
  PROMPT('1ST VALUE')
PARM KWD(D) TYPE(*DEC) LEN(10 2) +
  PROMPT('2ND VALUE')
PARM KWD(H) TYPE(*CHAR) LEN(10) +
  PROMPT('3RD VALUE')
PARM KWD(I) TYPE(*CHAR) LEN(1) +
  PROMPT('4TH VALUE')
PARM KWD(J) TYPE(*LGL) LEN(10 2) +
  PROMPT('5TH VALUE')

図8-82. 活用 CL コマンド・プロンプトの例：

制御言語プログラム CLPROG1 は、コマンド・プロンプト MYCMD1 からユーザー入力を受け取ると、その入力値を C++ プログラム MYPROG1 に渡します。このプログラムのソース・コードは、myprogl.cpp に含まれています。
// myprog1.cpp

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <bcd.h>

// Arguments are received by reference from CL program CLPROG1
// Incoming arguments are all pointers

int main(int argc, char *argv[])
{
    char *v;
    char *h;
    char *i;
    char *j;
    _DecimalT <10, 1> d;

    v = argv[1];
    d = *((_DecimalT <10,1> *) argv[2]);
    h = argv[3];
    i = argv[4];
    j = argv[5];
    cout << " v= " << v
         << " d= " << d
         << " h= " << h
         << " i= " << i
         << " j= " << j
         << endl;
}

図8-83. 参照によって引数（ポインタ）を受け取る C++ プログラムの例：

制御言語プログラム CLPROG1 以下のパラメーターを C++ プログラム MYPROG1 に渡します。
'123.4', 123.4, 'Hi', LO, and '1'

プログラム MYPROG1 の出力は、以下のようになります。

<table>
<thead>
<tr>
<th>v</th>
<th>123.4</th>
<th>Hi</th>
<th>LO</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>123.4</td>
<td>123.4</td>
<td>HI</td>
<td>LO</td>
</tr>
</tbody>
</table>

実行キーを押して端末セッションを終了してください。

ILE プログラムからの ILE C プロシージャーへのアクセス

ILE C プログラムは動的プログラム呼び出しによって呼び出されますが、活動化された ILE C プログラム内のプロシージャーは、以下のいずれかの手段によってアクセスします。

- 静的プロシージャー呼び出し
- プロシージャー・ポインタ呼び出し

注: 活動化されていない ILE C プログラムは、動的に呼び出す必要があります。

静的プロシージャー呼び出し

静的プロシージャー呼び出しは、以下のいずれかを呼び出すことができます。

- 同じモジュール内のプロシージャー
- 同じ ILE C プログラムまたはサービス・プログラム内の別のモジュールのプロシージャー
- 別の ILE C サービス・プログラム内のプロシージャー

注: ILE の「プロシージャー」という用語は、ILE C の「関数」という用語と同じです。
静的プロシージャー呼び出しはコンパイル時に解決され、バインドされているため、静的プロシージャー呼び出しでは、実行時（特にプログラムを活動化するとき）に使用するシステム資源が、動的プログラム呼び出しより少なくなります。動的プログラム呼び出しの記号は、呼び出しが実行されるたびにアドレスに解決されます。

注：「静的プロシージャー呼び出し」という用語は、静的ストレージ・クラスを指すのではなく、バインド済みモジュールまたはサービス・プログラム内のバインド済みプロシージャー呼び出しを指します。以下の場合は、操作記述子を使用して、文字ストリングの表記における差異を解決することができます。

- 静的呼び出し、C または C++ 以外の言語で作成されたプロシージャーに対するものである
- このデータ・タイプの値が引数として受け渡される

プロシージャー・ポインター呼び出し

プロシージャー・ポインター呼び出しは、プロシージャーを動的に呼び出す方法を提供します。例えば、プロシージャー名またはアドレスからなる配列またはテーブルを操作することによって、1 つのプロシージャー呼び出しを複数のプロシージャーに動的に経路指定することができます。

呼び出されるプロシージャーおよび操作記述子

操作記述子は、呼び出されたプロシージャーが引数の形式（例えば、種々のタイプのストリング）を正確に予期できない場合に、呼び出されたプロシージャーに記述情報を提供します。この追加情報によって、プロシージャーはストリングを適切に解釈することができます。呼び出されたプロシージャー（通常は ILE バインド可能 API）に必要な場合にだけ、操作記述子を使用する必要があります。

例：ILE C からの ILE API の呼び出し：以下の図は、ILE API の呼び出しが含まれる func1() の ILE C ソースコードを示しています。この API は、func1() 内に宣言されたストリング引数の STRING 型、長さ、および最大長を決定するために使用されます。typeCharZ および typeCharV2 の値は、ILE API ヘッダー・ファイル <leod.h> 内にあります。
#pragma descriptor

(figsize)

図8-84. 関数内のストリング引数を決定する ILE C ソース

操作記述子および #pragma descriptor ディレクティブ

ILE C/C++ は、引数に操作記述子がある関数を識別するために #pragma descriptor ディレクティブを提供しています。ILE バインド可能 API の操作記述子情報検索 (CEEDOD) およびストリング引数に関する操作記述情報取得 (CEESGI) を使用して、操作記述子から情報を取得できます。ILE C/C++ は、ストリング引数に対して操作記述子をサポートします。

注: 「静的プロシージャー呼び出し」という用語は、静的ストレージ・クラスを指すのではなく、バインド済みモジュールまたはサービス・プログラム内のバインド済みプロシージャー呼び出しを指します。以下の場合は、文字ストリングにおける表記の相違を解決するために、操作記述子を使用できます。

・ 静的呼び出しが、C または C++ 以外の言語で作成されたプロシージャーに対するものである
・ このデータ・タイプの値が引数として渡される。

例: 操作記述子を使用する必要がある関数の宣言: 以下の図は、一部の引数に操作記述子を必要とする関数を宣言する方法を示しています。

```c
int func (char *, int *, int, char *, ...); /* prototype */
#pragma descriptor (void func ("", void, void, "))
```

図8-85. 操作記述子を必要とする関数を宣言する ILE C ソース

func() という名前の関数が宣言されています。func() の #pragma 記述子は、func() が呼び出されたときには、必ず ILE C コンパイラが func() の最初と 4 番目の引数にストリング操作記述子を生成する必要があることを指定します。
例: 操作記述子の生成: 以下の図は、ヘッダー・ファイル oper_desc.h 内の関数 func1 に #pragma descriptor ディレクティブがあることを示しています。

/* Function prototype in oper_desc.h */
int func1( char a[5], char b[5], char *c );
#pragma descriptor(void func1( "", "", "" ))

図 8-86. 操作記述子を生成する ILE C ソース

func1 という名前の関数が宣言されています。func1 の #pragma descriptor は、ILE C コンパイラーが 3つの引数にストリング操作記述子を生成する必要があることを指定します。

OPM CL の例: ILE C プログラムからの OPM、COBOL、および RPG プログラムの呼び出し

このアプリケーションでは、セッション入力を使用して以下を行います。

- 税の計算
- 出力の形式設定
- 監査ファイルの書き込み

注: この例の ILE バージョンについては、8-110 ページの『ILE CL の例: ILE C プログラムからの OPM、COBOL、および RPG プログラムの呼び出し』を参照してください。

基本プログラム構造に示されているように、このアプリケーションは、名前、価格、数量を入力項目として取り入れる小さなトランザクション処理プログラムです。出力として、アプリケーションは、画面に指定された項目の合計コストを表示し、トランザクションの監査証跡をファイルに書き込みます。
基本プログラム構造:

基本プログラム構造に示したように、この例は、以下のものから構成されます。

- CL コマンド (T1520CM2)。ユーザーの入力を受け入れ、それを制御言語プログラム (T1520CL2) に渡します。
- 制御言語プログラム (T1520CL2)。その入力を処理し、それを ILE C プログラム (T1520IC5) に渡します。
- ILE C (T1520IC5) プログラム。入力を処理する OPM COBOL プログラム (T1520CB1) および監査証跡を外部記述ファイルに書き込む OPM RPG プログラム (T1520RP1) を呼び出します。
- OPM COBOL プログラム (T1520CB1)。計算を実行し、コストをフォーマットします。
- OPM RPG プログラム (T1520RP1)。監査ファイル (T1520DD2) を各トランザクションで更新します。

注: CMD、CL、ILE C、OPM RPG、および OPM COBOL のソースに加えて、出力ファイルの DDS ソースが必要になります。DDS フロースは、監査ファイルのフィールドを定義します。

プログラム・モジュールおよび活動化グループ: [8-104 ページの図 8-88] に示したように、制御言語プログラ
M、COBOL プログラム、および RPG プログラムは、ユーザー・デフォルト設定の活動化グループ内でアクティブ化されています。新規活動化グループは、制御言語プログラムが ILE C プログラムを呼び出すと開始されます。ILE C プログラムは ACTGRP キーワードにデフォルトの *NEW を指定した CRTPGM によって作成されるからです。
注: CRTBND C コマンドに CRTPGM パラメーターがない場合、CRTPGM パラメーターのデフォルトが使用されます。

図 8-88. ILE C でのプログラムの構造

以下のプログラミング・ステップでは、次の方法について説明します。

• ILE C プログラムの監査証跡が入る物理ファイルを作成する。
• ILE C プログラムに品名、価格、数量、およびユーザー ID の各パラメーターを渡す制御言語プログラムを作成する。
• 品名、価格、および数量のデータを入力するための CL コマンド・ブロントを作成する。 OPM CL コマンド・ブロントは、データを制御言語プログラムに渡し、その制御言語プログラムは ILE C プログラムを呼び出します。
• main() 関数を使用して、制御言語プログラムから入力引数を受け取り、その後、金額計算を実行してから、合計コストをフォーマットする OPM COBOL プログラムを呼び出す 1 つのプログラムを作成する。さらにこのプログラムは、監査レコードを書き込む OPM RPG プログラムを呼び出します。
• 税金計算を実行し、合計コストをフォーマットする OPM COBOL プログラムを作成する。
• アプリケーションの監査証跡を書き込む OPM RPG プログラムを作成する。

プログラミング・タスク:
1. 8-105 ページの図 8-89 に示されているソースを使用して、物理ファイル T1520DD2 を作成します。コマンド行で、次のように入力します。
   CRTPF FILE(MYLIB/T1520DD2) SRCFILE(QCPPLE/QADDSSRC) MAXMBRS(+NOMAX)

   このソース・ファイルには、ILE C プログラム T1520IC5 の監査証跡が含まれています。
図8.90 T1520CL2 ー ILE C プログラムに変数を渡すための CL ソース

注:

a. CL 変数 item name、price、quantity、および user ID は、ILE C プログラム T1520IC5 へ参照によって渡されます。
b. データ処理の確認 (RTVJOBA) コマンドは、監査証明のユーザー ID を取得します。
c. 引数は参照によって渡されます。引数は、受け取り側の ILE C プログラムが変更できます。
d. 変数 item name は、制御言語プログラムでnull終了です。
e. CL 変数と数値定数は、null終了ストリングで ILE C プログラムに渡されることはありません。
f. 文字定数と論理リテラルは、null終了ストリングとして渡されますが、プランクで拡張されることはありません。
g. サイコロ 10 進数などの数値定数は、15.5 (8 バイト) として渡されます。
h. 浮動小数点定数は、倍精度浮動小数点数値（例えば、1.2E+15）として渡されます。
i. ILE プログラムの実行時にパラメーターをそれに渡す場合、CL 呼び出し (CALL) コマンドの PARM オプションを使用します。

3. CL コマンド・プロンプト T1520CM2 を作成して、ILE C プログラム T1520IC5 への品目名、価格、および数量を入力します。8-106ページの図8-91に示されているソースを使用します。コマンド行で、次のように入力します。
4.  The program is called by a CL program that passes an item name, price, quantity and user ID. A COBOL program is called to calculate and format the total cost. An RPG program is called to write an audit trail.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>

#pragma map(calc_and_format, "T1520CB1")
#pragma map(write_audit_trail, "T1520RP1")

void calc_and_format(decimal(10,2),
    short int,
    decimal(2,2),
    char[],
    char[]);

void write_audit_trail(char[],
    char[],
    decimal(10,2),
    short int,
    decimal(2,2),
    char[]);

int main(int argc, char *argv[])
{
    /* Incoming arguments from a CL program have been verified by */
    /* the "CMD and null end within the CL program. */
    /* Incoming arguments are passed by reference from a CL program. */
    char    *user_id;
    char    *item_name;
    short int quantity;
    decimal(10,2) price;
    decimal(2,2) taxrate = .15D;
    char    formatted_cost[22];
    /* Remove null end for RPG program. Item name is null ended for C. */
    char    rgp_item_name[20];
    char    null_formatted_cost[22];
    char    success_flag = 'N';
    int    i;

    /* Incoming arguments are all pointers. */
```
item_name = argv[1];
price = *((decimal (10, 2) *) argv[2]);
quantity = *((short *) argv[3]);
user_id = argv[4];

/* Call the COBOL program to do the calculation, and return a */
/* Y/N flag, and a formatted result. */
calc_and_format(price,
    quantity,
    taxrate,
    formatted_cost,
    &success_flag);

memcpy(null_formatted_cost,formatted_cost,sizeof(formatted_cost));

/* Null end the result. */
formatted_cost[21] = '\0';
if (success_flag == 'Y')
{
    for (i=0; i<20; i++)
    {
        /* Remove null end for the RPG program. */
        if (*(item_name+i) == '\0')
        {
            rpg_item_name[i] = '\0';
        }
        else
        {
            rpg_item_name[i] = *(item_name+i);
        }
    }

    /* Call an RPG program to write audit records. */
    write_audit_trail(user_id,
        rpg_item_name,
        price,
        quantity,
        taxrate,
        formatted_cost);

    printf("\n%d %s plus tax = %s\n", quantity,
        item_name,
        null_formatted_cost);
}
else
{
    printf("Calculation failed\n");
}

注:
a. このプログラムの main() 関数は、CL コマンド・プロンプト T1520CM2 によって検査され、制御
言語プログラム T1520CL2 内で処理を終了する、制御言語プログラム T1520CL2 からの入力引数を受
け取ります。すべての入力引数はポインタです。
b. また、main() 関数は、COBOL 名にマップされる calc_and_format() を呼び出します。これは、価
格、数量、税率、フォーマット設定されたコスト、および成功フラグを OS リンケージ規則によっ
て渡します。
c. OPM COBOL プログラムでは、拡張されたパラメーター (ILE C のデフォルト) が期待されるわけ
ではないため、nowiden が #pragma linkage ディレクティブで使用されます。フォーマット設定さ
れたコストおよび成功フラグの値は、プログラム T1520IC5 で更新されます。
d. calc_and_format() が正常に戻った場合、OPM RPG プログラムの write_audit_trail() によってレコードが監査証跡に書き込まれます。このプログラム (T1520IC5) の main() 関数は、RPG プログラム名にマップされる write_audit_trail() を呼び出します。これは、ユーザー ID、品目名、価格、数量、税率、およびフォーマット設定されたコストを OS リンケージ規則によって渡します。

e. デフォルトでは、ILE コンパイラは、#pragma linkage ディレクティブで nowiden パラメーターが指定されていない限り、短整数を整数に変換します。例えば、ILE C プログラムの短整数は整数に変換されてから、OPM RPG プログラムに渡されます。RPG プログラムでは、quantity 変数として 4 パイトの整数が期待されています。

f. OUTPUT(*PRINT) は、コンパイラ・リストが必要であることを指定します。OPTION(*SHOWINC *NOLOGMSG) は、コンパイラ・リストの組み込みファイルを展開し、CHECKOUT オプション・メッセージを記録しないことを指定します。

g. FLAG(30) は、重大度レベル 30 のメッセージがリストで表示されるように指定します。
MSGLMT(10) は、重大度レベル 30 のメッセージが 10 件出た後にコンパイルを停止するように指定します。CHECKOUT(*PARM) は、使用されていない関数名パラメーターのリストを示します。
DBGVIEW(*ALL) は、3 つすべてのビューおよびデバッグ・データでこのプログラムをデバッグするように指定します。

5. [8-109 ページの図 8-92] に示されているソースを使用して、OPM COBOL プログラムを作成します。コマンド行で、次のように入力します。
  CRTCBPDM PGM(MYLIB/T1520CB1) SRCPFLE(QCPLLE/QALBLSRC)
IDENITIFICATION DIVISION.
PROGRAM-ID. T1520CB1.

****************************************************
* parameters: *
* incoming: PRICE, QUANTITY *
* returns: TOTAL-COST (PRICE+QUANTITY+1.TAXRATE) *
* SUCCESS-FLAG. *
****************************************************

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-I.
OBJECT-COMPUTER. IBM-I.
DATA DIVISION.

WORKING-STORAGE SECTION.
01 WS-TAXRATE PIC S9V99 COMP-3.

LINKAGE SECTION.
01 LS-PRICE PIC S9(8)V9(2) COMP-3.
01 LS-QUANTITY PIC S9(4) COMP-4.
01 LS-TAXRATE PIC SV99 COMP-3.
01 LS-TOTAL-COST PIC $$,$$$,$$$,$$$,$$$.99 DISPLAY.
01 LS-OPERATION-SUCCESSFUL PIC X DISPLAY.

PROCEDURE DIVISION USING LS-PRICE 0 LS-QUANTITY LS-TAXRATE LS-TOTAL-COST LS-OPERATION-SUCCESSFUL.

MAINLINE.
MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
PERFORM CALCULATE-COST.
PERFORM FORMAT-COST.
EXIT PROGRAM.

CALCULATE-COST.
MOVE LS-TAXRATE TO WS-TAXRATE.
ADD 1 TO WS-TAXRATE.
COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY * LS-PRICE * WS-TAXRATE ON SIZE ERROR
MOVE "N" TO LS-OPERATION-SUCCESSFUL
END-COMPUTE.

FORMAT-COST.
MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

IBM i機能を使用した処理 8-109
図 8-93. T1520RP1 — 監査証跡書き込むための OPM RPG ソース

注:

a. プログラム T1520RP1 の write_audit_trail() 関数は、プログラムの監査証跡を書き込みます。
b. 「RPG/400® User's Guide」には、OPM RPG プログラムをコンパイルする方法に関する情報が含まれています。

7. プログラム T1520IC5 へのデータを入力します。

a. コマンド行に次のように入力します。
   
   T1520CM2

   次に、F4 (プロンプト) を押します。
b. 以下のデータを T1520CM2 に入力します。

   Hammers
   1.98
   5000
   Nails
   0.25
   2000

出力は以下のように変わります。

| 5000 HAMMERS plus tax = $11,385.00 |
| 2000 NAILS plus tax = $575.00 |

実行キーを押して帳簿セッションを終了してください。

物理ファイル T1520DD2 には、データが以下のように含まれています。

SMITHE HAMMERS 00000000198500015 11,385.00 072893
SMITHE NAILS 00000000252000015 575.00 072893

ILE CL の例: ILE C プログラムからの OPM、COBOL、および RPG プログラムの呼び出し

この例は、小規模トランザクション処理プログラム 8-102 ページの「OPM CL の例: ILE C プログラムからの OPM、COBOL、および RPG プログラムの呼び出し」の ILE バージョンです。
図 8-94 基本オブジェクト構造

この例は、以下の要素から構成されます。
- ユーザー入力を受け入れて ILE 制御言語プログラムに渡す CL コマンド。
- 入力を処理して ILE C プログラムに渡す ILE 制御言語プログラム。
- ILE プロシージャーを呼び出して入力を処理する ILE C プログラム。その後、出力は、ユーザーの端末に書き込まれます。
- 計算を完了し、コストをフォーマット設定する ILE COBOL プロシージャー。
- 各トランザクションで監査ファイルを更新する ILE RPG プロシージャー。

注: CMD, ILE CL, ILE C, ILE RPG, および ILE COBOL のソースに加え、出力ファイルの DDS ソース (前の例と同じ) が必要です。

プログラム・モジュールおよび活発化グループ：8-112 ページの図 8-95 に示したように、制御言語プログラムおよび C プログラムは、新規活発化グループ内でアクティブ化されています。ILE 制御言語プログラムは、ACTGRP パラメーターにデフォルトの ACTGRP(*NEW) を指定した CRTPGM によって作成されています。ILE C プログラムは、ACTGRP(*CALLER) を指定して作成されています。
プログラミング・タスク：以下のステップでは、次を行う方法を示します。

- ILE C プログラムの監査証跡を入力物理ファイルを作成する。
- ILE C プログラムに品名、価格、数量、およびユーザー ID の各パラメーターを渡す ILE 制御言語プログラムを作成する。
- 品名、価格、および数量のデータを入力するための CL コマンド・プロンプトを作成する。コマンドは、データを ILE 制御言語プログラムに渡し、その ILE 制御言語プログラムは ILE C プログラムを呼び出します。
- 税金計算を実行し、合計コストをフォーマットする ILE COBOL モジュールを作成する。
- アプリケーションの監査証跡を書き込む ILE RPG モジュールを作成する。
- データ項目をエクスポートするサービス・プログラムを作成する。
- RPG プロシージャーをエクスポートするサービス・プログラムを作成する。
- main() 関数を使用して、制御言語プログラムから入力引数を受け取る 1 つのプログラムを作成する。このプログラムは、税金計算を実行し、合計コストをフォーマットする ILE COBOL プロシージャーを呼び出し、さらに監査レコードを書き込む ILE RPG プロシージャーを呼び出します。

1. 8-113 ページの図 8-96 に示したソースを使用して物理ファイル T1520DD2 を作成するには、以下を入力します。

```
CRTPF FILE(MYLIB/T1520DD2) SRCFILE(QCPPLE/QADDSSRC) MAXMBRS(*NOMAX)
```
このファイルには、ILE C プログラム T1520ICB の監査証跡が入ります。

2. 図 8-96 に示したソースを使用して制御言語プログラム T1520CL3 を作成するには、以下を入力します。

CRTCLMOD MODULE(MYLIB/T1520CL3) SRCFILE(QCPPLE/QACLSSRC)
CRTPGM PGM(MYLIB/T1520CL3) MODULE(MYLIB/T1520CL3) ACTGRP(*NEW)

/* ILE version of T1520CL2 */
PGM PARM(&ITEMIN &PRICE &QUANTITY)
   DCL VAR(&USER) TYPE(*CHAR) LEN(10)
   DCL VAR(&ITEMIN) TYPE(*CHAR) LEN(20)
   DCL VAR(&ITEMOUT) TYPE(*CHAR) LEN(21)
   DCL VAR(&PRICE) TYPE(*DEC) LEN(10 2)
   DCL VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
   /* ADD NULL TERMINATOR FOR THE ILE C PROGRAM */
   CHGVAR VAR(&ITEMOUT) VALUE(&ITEMIN *TCAT &NULL)
   /* GET THE USERID FOR THE AUDIT FILE */
   RTVJOBA USER(&USER)
   /* ENSURE AUDIT RECORDS WRITTEN TO CORRECT AUDIT FILE MEMBER */
   OVRDBF FILE(T1520DD2) TOFILE(*LIBL/T1520DD2) +
                     MBR(T1520DD2) OVRSCOPE(*CALLLVL) SHARE(*NO)
   CALL PGM(T1520ICB) PARM(&ITEMOUT &PRICE &QUANTITY + &USER)
   DLTOVR FILE(*ALL)
ENDPGM

注: ILE プログラムの実行時にパラメーターをそれに渡す場合、CALL 呼び出し (CALL) コマンドの PARM オプションを使用します。

図 8-97. T1520CL3 — ILE C プログラムに変数を渡すための ILE CL ソース

このプログラムは、品名、価格、数量、およびユーザー ID の各 CL 変数を参照によって ILE C プログラム T1520ICB に渡します。ジョブ属性検索 (RTVJOBA) コマンドは、監査証跡のユーザー ID を取得します。引数は参照によって渡されます。引数は、受け取り側の ILE C プログラムが変更できます。変数 item_name は制御言語プログラマ内でスル終了します。

3. 8-114 ページの図 8-98 のソースを使用して CL コマンド・プロンプト T1520CM2 を作成するには、以下を入力します。

CRTCMD CMD(MYLIB/T1520CM2) PGM(MYLIB/T1520CL3) SRCFILE(QCPPLE/QACMDSRC)
/* This program demonstrates the interlanguage call capability */
/* of an ILE C program. This program is called by a CL */
/* program that passes an item name, price, quantity and user ID. */
/* A COBOL procedure is called to calculate and format total cost. */
/* An RPG procedure is called to write an audit trail. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>

#pragma map(calc_and_format,"T1520CB2")
#pragma map(write_audit_trail,"T1520RP2")

void calc_and_format(decimal (10,2),
                      short int,
                      char [],
                      char *);

void write_audit_trail(char [],
                        char [],
                        decimal (10,2),
                        short int,
                        char []);

extern decimal (2,2) TAXRATE; /* TAXRATE is in *SRVPGM T1520SP3 */

int main(int argc, char *argv[])
{
    char *user_id;
    char *item_name;
    short int quantity;
    decimal (10,2) price;
    char formatted_cost[22];
    char rpg_item_name[20];
    char null_formatted_cost[22];
    char success_flag = 'N';
    int i;

    item_name = argv[1];
    price = *((decimal (10,2) *) argv[2]);
    quantity = *((short *) argv[3]);
    user_id = argv[4];
/* Call the COBOL program to do the calculation, and return a */
/* Y/N flag, and a formatted result. */
calc_and_format(price,
quantity,
formatted_cost,
&success_flag);

memcpy(null_formatted_cost,formatted_cost,sizeof(formatted_cost));

/* Null end the result. */
formatted_cost[21] = '\0';
if (success_flag == 'Y')
{
    for (i=0; i<20; i++)
    {
        /* Remove the null end for the RPG for iSeries program. */
        if (*(item_name+i) == '\0')
        {
            rpg_item_name[i] = '' ;
        }
        else
        {
            rpg_item_name[i] = *(item_name+i);
        }
    }
    /* Call an RPG program to write audit records. */
    write_audit_trail(user_id,
rpg_item_name,
price,
quantity,
formatted_cost);

    printf("\n%d %s plus tax = %-s\n", quantity,
        item_name,
        );
}
else
{
    printf("Calculation failed\n");
}

注:
a. このモジュールの main() 関数は、CL コマンド T1520CM2 によって検証され、ILE 制御言語プログラム T1520CL3 内でヌルで終了する入力引数をこの制御言語プログラム T1520CL3 から受け取ります。すべての入力引数はポインターです。
b. このプログラムの main() 関数は、ILE COBOL プロシージャ名にマップされた calc_and_format() を呼び出します。これは、OS リンクージェイション規則によって価格、数量、フォーマットされたコスト、および成功フラグを渡します。ILE OPM COBOL プロシージャーは、ILE C のデフォルトである拡大されたパラメーターを要求しません。このため、#pragma argument ディレクティブでは拡大なしが使用されます。フォーマットされたコストおよび成功フラグの値は、プロシージャー T1520CB2 で更新されます。
c. calc_and_format() が正常に main() 関数を戻した場合、このプログラム (T1520ICB) は ILE RPG プロシージャ名にマップされた write_audit_trail() を呼び出します。さらに、OS リンクージェイション規則（例参照とも呼ばれます）によって、ユーザー ID、品名、価格、数量、およびフォーマットされたコストを渡します。ILE C コンパイラは、#pragma argument ディレクティブに拡大なしパラメーターが指定されていない限り、デフォルトで短整数を整数に変換します。例えば、ILE C プロ
グラムの短整数は整数に変換されてから、ILE RPG プロシージャーに渡されます。RPG プロシージャーは、数量変数には 4 バイトの整数を必要とします。

5. 図 8-99 に示したソースを使用してモジュール T1520ICC を作成するには、以下を入力します。

```c
CRTCMOD MODULE(MYLIB/T1520ICC) SRCFILE(QCPPLE/QACSRC)
```

/* Export the tax rate data. */
#include <decimal.h>
const decimal (2,2) TAXRATE = .15D;

図 8-99. T1520ICC — 税率データをエクスポートするためのソース・コード

TAXRATE は、このモジュールから ILE の C, COBOL, および RPG プロシージャーにエクスポートされます。

注: 弱定義 (COBOL の EXTERNAL) は、サービス・プログラムから C などの強定義言語にエクスポートできません。C は COBOL にエクスポートできます。そのため、TAXRATE の言語として選択されます。

6. 8-117 ページの図 8-100 に示したソースを使用して ILE COBOL プロシージャーを作成するには、以下を入力します。

```c
CRTCBLMOD MODULE(MYLIB/T1520CB2) SRCFILE(QCPPLE/QALBLSRC)
```
IDENTIFICATION DIVISION.
PROGRAM-ID. T1520CB2 INITIAL.
****************************
* parameters:             *
* incoming:  PRICE, QUANTITY *
* returns:   TOTAL-COST (PRICE*QUANTITY+1.TAXRATE) *
* SUCCESS-FLAG.          *
* TAXRATE : An imported value.   *
******************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-I.
OBJECT-COMPUTER. IBM-I.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 WS-TAXRATE PIC S9V99 COMP-3 VALUE 1.
  01 TAXRATE EXTERNAL PIC SV99 COMP-3.
LINKAGE SECTION.
  01 LS-PRICE PIC S9(8)V9(2) COMP-3.
  01 LS-QUANTITY PIC S9(4) COMP-4.
  01 LS-TOTAL-COST PIC $$$$,$$$,$$$,$$$,$$99 DISPLAY.
  01 LS-OPERATION-SUCCESSFUL PIC X DISPLAY.
PROCEDURE DIVISION USING LS-PRICE
    LS-QUANTITY
    LS-TOTAL-COST
    LS-OPERATION-SUCCESSFUL.
MAINLINE.
  MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
  PERFORM CALCULATE-COST.
  PERFORM FORMAT-COST.
  EXIT PROGRAM.
CALCULATE-COST.
  ADD TAXRATE TO WS-TAXRATE.
  COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY *
    LS-PRICE *
    WS-TAXRATE
    ON SIZE ERROR
      MOVE "N" TO LS-OPERATION-SUCCESSFUL
    END-COMPUTE.
FORMAT-COST.
  MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

図 8-100. T1520CB2 － 税金を計算し、コストをフォーマットするための ILE COBOL ソース

注:

a. このプログラムは、変数の価格および数量の値を指すポインター、および formatted_cost および
success_flag を指すポインターを受け取ります。

b. calc_and_format() 関数はプロシージャー T1520CB2 です。これは、合計コストを計算し、フォーマットします。

c. ILE COBOL ソース・プログラムのコンパイルについては、ILE COBOL プログラマーの手引きを
参照してください。

7. [8-118 ページの図 8-101] に示したソースを使用して ILE RPG プロシージャーを作成するには、以下を
入力します。

CRTRPGMOD MODULE(MYLIB/T1520RP2) SRCFILE(QCPPLE/QARPGSRC)
図 8-101. T1520RP2 に監査証跡を書き込むための ILE RPG ソース

注:

a. `write_audit_trail()` 関数はプロシージャー T1520RP2 です。これは、プログラムの監査証跡を書き込みます。

b. ILE RPG ソース・プログラムのコンパイル方法については、ILE RPG プログラマーの手引きを参照してください。

8. モジュール T1520ICC からサービス・プログラム T1520SP3 を作成するには、以下を入力します。

   CRTSRVPGM SRVPGM(MYLIB/T1520SP3) MODULE(MYLIB/T1520ICC) +
   EXPORT(*SRCFILE) SRCFILE(QCPPLE/QASRVSRC)

   T1520SP3 サービス・プログラムは税率をエクスポートします。エクスポート・リストは QASRVSRC 内の T1520SP3 に指定されています。

9. モジュール T1520RP2 からサービス・プログラム T1520SP4 を作成するには、以下を入力します。

   CRTSRVPGM SRVPGM(MYLIB/T1520SP4) MODULE(MYLIB/T1520RP2) +
   EXPORT(*SRCFILE) SRCFILE(QCPPLE/QASRVSRC)

   T1520SP4 サービス・プログラムはプロシージャー T1520RP2 をエクスポートします。エクスポート・リストは QASRVSRC 内の T1520SP4 に指定されています。

10. プログラム T1520ICB を作成するには、以下を入力します。

    CRTPGM PGM(MYLIB/T1520ICB) MODULE(MYLIB/T1520ICB MYLIB/T1520CB2) +
    BNDSRVPGM(MYLIB/T1520SP3 MYLIB/T1520SP4) ACTGRP(*CALLER)

    T1520ICB は、アプリケーションのメインプログラムとみなされます。これは、T1520CL3 の呼び出し時に作成された新規活動化グループで実行します。

11. プログラム T1520ICB のデータを入力するには、T1520CM2 を入力して、F4 (プロンプト) を押します。

    以下のデータを T1520CM2 に入力します。

    Hammers
    1.98
    5000
    Nails
    0.25
    2000

8-118 IBM i: ILE C/C++ プログラマーの手引き
出力は以下のようにになります。

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5000 HAMMERS plus tax =</td>
<td>$11,385.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>実行キーを押して端末セッションを終了してください。</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000 NAILS plus tax =</td>
<td>$575.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>実行キーを押して端末セッションを終了してください。</strong></td>
</tr>
</tbody>
</table>

物理ファイル T1520DD2 は、以下のデータが含まれます。

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITHE</td>
<td>HAMMERS</td>
<td>00000000198500015</td>
<td>$11,385.0072893</td>
</tr>
<tr>
<td>SMITHE</td>
<td>NAILS</td>
<td>00000000252000015</td>
<td>$575.0072893</td>
</tr>
</tbody>
</table>

**ILE-OPM CL の例: ILE C++ プログラムからの OPM、COBOL、RPG プログラムの呼び出し**

このプログラムでは、複数の ILE および OPM プログラミング言語を使用するプログラムを作成する際の、一般的なステップの例を示しています。

**プログラムの説明:** このプログラムは、入力として 1 つ以上のプロダクトの項目名、価格、数量を取り入れる小さなトランザクション処理プログラムです。出力として、このプログラムは、画面に指定された項目の合計コストを表示し、トランザクションの監査証跡をファイルに書き込みます。

図 8-102 に、このプログラムの基本的な流れを示します。

**プログラム構造:** プログラムは、以下のコンポーネントから構成されます。
プログラマの活動化: ILE C++ プログラム T21231C5 は、CRTPGM における ACTGRP パラメーターのデフォルトの ACTGRP(*NEW) で作成されます。制御言語プログラムが ILE C++ プログラムを呼び出すと、新規活動化グループが開始されます。

OPM CL、COBOL、および RPG のプログラムは、OPM のデフォルト活動化グループ内で活動化されます。

図 8-103 は、ILE におけるこのプログラムの構造を示しています。

図 8-103. ILE C++ におけるプログラムの構造

プログラム・ファイル: このプログラムを構成する各ファイルのソース・コードは、外部記述ファイル、制御言語プログラム、CL コマンド・プロンプト、C++ ソース・ファイル、および OPM COBOL プログラムと OPM RPG プログラムです。

外部記述ファイル T2123DD2: ファイル T2123DD2 には、C++ プログラム T21231C5 の監査証跡が含まれます。次のような DDS ソースで監査ファイルのフィールドが定義されています。

```plaintext
R T2123DD2R
  USER  10  COLHDG('User')
  ITEM  20  COLHDG('Item name')
  PRICE 10S 2  COLHDG('Unit price')
  QTY  4S  COLHDG('Number of items')
```
制御言語プログラム T2123CL2: 制御言語プログラム T2123CL2 は、CL 变数 item_name、price、quantity、および user_id を参照によって ILE C++ プログラム T2123IC5 に受け渡します。

PGM  PARM(&ITEMIN &PRICE &QUANTITY)
DCL VAR(&USER) TYPE(*CHAR) LEN(10)
DCL VAR(&USEROUT) TYPE(*CHAR) LEN(11)
DCL VAR(&ITEMIN) TYPE(*CHAR) LEN(20)
DCL VAR(&ITEMOUT) TYPE(*CHAR) LEN(21)
DCL VAR(&PRICE) TYPE(*DEC) LEN(10 2)
DCL VAR(&QUANTITY) TYPE(*DEC) LEN(2 0)
DCL VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
/* ADD NULL TERMINATOR FOR THE ILE C PROGRAM */
CHGVAR VAR(&ITEMOUT) VALUE(&ITEMIN *TCAT &NULL)
CHGVAR VAR(&USEROUT) VALUE(&USER *TCAT &NULL)
/* GET THE USERID FOR THE AUDIT FILE */
RTVJOBA USER(&USER)
/* ENSURE AUDIT RECORDS WRITTEN TO CORRECT AUDIT FILE MEMBER */
OVRDBF FILE(T2123DD2) TOFILE(*LIBL/T2123DD2) +
MBR(T2123DD2) OVRSCOPE(*CALLLVL) SHARE(*NO)
CALL PGM(T2123IC5) PARM(&ITEMOUT &PRICE &QUANTITY +
&USEROUT)
DLTOVR FILE(*ALL)
ENDPGM

ジョブ属性検出 (RTVJOBA) コマンドは、監査証跡のユーザー ID を取得します。引数は参照によって受け渡されます。これらは、受け取り側の ILE C++ プログラムによって変更可能です。ユーザー名と項目名を含む変数は、制御言語プログラムで明示的にヌル終了する処理が行われます。

注:
1. CL 変数と数値リテラルは、ヌル終了ストリングを使用して ILE C++ プログラムに受け渡されません。
2. 文字リテラルと論理リテラルは、ヌル終了ストリングとして受け渡されますが、プランクで拡張されません。
3. バック 10 進数などの数値リテラルは、15.5 (8 バイト) として受け渡されます。浮動小数点の定数は、倍精度浮動小数点値 (1.2E+15) として受け渡されます。
4. ILE プログラムの実行時にパラメーターをそれに渡す場合、CALL 呼び出し (CALL) コマンドの PARM オプションを使用します。

CL コマンド・プロンプト T2123CM2: CL コマンド・プロンプト T2123CM2 を使用して、C++ プログラム T2123IC5 によって使用される項目名、価格、および数量の入力をユーザーに求めます。

CMD  PROMPT('CALCULATE TOTAL COST')
PARM  KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
MIN(1) ALWNPRT(*NO) PROMPT('Item name' 1)
PARM  KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
RANGE(0.01 99999999.99) MIN(1) +
ALWNPRT(*YES) PROMPT('Unit price' 2)
PARM  KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
9999) MIN(1) ALWNPRT(*YES) +
PROMPT('Number of items' 3)
C++ ソース・ファイル T212IC5: C++ ソース・ファイル T212IC5 に含まれる main() 関数では、制御言語プログラム T212CL2 から入力引数を受け取ります。これらの引数は CL コマンド・プロンプト T212CM2 で検証され、制御言語プログラム T212CL2 の内でヌル終了の処理が行われています。すべての入力引数はポインタです。

main() 関数は、COBOL 名にマップされた関数 CalcAndFormat() を呼び出します。 OPM COBOL プログラムは拡張されたパラメーターを予期していないため、extern "OS nowiden" のリンクージ指標を使用して、price、quantity、taxrate、formatted_cost、および success_flag を OPM COBOL プログラム T212CB1 に受け渡します。

formatted_cost と success_flag の値が、C++ プログラム T212IC5 で更新されます。

CalcAndFormat() が正常終了で戻ると、OPM RPG プログラム内の WriteAuditTrail() によって監査証跡にレコードが書き込まれます。

プログラム T212IC5 の main() 関数は、RPG プログラム名にマップされた WriteAuditTrail() を呼び出し、extern "OS" のリンクージ指標を使用して、user_id、item_name、price、quantity、taxrate、および formatted_cost を受け渡します。

注: デフォルトでコンパイラは、extern のリンクージ指標で nowiden パラメーターが指定されていない限り、short の整数を integer に変換します。C++ プログラム内の short の整数は integer に変換されてから、OPM RPG プログラムに受け渡されます。RPG プログラムは、quantity 変数に 4 パイト整数を予期しています。データ・タイプの互換性については、11-1 ページの『言語間のデータ・タイプ互換』を参照してください。

// This program is called by a CL program that passes an item
// name, price, quantity and user ID.
// COBOL is called to calculate and format the total cost.
// RPG is called to write an audit trail.

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <bcd.h>

// The #pragma map directive maps a new program name to the existing
// program name so that the purpose of the program is clear.
// Tell the compiler that there are dynamic program calls so
// arguments are passed by value-reference.

extern "OS nowiden" void CalcAndFormat(_DecimalT<10,2>,
short int, _DecimalT<2,2>, char[],
char *);

#pragma map(CalcAndFormat,"T2123CB1")

extern "OS" void WriteAuditTrail(char[], char[] ,
_DecimalT<10,2>, short int,
_DecimalT<2,2>, char[]);

#pragma map(WriteAuditTrail,"T2123RP1")

int main(int argc, char *argv[])
{
// Incoming arguments from a CL program have been verified by
// the "CMD and null-terminated within the CL program.
// Incoming arguments are passed by reference from a CL program.

char       *user_id;
char       *item_name;

short int quantity;
.DecimalT<10, 2> price;
.DecimalT<2,2> taxrate = D(".15");
char formatted_cost[22];

// Remove null terminator for RPG program. Item name is null
// terminated for C++.

char rpg_item_name[20];
char null_formatted_cost[22];
char success_flag = 'N';
int i;

// Incoming arguments are all pointers.

item_name = argv[1];
price = *((_DecimalT<10, 2> *) argv[2]);
quantity = *((short *) argv[3]);
user_id = argv[4];

// Call the COBOL program to do the calculation, and return a
// Y/N flag, and a formatted result.

CalcAndFormat(price, quantity, taxrate, formatted_cost,
&success_flag);

memcpy(null_formatted_cost,formatted_cost,sizeof(formatted_cost));

// Null terminate the result.

formatted_cost[21] = '\0';
if (success_flag == 'Y')
{
    for (i=0; i<20; i++)
    {
        // Remove null terminator for the RPG program.
        if (*(item_name+i) == '\0')
        {
            rpg_item_name[i] = '\';
        }
        else
        {
            rpg_item_name[i] = *(item_name+i);
        }
    }

    // Call an RPG program to write audit records.
    WriteAuditTrail(user_id, rpg_item_name, price, quantity,
taxrate, formatted_cost);

    cout <<quantity <<item_name << "plus tax = "
    <<null_formatted_cost <<endl;
}
else
{
    cout <<"Calculation failed" <<endl;
}


**OPM COBOL プログラム T2123CB1:** OPM COBOL プログラム T2123CB1 は、変数 price、quantity、
および taxrate の値へのポインターと、formatted_cost および success_flag へのポインターを受け取
ります。
プログラム T2123CB1 の CalcAndFormat() 関数は、合計コストの計算とフォーマット設定を行います。

ILE C++ プログラムから OPM COBOL プログラムにパラメーターが受け渡され、税額を計算します。

IDENTIFICATION DIVISION.
PROGRAM-ID. T2123CB1.
*****************************************************
* parameters: *
* incoming: PRICE, QUANTITY *
* returns : TOTAL-COST (PRICE*QUANTITY*1.TAXRATE) *
* SUCCESS-FLAG. *
*****************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-I.
OBJECT-COMPUTER. IBM-I.
DATA DIVISION.
WORKING-STORAGE SECTION.

01 WS-TAXRATE PIC S9V99 COMP-3.

LINKAGE SECTION.

01 LS-PRICE PIC S9(8)V9(2) COMP-3.
01 LS-QUANTITY PIC S9(4) COMP-4.
01 LS-TAXRATE PIC SV99 COMP-3.
01 LS-TOTAL-COST PIC $$,$$$,$$$,$$$,$$$$.99 DISPLAY.
01 LS-OPERATION-SUCCESSFUL PIC X DISPLAY.


MAINLINE.
MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
PERFORM CALCULATE-COST.
PERFORM FORMAT-COST.
EXIT PROGRAM.

CALCULATE-COST.
MOVE LS-TAXRATE TO WS-TAXRATE.
ADD 1 TO WS-TAXRATE.
COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY * LS-PRICE * WS-TAXRATE
ON SIZE ERROR
MOVE "N" TO LS-OPERATION-SUCCESSFUL
END-COMPUTE.

FORMAT-COST.
MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

OPM RPG プログラム T2123RP1: OPM RPG プログラム T2123RP1 には、プログラムの監査証跡を書き込む WriteAuditTrail() 関数が含まれています。

FT2123DD2O E DISK A
F T2123DD2R KRENAMED2R
IQTYYIN DS B 1 40QTYBIN
C *ENTRY PLIST
C PARM USER 10
C PARM ITEM 20
C PARM PRICE 102
C PARM QTYIN

8-124 IBM i: ILE C/C++ プログラマーの手引き
ILE-OPM プログラムの呼び出し: プログラム T2123IC5 にデータを入力するには、コマンド T2123CM2 を入力して F4 (ボンプット) を押します。

以下のデータを T2123CM2 に入力できます。

<table>
<thead>
<tr>
<th>Hammers</th>
<th>1.98</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>Nails</td>
<td>0.25</td>
</tr>
<tr>
<td>2000</td>
<td></td>
</tr>
</tbody>
</table>

出力は次のとおりです。

5000 HAMMERS plus tax = $11,385.00
実行キーを押して端末セッションを終了してください。

2000 NAILS plus tax = $575.00
実行キーを押して端末セッションを終了してください。

物理ファイル T2123DD2 には、次のデータが含まれています。

<table>
<thead>
<tr>
<th>SMITHE</th>
<th>HAMMERS</th>
<th>0000000198500015</th>
<th>$11,385.0007</th>
<th>2093</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITHE</td>
<td>NAILS</td>
<td>0000000025200015</td>
<td>$575.0007</td>
<td>2093</td>
</tr>
</tbody>
</table>

ILE プロシージャーを呼び出すためのリンクエージ指定の使用

C++ には、プロシージャー呼び出し、および C++ 呼び出し元と呼び出し先プロシージャー間でのデータの共有を可能にするリンクエージ指定が用意されています。構文については、「8-126 ページの「C++ 動的プログラム呼び出しにおけるリンクエージ指定の使用」を参照してください。

ILE プロシージャーを呼び出すためのリンクエージ指定の有効なストリング・リテラルは、以下のとおりです。

リンクエージ指定

呼び出されるプロシージャーのタイプ

"C++" ILE C++ プロシージャー (デフォルト)

"C" ILE C プロシージャー

"C nowiden"
拡張されたパラメーターのない ILE C プロシージャー

"RPG" ILE RPG プロシージャー

"COBOL"
ILE COBOL プロシージャー

"CL" ILE CL プロシージャー

"ILE" 一般 ILE 関数呼び出し
"ILE nowiden"
拡張されたパラメーターのない ILE 関数呼び出し

"VREF" 一時ストレージ内のポインターを使用した ILE 関数呼び出し。(パラメーターが参照による場合のように関数に渡される点を除いて、通常の呼び出しと同じ動作)。

"VREF nowiden"
拡張されたパラメーターがない点を除いて、"VREF" 同じ

### C++ 動的プログラム呼び出しにおけるリンケージ指定の使用
C++ プログラムから OPM、ILE、または EPM プログラムを呼び出すことができます。OPM、ILE、または EPM プログラムからもまた、C++ プログラムを呼び出すことができます。

#### C++
C++ が提供するリンケージ指定により、動的プログラム呼び出しおよびそれらの間のデータ共有が可能になります。構文図および追加情報については、「ILE C/C++ 解説書」を参照してください。

#### 有効なストリング・リテラル
"string-literal" は、特定の関数に関連したリンケージを指定するために使用されます。リンケージ指定で使用されるストリング・リテラルには、大小文字の区別はありません。プログラムを呼び出すためのリンケージ指定に有効なストリング・リテラルは以下のとおりです。

"OS" OS リンケージ呼び出し

"OS nowiden"
拡大されたパラメーターを使用しない OS リンケージ呼び出し。詳細については 8-82 ページの「関数の外部 (OS) リンケージの指定」を参照してください。

#### リンケージ指定

#### C++
C++ プログラムで ILE、OPM、または EPM プログラム (*PGM) を呼び出したい場合、C++ ソース内で extern "OS" リンケージ指定を使用し、呼び出されるプログラムがバインドされた ILE プロシージャーではなく、外部プログラムであることをコンパイラに指示します。例えば、C++ プログラムで OPM COBOL プログラム (*PGM) を呼び出したい場合、C++ ソース内のこの extern "OS" リンケージ指定は、COBOL_PGM がバインドされた ILE プロシージャーではなく、外部プログラムであることをコンパイラに指示します。

extern "OS" void COBOL_PGM(void);

ILE、OPM、または EPM プログラムで C++ プログラムを呼び出したい場合、ILE、OPM、または EPM 言語固有の CALL ステートメントを使用します。

#### C プログラムでのパック 10 進数データの使用

このトピックでは、以下を行うための方法について説明します。
- パック 10 進数データ・タイプから変換する
- パック 10 進数データへのポインターを関数に渡す
バック 10 進数データが含まれた別のプログラムを呼び出す
バック 10 進数データでライブラリー関数を使用する
バック 10 進数データ・タイプのエラーを理解する

バック 10 進数データ・タイプ表現には、整数部分と小数部分が含まれています。ILE C コンパイラでは、ISO C の拡張としてバック 10 進数データ・タイプがサポートされます。

注: これは、厳格に C データ・タイプです。bcd クラスで C++ 10 進数サポートが提供されます。ヘッダー・ファイルは bcd.h です。詳しくは、「ILE C/C++ 解説書」を参照してください。

バック 10 進数データ・タイプを使用して、以下を行うことができます。
・特に金融計算におけるビジネスおよび商用アプリケーションで、大きな数値を正確に表す。例えば、小数点に続く 2 桁によって、ドルの小数部分を正確に表すことができます。

注: 浮動小数点数演算を使用する必要はありません。浮動小数点は、多くの場合、以下のような数値を使用する科学および工学での計算により適しています。
- 最大のバック 10 進数変数が保持できるよりもはるかに大きい
- 最小のバック 10 進数よりもはるかに小さいが、商用に十分な精度がない

・バック 10 進数メンバーを持つタイプ定義、配列、構造体、および共用体を宣言する。バック 10 進数変数に対して演算子（単項演算子）を適用できます。バック 10 進数データには、ビット単位演算は適用されません。ILE C におけるバック 10 進数データ・タイプは、RPG および COBOL におけるバック 10 進数表現と互換性があります。マクロを定義し、バック 10 進数引数を指定してライブラリー関数を呼び出すこともできます。「ILE C/C++ 解説書」に、バック 10 進数データ・タイプに関する情報が含まれています。

注: コードで decimal、digitsof、および precisionof マクロを使用するには、ILE C ソースで <decimal.h> ヘッダー・ファイルを指定する必要があります。

バック 10 進データ型からの変換
変換されるバック 10 進型の値が、正確に表現できる値の範囲内にある場合、そのバック 10 進型の値は変更されません。バック 10 進数は、その型が同じであれば互換性があります。例えば、((n_1 = n_2) かつ (p_1 = p_2)) である場合に限り、decimal(n_1, p_1) と decimal(n_2, p_2) は互換性のある型になります。

バック 10 進数型からバック 10 進数型への変換
以下の例は、バック 10 進数型からサイズの異なるバック 10 進数型へのさまざまな変換を示しています。変換されるバック 10 進数型の値が正確に表現できる値の範囲内にない場合、バック 10 進数型の値が切り捨てられます。小数部分の切り捨てが発生した場合、結果は切り捨てられ、実行時エラーはありません。

```c
#include <decimal.h>
int main (void)
{
  decimal(4,2) targ_1, targ_2;
  decimal(6,2) op_1=1234.56d, op_2=12.34d;
  targ_1=op_1;  /* A runtime error is generated because the integral
                  part is truncated; targ_1=34.56d. */
  targ_2=op_2;  /* No runtime error is generated because neither the
                  integral nor the fractional part is truncated;
                  targ_2=12.34d. */
}
```

図 8-104. バック 10 進数に変換するための ILE C ソース
代入で整数部分に切り捨てが発生した場合、実行時エラーとなります。実行時例外は、変換を必要とする操作に関係なく、異なる型への変換で整数値が失われたときに発生します。実行時例外の例については、8-136 ページの『バック 10 進数データ・タイプ・エラーの理解』を参照してください。

例:

より小さいターゲットへの代入でのコンパイル時には、警告またはエラーは発生しません。変換の際のコンパイル時エラーおよび実行時エラーについては、8-136 ページの「バック 10 進数データ・タイプ・エラーの理解」を参照してください。

以下の例は、あるバック 10 進数型からより小さい精度の別のバック 10 進数型への変換を示しています。結果の小数部分で切り捨てが起こります。

```c
#include <decimal.h>
int main(void)
{
    decimal(7,4) x = 123.4567D;
    decimal(7,1) y;
    y = x; /* y = 123.4D */
}
```

図 8-105. バック 10 進数をより小さい精度のバック 10 進数に変換するための ILE C プログラム

次の例は、1 つのバック 10 進数型からより小さい整数部分を持つ別のバック 10 進数型への変換を示しています。結果の整数部分で切り捨てが起こります。 #pragma nosigtrunc ディレクティブは、オーバーフローによる例外生成をオフにします。

```c
#pragma nosigtrunc
#include <decimal.h>
int main(void)
{
    decimal(8,2) x = 123456.78D;
    decimal(5,2) y;
    y = x; /* y = 456.78D */
}
```

図 8-106. バック 10 進数をより小さい整数部分のバック 10 進数に変換するための ILE C プログラム

次の例は、1 つのバック 10 進数型からより小さい整数部分とより小さい精度を持つ別のバック 10 進数型への変換を示しています。結果の整数部分と小数部分の両方で切り捨てが起こります。 #pragma nosigtrunc ディレクティブは、オーバーフローによる例外生成をオフにします。

```c
#pragma nosigtrunc
#include <decimal.h>
int main(void)
{
    decimal(8,2) x = 123456.78D;
    decimal(4,1) y;
    y = x; /* y = 456.7D */
}
```

図 8-107. バック 10 進数をより小さい整数部分とより小さい精度のバック 10 進数に変換するための ILE C プログラム

8-128 IBM i: ILE C/C++ プログラマーの手引き
バック 10 進数型から整数型への変換
バック 10 進数型の値を整数型に変換すると、値はバック 10 進数 (20.0) になり、その後これが整数型になります。整数型のサイズによっては、上位ビットが切り捨てられます。バック 10 進数を整数型に代入し、その結果、整数部分に切り捨てが起こる場合、実行時例外は発生しません。

例:
以下の例は、小数部分を持つバック 10 進数型から整数型への変換を示しています。

```c
#include <decimal.h>
int main (void)
{
    int op;
    decimal(7,2) op1 = 12345.67d;
    op = op1;    /* Truncation on the fractional */
    /* part. op=12345 */
}
```

図 8-108. 小数部分を持つバック 10 進数を整数に変換するための ILE C ソース

以下の例は、整数部分が 10 桁に満たないバック 10 進数型から整数型への変換を示しています。

```c
#include <decimal.h>
int main(void)
{
    int op;
    decimal(3) op2=123d;
    op = op2;    /* No truncation and op=123 */
}
```

図 8-109. 整数部分が 10 桁に満たないバック 10 進数を整数に変換するための ILE C ソース

以下の例は、整数部分が 10 桁を超えるバック 10 進数型から整数型への変換を示しています。

```c
#include <decimal.h>
int main (void)
{
    int op;
    long long op_2;
    decimal(15,2) op_1 = 1234567890123.12d;
    op = op_1;    /* High-order bits will be truncated. */
    /* op2 = 0x71FB04CB */
}
```

図 8-110. 整数部分が 10 桁を超えるバック 10 進数を整数に変換するための ILE C ソース

以下の例は、小数部分を持ち、整数部分が 10 桁を超えるバック 10 進数型から整数型への変換を示しています。

```c
#include <decimal.h>
int main (void)
{
    int op;
    long long op_2;
    decimal(15,2) op_1 = 1234567890123.12d;
    op = op_1;    /* High-order bits will be truncated. */
    /* op2 = 0x71FB04CB */
}
```

図 8-111. 小数部分を持ち、整数部分が 10 桁を超えるバック 10 進数を整数に変換するための ILE C ソース
バック 10 進数型から浮動小数点型への変換
バック 10 進数型の値を浮動小数点型に変換する場合、変換される値が表現できる値の範囲外にあると、振る舞いは未定義のものになります。変換される値が表現できる値の範囲内にあるが、正確に表現できない場合、結果が切り捨てられます。float または double をより小さい精度のバック 10 進数に変換すると、
float または double の小数部分が切り捨てられます。

以下の例は、バック 10 進数型から浮動小数点型への変換を示しています。

```c
#include <decimal.h>
#include <stdio.h>
int main(void)
{   
    decimal(5,2) dec_1=123.45d;
    decimal(11,5) dec_2=-123456.12345d;
    float f1,f2;
    f1=dec_1;
    f2=dec_2;
    printf("f1=%f
f2=%f
",f1,f2);
    /* f1=123.449997 */
    /* f2=-123456.125000 */
}
```

図 8-112. バック 10 進数を浮動小数点に変換するための ILE C ソース

出力は以下のようになります。

```
f1=123.449997
f2=-123456.125000
```
実行キーを押して終了セッションを終了してください。

オーバーフローの振る舞い
以下の表では、バック 10 進数がより小さいターゲットに代入されるときのオーバーフローの振る舞いについて説明します。以下の場合には例外が生成されません。

| バック 10 進数型のターゲット | オーバーフロー
|---------------------------|----------
| バック 10 進数型            | はい
| バック 10 進数型            | はい
| バック 10 進数型            | はい
| バック 10 進数型            | はい

表 8-113. バック 10 進数からより小さいターゲットへの代入の場合のオーバーフローの処理

<table>
<thead>
<tr>
<th>取り出しフィールド</th>
<th>収束フィールド</th>
<th>実行時エラー</th>
</tr>
</thead>
<tbody>
<tr>
<td>バック 10 進数</td>
<td>char, int, short, long, long, long, bit</td>
<td>いいえ</td>
</tr>
<tr>
<td>バック 10 進数</td>
<td>バック 10 進数</td>
<td>はい</td>
</tr>
<tr>
<td>バック 10 進数</td>
<td>浮動</td>
<td>いいえ</td>
</tr>
<tr>
<td>バック 10 進数</td>
<td>Double</td>
<td>いいえ</td>
</tr>
</tbody>
</table>

注:
1. バック 10 進数が double に代入される場合にオーバーフローを引き起こすような大きさのバック 10 進数はありません。
2. MI 命令 setca を使用して浮動小数点例外をアンマスクすると、浮動小数点の不正確な結果に対してエラー・メッセージ MCH1213 を受け取ります。
パック 10 進数データの関数への受け渡し
呼び出される関数にプロトタイプが含まれていない場合、パック 10 進数型の引数にはデフォルト引数プロモーションはありません。つまり、パック 10 進数の引数を含む関数定義はすべて、プロトタイプ化される必要があります。そうでない場合、その動作は未定義です。パック 10 進数型の引数の境界合わせは、パック 10 進数型のサイズによって異なります。詳しくは、以下のとおりです。

* 1 <= n <= 7 の場合、4 バイト境界で位置合わせ
* 8 <= n <= 15 の場合、8 バイト境界で位置合わせ
* 16 <= n <= 31 の場合、16 バイト境界で位置合わせ
* 32 <= n <= 63 の場合、32 バイト境界で位置合わせ

次の例は、パック 10 進数の変数を関数に受け渡す方法を示しています。

```c
#include <decimal.h>
#include <stdio.h>
decimal(3,1) d1 = 33.3d;
decimal(10,5) d2 = 55555.55555d;
decimal(28) d3 = 8888888888888888888888888888d;
void func1(decimal(3,1), decimal(10,5),
           decimal(10,5), decimal(28));
int main(void) {
    func1(d1, d2, 999.99d, d3); /* with prototype */
    /* The arguments are passed as followed*/
    /* 333f0000 00000000 05555555 555f0000 */
    /* 99999f00 00000000 00000000 00000000 */
    /* 88888888 88888888 88888888 88888f00 */
}
/* func1 is prototyped */
void func1(decimal(3,1) x1, decimal(10,5) x2,
           decimal(10,5) x3, decimal(28) x4) {
    /* no runtime error when referencing x1, x2, x3 or x4 */
    printf("x1 = D(3,1)\n", x1);
    printf("x2 = D(10,5)\n", x2);
    printf("x3 = D(10,5)\n", x3);
    printf("x4 = D(28)\n", x4);
}
```

図 8.113. パック 10 進数の変数を関数に受け渡すための ILE C のソース

出力は以下のようになります。

```
x1 = 33.3
x2 = 55555.5555
x3 = 999.99
x4 = 88888888888888888888888888888
```

関数へのパック 10 進変数を指すポインターの引き渡し
以下の例は、関数にパック 10 進変数を指すポインターを引き渡す方法を示しています。
/* This program shows how to pass a pointer to a packed decimal */
/* variable to a function. */
#include <decimal.h>
#include <stdio.h>
decimal(5,2) var=123.45d;
decimal(5,2) *p=&var;
decimal(5,2) *func_1(decimal(5,2) *);

int main(void)
{
    /* Call function with pointer to packed decimal argument. The */
    /* value that it returns is also a pointer to a packed decimal. */
    if(func_1(p)!=p)
    {
        printf("Function call not successful\n\n");
    }
    else
    {
        printf("The packed decimal number is: %D(5,2)\n",*func_1(p));
    }
    decimal(5,2) *func_1(decimal(5,2) *q)
    {
        return q;
    }
}

/* This function returns a pointer to a packed decimal */

/* This program shows how to pass a pointer to a packed decimal */
/* variable to a function. */
#include <decimal.h>
#include <stdio.h>
decimal(5,2) var=123.45d;
decimal(5,2) *p=&var;
decimal(5,2) *func_1(decimal(5,2) *);

int main(void)
{
    /* Call function with pointer to packed decimal argument. The */
    /* value that it returns is also a pointer to a packed decimal. */
    if(func_1(p)!=p)
    {
        printf("Function call not successful\n\n");
    }
    else
    {
        printf("The packed decimal number is: %D(5,2)\n",*func_1(p));
    }
    decimal(5,2) *func_1(decimal(5,2) *q)
    {
        return q;
    }
}

/* This function returns a pointer to a packed decimal */

/* This program shows how to pass a pointer to a packed decimal */
/* variable to a function. */
#include <decimal.h>
#include <stdio.h>
decimal(5,2) var=123.45d;
decimal(5,2) *p=&var;
decimal(5,2) *func_1(decimal(5,2) *);

int main(void)
{
    /* Call function with pointer to packed decimal argument. The */
    /* value that it returns is also a pointer to a packed decimal. */
    if(func_1(p)!=p)
    {
        printf("Function call not successful\n\n");
    }
    else
    {
        printf("The packed decimal number is: %D(5,2)\n",*func_1(p));
    }
    decimal(5,2) *func_1(decimal(5,2) *q)
    {
        return q;
    }
}
#include<stdio.h>
#include <decimal.h>
void CBLPGM(decimal(9,7));
#pragma datamodel(p128)
#pragma linkage(CBLPGM,OS)
#pragma datamodel(pop)
int main(void)
{
    decimal(9,7) arg=12.1234567d;
    /* Call an OPM COBOL program and pass a packed */
    /* decimal argument to it. */
    CBLPGM(arg);
    printf("The COBOL program was called and passed a packed decimal value\n");
}

図8-115. パック 10 進データを渡す ILE C プログラム用の ILE C ソース

以下の例は、パック 10 進変数を ILE C プログラムに渡すための COBOL ソースを示しています。

** IDENTIFICATION DIVISION.**
** PROGRAM-ID. CBLPGM.**
* Packed decimals: This is going to be called by an ILE C *
* program to pass packed decimal data. *
** ENVIRONMENT DIVISION.**
** CONFIGURATION SECTION.**
** SOURCE-COMPUTER. IBM-I.**
** OBJECT-COMPUTER. IBM-I.**
** INPUT-OUTPUT SECTION.**
** FILE-CONTROL.**
** DATA DIVISION.**
** FILE SECTION.**
** WORKING-STORAGE SECTION.**
77 PAC-DATA PIC X(30) VALUE "PROGRAM START".
77 PACK-IN-WS PIC 99.99999999.
** LINKAGE SECTION.**
01 PACK-DATA PIC 9(2)V9(7) PACKED-DECIMAL.
** PROCEDURE DIVISION USING PACK-DATA.**
** MAIN-LINE SECTION.**
MOVE PACK-DATA TO PACK-IN-WS.
DISPLAY "**** PACKED DECIMAL RECEIVED IS: " PACK-IN-WS.
GOBACK.

図8-116. パック 10 進データを ILE C プログラムから受け取る COBOL ソース

出力は以下のようになります。

**** PACKED DECIMAL RECEIVED IS: 12.1234567
実行キーを押して端末セッションを終了してください。

The COBOL program was called and passed a packed decimal value.
実行キーを押して端末セッションを終了してください。

パック 10 進データ型のライブラリー関数の使用

va_arg マクロを使用して、(n,p) という形式のパック 10 進データを受け入れることができます。パック 10 進定数をファイルに書き込み、それらをスキャンして戻すことが可能です。

次の図は、va_arg マクロを使用して、decimal(n,p) という形式のパック 10 進データを受け入れる方法を示しています。この va_arg マクロは、現行のパック 10 進引数を返します。
This program uses the va_arg macro to accept a static decimal data type of the form decimal(n,p). The va_arg macro returns the current packed decimal argument.

#include <decimal.h>
#include <stdio.h>
#include <stdarg.h>
define N1 3
define N2 6
int vargf(FILE *, char *, ...);
int main(void)
{
  int num1 = -1, num2 = -1;
  char fmt_1[] = "%d(3,2)%d(3,2)%d(3,2)";
  char fmt_2[] = "%d(3,2)%d(3,2)%d(3,2)%d(3,2)%d(3,2)%d(3,2)";
  decimal(3,2) arr_1[] = { 1.11d, -2.22d, 3.33d };
  decimal(3,2) arr_2[] = { -1.11d, 2.22d, -3.33d, 4.44d, -5.55d, 6.66d };
  FILE *stream_1;
  FILE *stream_2;
  stream_1 = fopen("file_1.dat", "wb+");
  num1 = vargf(stream_1, fmt_1, arr_1[0],
               arr_1[1],
               arr_1[2]);
  if (num1 < 0)
    {
      printf("An error occurred when calling function vargf first time\n");
    }
  else
    {
      printf("Number of char. printed when vargf is called first time is:%d\n", num1);
    }
  stream_2 = fopen("file_2.dat", "wb+");
  num2 = vargf(stream_2, fmt_2, arr_2[0],
               arr_2[1],
               arr_2[2],
               arr_2[3],
               arr_2[4],
               arr_2[5]);
  if (num2 < 0)
    {
      printf("An error occurred when calling function vargf second time\n");
    }
  else
    {
      printf("Number of char. printed when vargf is called a second time is:%d\n", num2);
    }
  fclose(stream_1);
  fclose(stream_2);
}
int vargf(FILE *str, char *fmt, ...)
{
  int result;
  va_list arg_ptr;
  va_start(arg_ptr, fmt);
  result = vfprintf(str, fmt, arg_ptr);
  va_end(arg_ptr);
  return result;
}

出力は以下のようになります。
Number of char. printed when vargf is called first time is: 13
Number of char. printed when vargf is called second time is: 27

以下の例では、パック 10 進定数をファイルに書き込む方法、およびそれをスキャンして戻す方法を説明しています。また、この例では、パック 10 進配列を配列に渡す方法も示しています。

```c
/* This program shows how to write packed decimal constants to a file */
/* and scan them back again. Also shows how to pass a packed decimal */
/* array to a function. */
#include <decimal.h>
#include <stdio.h>
#include <stdlib.h>
#define N 3 /* Array size */
/* for decimal declaration. */
FILE *stream; /* File pointer declaration. */
/* Declare valid packed decimal */
/* array. */
decimal(4,2) arr_1[] = {12.35d, 25.00d, -19.58d};
decimal(4,2) arr_2[N];
void write_num(decimal(4,2) a[N]); /*Declare function to */
/*write to a file. */
void read_num(decimal(4,2) b[N]); /*Declare function to */
/*read from a file. */

int main(void)
{
    int reposition=0;
    /* Open the file. */
    if ((stream = fopen("*CURLIB/OUTFILE","w+")) == NULL)
    {
        printf("Can not open file");
        exit(EXIT_FAILURE);
    }
    write_num(arr_1); /* Call function to write */
    /* values of packed decimal */
    /* array to outfile with fprintf*/
    /* library function. */
    reposition=fseek(stream, 0L, SEEK_SET);
    if (reposition!=0)
    {
        printf("FSEEK failed to position file pointer\n");
        exit(EXIT_FAILURE);
    }
    read_num(arr_2); /* Call function to read */
    /* values of packed decimal */
    /* array from file using */
    /* fscanf() function. */
    fclose(stream); /* Close the file. */
}
/* write_num is passed a packed decimal array. These values are */
/* written to a text file with the fprintf library function. */
/* If the function is successful a 0 is returned, otherwise a */
/* negative value is returned (indicating an error). */
void write_num(decimal(4,2) a[N])
{
    int i, j;
    for (i=0; i < N; i++)
    {
        j = fprintf(stream,"%D(4,2)\n",a[i]);
        if (j < 0)
            printf("Number not written to file %D(4,2)\n",a[i]);
    }
```
/* read_num is passed a packed decimal array. The values are */
/* read from a text file with the fscanf library function. */
/* If the function is successful a 0 is returned, otherwise a */
/* negative value is returned (indicating an error). */

void read_num(decimal(4,2) b[N])
{
    int i, j;
    for (i=0; i < sizeof(b)/sizeof(b[0]); i++)
    {
        j = fscanf(stream,"%D(4,2)\n",&b[i]);
        if (j < 0)
            printf("Error when reading from file\n");
        printf("b[%d]=%D(4,2)\n", b[i], b[i], b[i], b[i]);
    }
}

出力は以下のようにになります。

| b[0] | 12.35 |
| b[1] | 25.00 |

以下の例は、%D(*,*) 指定子を printf() 関数で使用する方法を示しています。出力される変数の n および p が変換指定子 %D(n,p) の n および p と一致しない場合、振る舞いは未定義のものになります。パック 10 進式の結果タイプのサイズが不明の場合は必ず、引数リスト内の単項演算子 digitsof(式) および precisionof(式) を使用して、D(*,*) 内の * を置き換えてください。

#include <decimal.h>
#include <stdio.h>
int main(void)
{
    decimal(6,2) op_1=1234.12d;
    decimal(10,2) op_2=-12345678.12d;
    printf("op_1 = %*.*D(*,*)\n", 6, 2, digitsof(op_1),
            precisionof(op_1), op_1);
    printf("op_2 = %*.*D(*,*)\n", 10, 2, digitsof(op_2),
            precisionof(op_2), op_2);
}

図 8-117. パック 10 進数を出力するための ILE C ソース

出力は以下のようにになります。

| op_1 | 1234.12 |
| op_2 | -12345678.12 |

実行キーを押して端末セッションを終了してください。

パック 10 進数データ・タイプ・エラーの理解

パック 10 進数データ・エラーのすべての警告およびメッセージは、実行時に発生と明示的に記述されていない限り、ILE C コンパイラによってコンパイル時に発行されます。オーバーフローについてコンパイル時に警告を受け取った場合、例外が実行時に生成されることがあります (SIGFPE が発生します)。

実行時にエラーは、以下のパック 10 進数操作時に発生します。
包括
• キャスト
• 初期化
• 算術演算
• 関数呼び出し

注: 一部のオーバーフロー状態は、コンパイル時に警告によって通知されます (桁落ちが発生する可能性があります)。

パック 10 進数の警告およびエラー状態

以下の図は、パック 10 進数式について ILE C コンパイラによって出されるすべての警告およびエラー状態を示しています。

```c
#include <decimal.h>

decimal (999,99) s1 = 1234.56d; /* Generates a severe error because */
    /* the decimal size is greater than 63. */

decimal (10, 64) s2 = 1234.56d; /* Generates a severe error because */
    /* the precision size exceeds 63. */

decimal (1.2, 3) s3 = 1.2d; /* Generates a severe error because */
    /* the decimal size is an invalid number. */

decimal (2,1.2) s4 = 1.3d; /* Generates a severe error because the */
    /* precision size is an invalid number. */

decimal (1,3) s5 = 1.345d; /* Generates a severe error because the */
    /* precision size exceeds the decimal size. */

decimal (63,62) s6 = 1234567890123456789012345678901234567890123456789d;
    /* Generates a severe error because decimal */
    /* constant is out of range of a valid */
    /* packed decimal constants. */

decimal(10,2) s7 = 123456789012345678901234567890123456789012345678901234567890123456789012345d
    + 12345.1234567891d;
    /* s7 = (63,5) + (15,10) */
    /* Generates a severe error because of */
    /* truncation of precision values in */
    /* intermediate result. */

decimal (10,2) s8 = 123456789012345678901234567890123456789012345678901234567891d;
    /* s8 = (35,5) + (28,2) */
    /* Generates a severe error because of */
    /* truncation of precision values in */
    /* intermediate result. */

decimal (10,2) s9 = 123456789012345678901234567890123456789012345678901234567890123456789012345d
    / 1234567890.123456789012345678901234567891d;
    /* Generates a warning for possible loss of */
    /* whole-digit data in intermediate value. */
    /* Generates a severe error for loss of */
    /* whole-digit data loss in result. */
```

図 8-118. パック 10 進数の警告およびエラー状態

注:
• パック 10 進数を収めるのに大きすぎるターゲット・フィールドへの代入の場合、ILE C コンパイラでは、静的、外部、または自動の初期化については警告またはエラーを発行しません。
実行時オーバーフロー例外の抑制

```c
#include <decimal.h>
#pragma nosigtrunc

void f(decimal(4,2) a)
{
    int main(void)
    {
        decimal(8,4) arg=1234.1234d;
        decimal(5,2) op_1=1234567.1234567d; /* Overflow in initialization. */
        decimal(2) op_2;
        decimal(20,5) op_3=12.34d;
        decimal(15,2) op_4=1234567890.12d;
        decimal(6,2) op_5=1234.12d, cast;
        decimal(31,2) res;
        cast=(decimal(2))op_5; /* Overflow in casting. */
        op_2=arg; /* Overflow in assignment. */
        f(arg); /* Overflow in function call. */
        res=op_3*op_4; /* Overflow in arithmetic */
        /* operation. */
    }
    }
}
```

図 8-119. 実行時例外を抑制するための ILE C ソース

注: 実行時例外はジョブ・ログに記録されません。
### C++ プログラムでのパック 10 進数データの使用

パック 10 進数データ・タイプ表現には、整数部分と小数部分が含まれています。ILE C++ では、最大のパック 10 進数表現は 63 枠です。

注: IBM i 7.1 以前では、最大のパック 10 進数表現は 31 枠です。

ILE C++ コンパイラでは、bcd クラスのパック 10 進数データ・タイプがサポートされます。ヘッダー・ファイルは <bcd.h> です。

注: ILE C コンパイラでは、ISO C の拡張としてパック 10 進数データ・タイプがサポートされます。詳しくは、8-126 ページの『C プログラムでのパック 10 進数データの使用』を参照してください。

パック 10 進数データ・タイプを使用して、以下を行うことができます。

- 特に金融計算におけるビジネスおよび商用アプリケーションで、大きな数値を正確に表す。

注: 浮動小数点数演算を使用する必要はありません。浮動小数点は、多くの場合、以下のような数値を使用する科学および工学での計算により適しています。
- 最大のパック 10 進数数が保管できるような数値が大きい
- 最小のパック 10 進数数が小さいが、商用的に十分な精度がない
- パック 10 進数データを持つタイプ定義、配列、構造体、および共用体を宣言する。パック 10 進数データに対して単項演算子を適用できます。パック 10 進数データには、ビット単位演算は適用されません。パック 10 進数データ・タイプについて詳しくは、「ILE C/C++ 解説書」を参照してください。

このトピックでは以下について説明します。

- **IBM i 2 進化 10 進数 (BCD) ヘッダー・ファイル**
  - _DecimalT クラス・テンプレートの使用
  - 他のデータ・タイプへの変換
  - _DecimalT クラス・テンプレートでの C++ 例外処理
  - 関数への _DecimalT クラス・テンプレート・オブジェクトの受け渡し
  - _DecimalT クラス・テンプレート・オブジェクトへのポインタの受け渡し
  - _DecimalT クラス・テンプレートが含まれた別のプログラムの呼び出し
  - ファイルへの _DecimalT クラス・テンプレートの内容の書き込み

### IBM i 2 進化 10 進数 (BCD) ヘッダー・ファイル

C++ _DecimalT クラス・テンプレートのクラス・テンプレート定義と関数テンプレート定義および _DecimalT クラス・テンプレートの数値限度は、ヘッダー・ファイル <bcd.h> 内に定義されています。

8-140 ページの表 8-14 を参照してください。

_DecimalT クラス・テンプレートを使用する C++ ソース・ファイルは、bcd.h ヘッダー・ファイルを組み込む必要があります。#include <bcd.h> ステートメントは、_DecimalT クラス・テンプレートの使用より前に記述する必要があります。
### _DecimalT クラス・テンプレートの使用

ILE C++ で作業している場合、_DecimalT クラス・テンプレートにより、整数部分と小数部分を含めて最大 63 術の有効桁を表すことができます。小数点に続く 2 術によって、ドルの小数部分を正確に表すことができます。

**注:** IBM i 7.1 以前では、最大のパック 10 進数表現は 31 術です。

**注:** 浮動小数点数演算を使用する必要はありません。浮動小数点は、多くの場合、のような数値を使用する科学および工学での計算により適しています。

- 最大のパック 10 進数変数が保管できるよりもはるかに大きい
- 最小のパック 10 進数よりもはるかに小さいが、商用に十分な精度がない

浮動などの他のデータ・タイプで使用するのと同じ宣言および演算子を _DecimalT クラス・テンプレートに適用することができます。ただし、共用体およびビット単位演算子は例外で、_DecimalT クラス・テンプレートには適用されません。

以下のことが可能です。
- _DecimalT クラス・テンプレートを持つタイプ定義、配列、および構造体を宣言する。
- _DecimalT クラス・テンプレートで算術、比較、代入、コンマ、条件、等価、論理、および単項の各演算子を適用する。
- 関数呼び出しで _DecimalT クラス・テンプレートを渡す。_DecimalT クラス・テンプレートは、ILE 言語のパック 10 進数表現と互換性があります。
- マクロを定義し、_DecimalT クラス・テンプレートを指定してライブラリ関数を呼び出す。
- ILE システム・デバッガーの使用時に _DecimalT クラス・テンプレートを表示する。

---

<table>
<thead>
<tr>
<th>定数名</th>
<th>説明</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC_DIG</td>
<td>IBM i 7.1 以前で _DecimalT クラス・テンプレートが保持できる最大有効桁数。</td>
</tr>
<tr>
<td>DEC_PRECISION</td>
<td>IBM i 7.1 以前で _DecimalT クラス・テンプレートが保持できる小数点以下の最大桁数。</td>
</tr>
<tr>
<td>DEC63_DIG</td>
<td>IBM i 7.2 で _DecimalT クラス・テンプレートが保持できる最大有効桁数。</td>
</tr>
<tr>
<td>DEC64_PRECISION</td>
<td>IBM i 7.2 で _DecimalT クラス・テンプレートが保持できる小数点以下の最大桁数。</td>
</tr>
<tr>
<td>DFT_DEC_DIG</td>
<td>ターゲットが IBM i 7.1 以前の場合は DEC_DIG に設定し、それ以外の場合は DEC63_DIG に設定します。</td>
</tr>
<tr>
<td>DFT_DEC_PRECISION</td>
<td>ターゲットが IBM i 7.1 以前の場合は DEC_PRECISION に設定し、それ以外の場合は DEC63_PRECISION に設定します。</td>
</tr>
<tr>
<td>DEC_INT_DIG</td>
<td>整数タイプへの変換時における 2 進化 10 進数オブジェクトの有効桁数。値 10 が DEC_INT_DIG に保管されます。</td>
</tr>
<tr>
<td>DEC_INT_PREC</td>
<td>整数タイプからの変換時における 2 進化 10 進数オブジェクトの小数点以下の桁数。値 0 が DEC_INT_PREC に保管されます。</td>
</tr>
</tbody>
</table>

---

**表 8-14. bcd.h で定義されている定数**
_DecimalT クラス・テンプレートを表示したときに、デバッガーから演算子にアクセスすることはできません。ILE システム・デバッガーについては、5.1 ページの『プログラムのデバッグ』を参照してください。

_DecimalT クラス・テンプレート・オブジェクトの宣言
オブジェクトを _DecimalT クラス・テンプレートとして宣言するには、以下のようになります。

_DecimalT<10,2> x;
_DecimalT<5,0> z;
_DecimalT<18,10> *ptr;
_DecimalT<8,2> arr[100];

注:
1. 変数 x には __D("-99999999.99") から __D("+99999999.99") の値を使用できます。
2. 変数 z には __D("-99999") から __D("+99999") の値を使用できます。
3. ptr は、_DecimalT<18,10> という型のオブジェクトを指すポインターです。
4. arr は、100 個のエレメントの配列で、各エレメントは _DecimalT<8,2> というクラスです。
5. 先行ゼロは、_DecimalT クラス・テンプレートのサイズを指数で示します。_DecimalT クラス・テンプレートに先行ゼロを入力する必要はありません。

_D マクロを使用したコードの単純化
_ConvertDecimal コンストラクターを頻繁に使用する必要があるコードを単純化するには、_D マクロを使用します。

_ConvertDecimal(char *);

_ConvertDecimal オブジェクトを使用して、_DecimalT クラス・テンプレートを初期化することができま

_DecimalT<5,2> x = _D("123.45");
_DecimalT<DEC_DIG, DEC_PRECISION> a = _D(".0000000000000000000000000000005");
_DecimalT<6,2> b[] ={_D("1.2"), _D("2"), _D("123.56"), _D("-3.3");
_DecimalT<20,20> a = _D("-12.123456789");

注:
ストリングに使用できるのは以下の文字だけです。
0 1 2 3 4 5 6 7 8 9 . - +

_DecimalT クラス・テンプレートの入出力
_DecimalT オブジェクトの入出力には、標準 (拡張子なし) ヘッダーと USL (.h 拡張子) ヘッダーの両方を使用できます。C <stdio> 定義入出力関数を使用するには、C 言語の内部定義 10 進数型に使用される %D(n,p) 型フォーマットを指定します。
.h 拡張子を持つヘッダー・ファイルを使用していると想定します。
・ _DecimalT クラス・テンプレート値を印刷するために、cout 定義済みストリームを使用します。
・ _DecimalT クラス・テンプレート値を読み取るために、cin 定義済みストリームを使用します。

注: 拡張子のないヘッダー・ファイル (<stdio> など) を使用している場合は、stdout および stdin 定義済みストリームを使用します。
_DecimalT クラス・テンプレートの値を印刷するには、fprintf()、printf()、sprintf()、vprintf()、または vsprintf() 関数を使用できます。

_DecimalT クラス・テンプレートの値を読み取るには、fscanf()、scanf()、または sscanf() 関数を使用できます。

_DecimalT クラス・テンプレートでの演算子の使用

_DecimalT クラス・テンプレートでは、以下の演算子を使用できます。
- 算術 『DecimalT クラス・テンプレートを使用した算術演算子の使用』を参照
- 代入
- コンマ
- 条件 (8-144 ページの『DecimalT クラス・テンプレートを使用した条件式の使用』を参照)
- 等価 (8-144 ページの『DecimalT クラス・テンプレートを使用した等価演算子の使用』を参照)
- 関係 (8-143 ページの『DecimalT クラス・テンプレートでの比較演算子の使用』を参照)
- 単項 (8-145 ページの『DecimalT クラス・テンプレートでの単項演算子の使用』を参照)

注:
1. _DecimalT クラス・テンプレートの場合には論理演算子は実装されません。
2. 代入時の実行時例外については、8-146 ページの『C++ バック 10 進数データの変換』を参照してください。

_DecimalT クラス・テンプレートを使用した算術演算子の使用: 8-143 ページの図 8-120 では、下記の方法について記載してあります。
- オペランドおよび結果を定義する
- オペランドで算術演算を実行する
- 出力を標準出力ストリーム cout に送信する
```cpp
#include <bcd.h> // bcd Class Header File
#include <iostream.h>

int main()
{
  _DecimalT<10,2> op_1 = __D("12");
  _DecimalT<5,5> op_2 = __D("-.12345");
  _DecimalT<24,12> op_3 = __D("12.34");
  _DecimalT<20,5> op_4 = __D("11.01");

  _DecimalT<14,5> res_add;
  _DecimalT<25,2> res_sub;
  _DecimalT<15,7> res_mul;
  _DecimalT<31,14> res_div;

  res_add = op_1 + op_2;
  res_sub = op_3 - op_1;
  res_mul = op_2 * op_1;
  res_div = op_3 / op_4;

  cout <<"res_add =" <<res_add <<endl;
  cout <<"res_sub =" <<res_sub <<endl;
  cout <<"res_mul =" <<res_mul <<endl;
  cout <<"res_div =" <<res_div <<endl;
}
```

図8-120: _DecimalT クラス・テンプレート用の算術演算子:

出力は次のとおりです。

```
res_add =11.87655
res_sub =0.34
res_mul =-1.4814000
res_div =1.12079927338782
```

_DecimalT クラス・テンプレートでの比較演算子の使用: _DecimalT クラス・テンプレートで比較演算子を使用する際には、以下の点を考慮に入れてください。

- _DecimalT クラス・テンプレートで比較演算子 less than (<) を使用し、_DecimalT クラス・テンプレートで他の数値型 (つまり、integer, float, double, および long double) と比較できます。
- 暗黙的変換は、算術変換規則を使用して実行されます。

8-144 ページの図8-121に、_DecimalT クラス・テンプレートで比較演算子を使用する方法を示します。


```cpp
#include <bcd.h>
#include <iostream.h>

_DECIMALT<10,3> pdval = __D("0000023.423"); // bcd declaration
int ival = 1233.1; // Integer declaration
float fval = 1234.34f; // Float declaration
double dval = 251.5832; // Double declaration
long double lval = 37486.234; // Long double declaration

main( )
{
    _DecimalT<15,6> value = __D("000485860.085999");
    // Perform relational operation between other data types and
    // bcd class
    if (pdval < ival) cout <<"pdval is the smallest !"<<endl;
    if (pdval < fval) cout <<"pdval is the smallest !"<<endl;
    if (pdval < dval) cout <<"pdval is the smallest !"<<endl;
    if (pdval < lval) cout <<"pdval is the smallest !"<<endl;
    if (pdval < value) cout <<"pdval is the smallest !"<<endl;
}
```

図8-121. 比較演算子および_DECIMALクラス・テンプレート:

出力は次のとおりです。

```
pdval is the smallest !
pdval is the smallest !
pdval is the smallest !
pdval is the smallest !
pdval is the smallest !
```

_DECIMALクラス・テンプレートを使用した条件式の使用: 図8-122では、下記の方法について記載してあります。

- オペランドおよび結果を定義する
- オペランドを使用して条件式を作成する
- 結果が指定条件を満たした場合にのみ出力を標準出力ストリーム cout に送信する

```cpp
#include <bcd.h>
#include <iostream.h>
int main ()
{
    _DecimalT<10,2> x, y, z;
    x = __D("1.20");
y = __D("01.2");
z = (x==y)? __D("9.9"):__D("2.45");
if (z== __D("9.9"))
{  
cout <<"x equals y" <<endl;
    }
}
```

図8-122. 条件演算子および_DECIMALクラス・テンプレート:

出力は次のとおりです。

```
x equals y
```

_DECIMALクラス・テンプレートを使用した等価演算子の使用: 8-145ページの図8-123では、下記の方法について記載してあります。
• オペランドおよび結果を定義する
• オペランドで等価演算を実行する
• 出力を標準出力ストリーム cout に送信する

```c
#include <bcd.h>
#include <iostream.h>

.DecimalT<1, 0> op_1 = __D("+0"); // Declare and initialize
.DecimalT<1, 0> op_2 = __D("-0"); // valid BCD
.DecimalT<9,4>  op_3 = __D("00012.3400");
.DecimalT<4,2>  op_4 = __D("12.34");

int main(void) // These statements
{// perform equality <=> test
  // on the above variable
    if (op_1 == op_2) // declarations
    {
        cout <<"op_1 equals op_2"<<endl;
    }
    if (op_3 != op_4)
    {
        cout <<"op_3 not equals op_4"<<endl;
    }
    else
    {
        cout <<"op_3 equals op_4"<<endl;
    }
}
```

図8-123 例: 等価演算子および DecimalT クラス・テンプレート

出力は次のとおりです。

```
op_1 equals op_2
op_3 equals op_4
```

_DECIMALT クラス・テンプレートでの単項演算子の使用: 単項式 には、1 つのオペランドと単項演算子が含まれています。すべての単項演算子 には、同じ優先順位が付けられ、右から左の結合順序が指定されます。単項演算子の多重定義については、「ILE C/C++ 解説書」を参照してください。

8-146 ページの図 8-124 では、下記の方法について記載してあります。
• オペランドを定義する
• オペランドで単項演算を実行する
• 出力を標準ストリーム cout に送信する
C++ バック 10 進数データの変換

変換するバック 10 進数タイプの値が、正確に表現できる値の範囲内にある場合、バック 10 進数タイプの値は変更されません。

バック 10 進数値、タイプが同じ場合は互換性があります。例えば、decimal(n1, p1) と decimal(n2, p2) のタイプに互換性があるのは、以下の場合のみです。

((n1 == n2) && (p1 == p2))

ある _DecimalT クラス・テンプレートから別のクラス・テンプレートへの値の変換

別の _DecimalT クラス・テンプレートに変換する _DecimalT クラス・テンプレートの値が、正確に表現できる値の範囲内にある場合は、以下の図に示すように、変換される _DecimalT クラス・テンプレートの値は切り捨てられます。

8-124. 例: 単項演算子および _DecimalT クラス・テンプレート

結果の出力は、以下のとおりです。

```
op_1++ => 12.00
op_1 after increment => 13.00
-op_2 => 0.12345
--op_3 => 11.340000000000
+op_4 => 11.01000
```
以下の図に示すように、ある _DecimalT クラス・テンプレートを、精度が小さい別の _DecimalT クラス・テンプレートに代入すると、結果として小数部分が切り捨てられます。

```c
DecimalT<7,4> x = __D("123.4567");
DecimalT<7,1> y;
y = x; // y = __D("123.4")
```
図 8-126. ある _DecimalT クラス・テンプレートから精度が小さい別のクラス・テンプレートへの変換例

以下の図に示すように、ある _DecimalT クラス・テンプレートを、整数部分が小さい別の _DecimalT クラス・テンプレートに代入すると、結果として整数部分が切り捨てられます。

```c
DecimalT クラス・テンプレートから整数データ型への値の変換
DecimalT クラス・テンプレートの値が整数型に変換されると、以下のようになります。
• CPYNV への直接呼び出しが行われます。
• 小数が整数で表現可能な桁数を超えると、CPYNV はまずパック 10 進数値を長精度整数に変換してから、その整数の 16 進数値を切り捨てます。

注: _DecimalT クラス・テンプレートを整数型に代入し、その結果、整数部分に切り捨てが起こる場合、実行時例外は発生しません。

```c
int op;
DecimalT<7,2> op1 = __D("12345.67");
op = op1;
    // Truncation on the fractional
    // part. op=12345
```
図 8-127. 小数部分を持つ _DecimalT クラス・テンプレートから整数型への変換例

```c
int op;
DecimalT<3, 0> op2 = __D("123");
op = op2; // No truncation; op=123
```
図 8-128. 整数部分の桁数が 10 桁に満たない _DecimalT クラス・テンプレートから整数型への変換例

```c
int op2;
DecimalT<12, 0> op3;
op3 = __D("123456789012");
op2 = op3;
    // Truncation occurs on the integral
    // part 123456789012 (0x1CBE991A14).
    // op2= 3197704724 (0xBE991A14); no runtime
    // exception.
```
図 8-129. 整数部分の桁数が 10 桁を超える _DecimalT クラス・テンプレートから整数型への変換例

```c
#include <bcd.h>

int op;
DecimalT<15,2> op_1 = __D("1234567890123.12");
op = op_1;
    // Truncation occurs on the integral and
    // fractional parts 1234567890123 (0x11F71FB04CB).
    // op=1912276171 (0x71FB04CB); no run-time exception.
```
図 8-130. 整数部分の桁数が 10 桁を超え、小数部分を持つ _DecimalT クラス・テンプレートから整数型への変換例
_DecimalT クラス・テンプレートから浮動小数点データ型への値の変換
_DecimalT クラス・テンプレートのクラスを浮動小数点データ型に変換するには、以下の図に似たソースコードを使用します。

```c
#include <bcd.h>
#include <iostream.h>

int main(void)
{
   _DecimalT<5,2> dec_1=__D("123.45");
   _DecimalT<11,5> dec_2=__D("-123456.12345");
   float f1,f2;
   f1=dec_1;
   f2=dec_2;
   cout <<"f1=" <<f1 <<endl <<"f2=" <<f2 <<endl; //f1=123.45
       // f2=-123456
}
```
図8-131. _DecimalT クラス・テンプレートの浮動小数点データ型への変換例

出力を以下に示します。

```
f1=123.45
f2=123456
```

_DecimalT クラス・テンプレートのサイズの判別
_DecimalT で sizeof 演算子を使用すると、_DecimalT クラス・テンプレートが占める合計バイト数を判別できます。

注: 各 _DecimalT クラス・テンプレートの数字は、1 バイトの半分を占めます。残り半バイトは符号に使用されます。_DecimalT によって使用されるバイト数は、(n + 1)/2 よりも大きいか等しい最小の整数です (例えば、sizeof(_DecimalT) = ceil((n + 1)/2))。

```c
int y;
_DecimalT <5, 2> x;  // This would be calculated to be 3 bytes
y =sizeof(x);          // (5+1)/2 = 3.
```
図8-132. _DecimalT クラス・テンプレートが占める合計バイト数の判別例

_DecimalT クラス・テンプレートの桁数の判別
_DecimalT クラス・テンプレートでメンバー関数 DigitsOf() を使用すると、_DecimalT クラス・テンプレートの総桁数 n を判別できます。
int n,n1;
_DecimalT <5, 2> x;
   n = x.DigitsOf();  // the result is n=5

図8-133. _DecimalT クラス・テンプレートの桁数の判別例

_DecimalT クラス・テンプレートの精度の判別
以下の図に示すように、_DecimalT クラス・テンプレートでメンバ関数 PrecisionOf() を使用すると、
_DecimalT クラス・テンプレートの小数部の桁数 p を判別できます。

int p,p1;
_DecimalT <5, 2> x;
P=x.PrecisionOf();  // The result is p=2

図8-134. _DecimalT クラス・テンプレートの小数桁数の判別例

オーバーフローの処理方法
表8-15 では、_DecimalT クラス・テンプレートが、より小さなターゲットに代入された場合のオーバーフローの動作について説明しています。

_DecimalT クラス・テンプレートが、より小さなターゲットの _Decimal クラス・テンプレートに代入されると、例外が生成されます。

以下の場合は、例外は生成されません。
• _DecimalT クラス・テンプレートが、より小さなターゲットの char、int、short、long、または bit
  フィールドに代入された場合
• _DecimalT クラス・テンプレートが、より小さなターゲットの浮動小数点に代入された場合

表8-15. _DecimalT クラス・テンプレートからより小さなターゲットへのオーバーフローの処理

<table>
<thead>
<tr>
<th>元のタイプ</th>
<th>対象のタイプ</th>
<th>実行時例外</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DecimalT クラス・テンプレート</td>
<td>char、int、short、long、bit</td>
<td>いいえ</td>
</tr>
<tr>
<td>_DecimalT クラス・テンプレート</td>
<td>_DecimalT クラス・テンプレート</td>
<td>いいえ</td>
</tr>
<tr>
<td>_DecimalT クラス・テンプレート</td>
<td>float</td>
<td>いいえ1</td>
</tr>
</tbody>
</table>

注: 1. _DecimalT クラス・テンプレートを double に代入した場合、オーバーフローを引き起こすために十分に大きな
_DecimalT クラス・テンプレートはありません。

_DecimalT テンプレートでの C++ 例外処理の使用
このセクションでは、_DecimalT template クラス・テンプレートで C++ 例外処理を使用している場合の
_DecimalT template クラス、エラー・クラス、およびデバッグ・マクロの実行時例外について説明します。実行時にエラーが検出されると、エラー・オブジェクトがスローされます。

このセクションでは、以下について説明します。
• _DecimalT クラス・テンプレートの実行時例外
• _DecErr クラスから派生されたクラスの使用
• デバッグ・マクロの使用
DecimalT クラス・テンプレートの実行時例外

ILE C では、バック 10 数は、ネイティブ・データ・タイプとして実装されます。このアプローチにより、有効ではない 10 数形式などのエラーをコンパイル時に検出できます。ILE C++ では、同様のエラーの検出は、実行時まで握え置かれます。

```cpp
#include <bcd.h>
void main()
{
  _DecimalT<10,20> b = __D("ABC"); // Runtime exception is raised
}
```

```cpp
#include <bcd.h>
void main()
{
  _DecimalT<65,2> a; // Max. dig. allow is 63. Again,
                   // runtime exception is raised
}
```

図 8-135 _DecimalT クラス・テンプレートの実行時例外

実行時例外の発生条件：実行時例外は、以下の操作で発生する場合があります。

- 算術演算
- 代入
- キャスト
- 関数呼び出し
- 初期化

注：コンパイル時に発生するオーバーフロー状態は、実行時まで握え置かれます。このため桁の消失が起こる可能性があります。

DecimalT クラス・テンプレートについてコンバイラーによって発行される実行時例外：DecimalT クラス・テンプレートについてコンバイラーによって発行される実行時例外は、以下の図に示しています。
#include <bcd.h>
static _DecimalT<5,2>  s1 = __D("12345678.0");
static _DecimalT<10,2> s2 = __D("1234567891234567891234567.12345")
    + __D("12345.1234567891");
    // s2 = (31,5) + (15,10)
static _DecimalT<10,2> s3 = __D("1234567891234562345")
    * __D("1234567891234.12");
    // s3 = (19,0) * (15,2)
static _DecimalT<10,2> s4 = __D("1234567891234567891234567.12345")
    / __D("12345.123456789123456");
    // s4 = (20,0) / (20,15)

int main(void)
{
    _DecimalT<5,2> a1 = __D("12345678.0");
    _DecimalT<10,2> a2, a3, a4;
    _DecimalT<5,2> a5 = (_DecimalT<5,2>) __D("123456.78");

    a2 = __D("1234567891234567891234567.12345") + __D("12345.1234567891");
    // a2 = (31,5) + (15,10)
    a3 = __D("1234567891234562345") * __D("1234567891234.12");
    // a3 = (20,5) * (15,2)
    // expression.
    // Note: Need (35,7) but
    // use (31,2), for example,
    // keep the integral part.
    a4 = __D("1234567891234567891234567.12345") / __D("12345.123456789123456");
    // a4 = (20,0) / (20,15)
    // Note: Need 35 digits to
    // calculate integral part
    // and the result becomes
    // (31,0).
}

図8-136. _DecimalT クラス・テンプレートについてコンバイラーによって発行される実行時例外

注:
1. _DecimalT クラス・テンプレート・オブジェクトを取扱うのに必要なターゲット・フィールドへの
   代入の場合、静的、外部、または自動の初期化によって発行される実行時例外があります。
2. 小数点位置合わせを必要とする式（すなわち加算、減算、または比較）の場合、位置合わせ時に許容最
   大桁数を超えると、静的、外部、および自動の初期化によって実行時例外が発生します。
3. 乗算の場合、式の評価の結果が許容最大桁数（DEC_DIG）を超える値になると、以下のようになります。
   • 静的または外部の初期化の場合、あるいは式が関数内にある場合、実行時例外が発行されます。
   • 整数部分が可能な限り保持されるように、小数部分で切り捨てが起こります。

C++ _DecimalT クラス・テンプレート例外ハンドラーの定義
実行時にエラーが発生した場合、エラー・クラスからインスタンス化されたオブジェクトがスローされます。独自の例外ハンドラーを定義して、例外を catch できます。

以下の図で、try-catch-throw を使用して _DecimalT クラス・テンプレート例外を処理する方法を示します。

```cpp
#include <bcd.h>
#include <iostream.h>

void main() {
    try {
        _DecimalT<10,2> = __D("AAA");
    } catch (DecErrInvalidConst Err) {
        cout << "Invalid Decimal constant!" << endl;
    }
}
```

図 8-137. C++ Try Catch Throw フィーチャーを使用して _DecimalT クラス・テンプレート例外を処理する例

_DecimalT クラス・テンプレート用のデバッグ・マクロの使用
_DecimalT クラス・テンプレートでは、C++ 例外処理が使用されます。実行時にエラーが検出されると、エラー・オブジェクトがスローされます。

表 8-16 に、アサーション検査をオンまたはオフにするために使用できる 3 つのマクロを定義します。

表 8-16. _DecimalT クラス・テンプレート用のデバッグ・マクロ

<table>
<thead>
<tr>
<th>マクロ</th>
<th>意味</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DEBUG</td>
<td>アサーション検査はオンになります。</td>
</tr>
<tr>
<td>_DEBUG_DECIMAL</td>
<td>_DecimalT クラス・テンプレートに対してのみアサーション検査がオンになります。</td>
</tr>
<tr>
<td>_NODEBUG_DECIMAL</td>
<td>デフォルト: アサーション検査はオフになります。このマクロは、_DEBUG マクロおよび _DEBUG_DECIMAL マクロをオーバーライドできます。</td>
</tr>
</tbody>
</table>

注:
1. アサーション検査がオンになっている場合、_DecimalT クラス・テンプレートの _DecimalT テンプレート・クラス定数およびパラメータが検証されます。
2. ゼロ除算、オーバーフロー、および _DecimalT テンプレート・クラスの桁の切り捨ての検査は、C++ ランタイム・ライブラリー内にハードコーディングされているため、_NODEBUG_DECIMAL マクロでオフにする事はできません。
3. _DEBUG マクロを使用して、_DecimalT クラス・テンプレートに対してアサーション検査をオンにすることができます。ソースで _DEBUG マクロを既に使用している場合は、_DEBUG_DECIMAL マクロを使用して、_DecimalT クラス・テンプレートに対してのみアサーション検査をオンにすることができます。
_DecimalT クラス・テンプレートのエラー検査の有効化または無効化: _DecimalT クラス・テンプレート内でエラー検査を有効にするには、以下の図に示すように、コンパイラの呼び出し時に、DEFINE(_DEBUG_DECIMAL) オプションまたは DEFINE(_DEBUG) オプションのいずれかを追加して、デバグ・マクロをオンにすることができます。

CRTCPPMOD DEFINE(_DEBUG)
CRTBNDCPP DEFINE(_DEBUG)
CRTCPPMOD DEFINE(_DEBUG_DECIMAL)
CRTBNDCPP DEFINE(_DEBUG_DECIMAL)

図 8-138. コンパイル時に _DecimalT クラス・テンプレートでエラー検査を有効にするコマンド

注:
1. DEFINE(_DEBUG) 呼び出しと DEFINE(_DEBUG_DECIMAL) 呼び出しの間の相違は、エラー検査の制御のため、その他のクラスで _DEBUG マクロが使用されるか否かにあります。別のクラスで _DEBUG マクロが使用される場合、DEFINE(_DEBUG) は、_DEBUG マクロを使用するすべてのクラスに影響し、DEFINE(_DEBUG_DECIMAL) は、_DecimalT クラス・テンプレートのみに影響します。

2. _DEBUG マクロを使用するファイルのグループにアサーション検査をオンにした場合、_NODEBUG_DECIMAL マクロを使用すると、_DEBUG マクロがオーバーライドされ、_DecimalT クラス・テンプレートのアサーション検査はオフになります。

_DecimalT クラス・テンプレート内でエラー検査を無効にするには、以下の図に示したコマンドを使用できます。

CRTCPPMOD DEFINE(_NODEBUG_DECIMAL)
CRTBNDCPP DEFINE(_NODEBUG_DECIMAL)

図 8-139. コンパイル時に _DecimalT クラス・テンプレートでのエラー検査を無効にするコマンド

注:
1. すべてのクラスにエラー検査を有効にすることができます。ただし、_DecimalT クラス・テンプレートは例外で、以下のいずれかのコマンドを使用して、_DEBUG を制御マクロとして使用します。

CRTCPPMOD DEFINE(_DEBUG _NODEBUG_DECIMAL)
CRTBNDCPP DEFINE(_DEBUG _NODEBUG_DECIMAL)

2. 同じ呼び出しで _DEBUG_DECIMAL マクロと _NODEBUG_DECIMAL マクロの両方のマクロを使用しようとするとき、_NODEBUG_DECIMAL マクロが優先され、_DecimalT クラス・テンプレートのエラー検査は無効になります。

関数への _DecimalT クラス・テンプレート・オブジェクトの受け渡し
呼び出される関数にプロトタイプが含まれていない場合、_DecimalT クラス・テンプレートを持つ引数には、デフォルト引数拡張はありません。_DecimalT クラス・テンプレート引数が含まれている関数定義は、プロトタイプ化する必要があります。そうしないと、コンパイラでエラーが出されます。

次の図に例を示します。
#include <bcd.h>
#include <iostream.h>

_DecimalT<3,1> d1 = _D("33.3");
_DecimalT<10,5> d2 = _D("55555.55555");
_DecimalT<28,0> d3 = _D("88888888888888888888888888888");

void func1( _DecimalT<3,1>, _DecimalT<10,5>,
_DecimalT<10,5>, _DecimalT<28,0>);

int main(void)
{
  func1(d1, d2, _D("999.99"), d3);
}

// func1 is prototyped
void func1( _DecimalT<3,1> x1, _DecimalT<10,5> x2,
_DecimalT<10,5> x3, _DecimalT<28,0> x4) {
    // no runtime error when referencing x1, x2, x3 or x4
    cout <<"x1 = " <<x1 <<endl;
    cout <<"x2 = " <<x2 <<endl;
    cout <<"x3 = " <<x3 <<endl;
    cout <<"x4 = " <<x4 <<endl;
}

図8-140  _DecimalT クラス・テンプレート・オブジェクトを関数に渡す例

出力は次のとおりです。

\[
\begin{align*}
x1 &= 33.3 \\
x2 &= 55555.55555 \\
x3 &= 999.99000 \\
x4 &= 88888888888888888888888888888
\end{align*}
\]

_DecimalT クラス・テンプレート・オブジェクトへのポインターの受け渡し

_DecimalT クラス・テンプレート・オブジェクトへのポインターを渡す際に、関数呼び出しの _DecimalT テンプレート・クラス引数は、関数プロトタイプ内の _DecimalT クラス・テンプレートと同じクラス・テンプレートでなければなりません。

_DecimalT クラス・テンプレート引数を指定した関数呼び出しでオーバーフローが発生した場合は、実行時例外が生成されます。

以下の図で、_DecimalT クラス・テンプレート・オブジェクトへのポインターを渡す例を示します。
// This program shows how to pass a pointer to a _DecimalT template
// class object to a function.

#include <bcd.h>
#include <iostream.h>

_Double<5,2> var=_D("123.45");
_Double<5,2> *p=&var;
_Double<5,2> *func_1(_DecimalT<5,2> *);

int main(void)
{
    // Call function with pointer to _DecimalT class template argument.
    // The value that it returns is a pointer to a _DecimalT
    // class template object.
    _Double<5,2> *tempvar;
    tempvar = func_1(p);
    if(tempvar == 0)
    {
        cout <<"Function call not successful" <<endl <<endl;
    } else
    {
        cout <<"The bcd value is " << *tempvar <<endl <<endl;
    }
    _Double<5,2> *func_1(_DoubleT<5,2> *q)
    { return q; }
}

図8-141. _DecimalT クラス・テンプレート・オブジェクトへのポインタを渡す例

出力は次のとおりです。

The bcd value is 123.45

_DecimalT クラス・テンプレートが含まれた別のプログラムの呼び出し

言語間呼び出しを行い、_DecimalT クラス引数を ILE RPG、ILE COBOL、および ILE C プログラムに渡すことができます。

以下の図に、ILE COBOL の例を示します。
C++ program shown below.

```cpp
// This program calls an ILE COBOL program
// and passes a bcd object.

#include <iostream.h>
#include <bcd.h>

extern "OS" void CBLPGM(_DecimalT<9,7>);

int main(void)
{
    _DecimalT<9,7> arg=__D("12.1234567");
    // Call an ILE COBOL program and pass a bcd object
    // to it.
    CBLPGM(arg);
    cout <<"The COBOL program was called and passed a bcd object"<endl;
}
```

ILE COBOL program is shown below.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBLPGM.

******************************************************************************
* Packed decimals: This is going to be called by a C++                        *
* program to pass packed decimal data.                                    *
******************************************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-I.
OBJECT-COMPUTER. IBM-I.

INPUT-OUTPUT SECTION.
FILE-CONTROL.

DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.
77 PAC-DATA PIC X(30) VALUE "PROGRAM START".
77 PACK-IN-WS PIC 99.9999999.

LINKAGE SECTION.
01 PACK-DATA PIC 9(2)V9(7) PACKED-DECIMAL.

PROCEDURE DIVISION USING PACK-DATA.

MAIN-LINE SECTION.
MOVE PACK-DATA TO PACK-IN-WS.
DISPLAY "**** BCD OBJECT RECEIVED IS: " PACK-IN-WS.
GOBACK.
```

**8-142. ILE C++** プログラムから **ILE COBOL** プログラムを呼び出し、_DecimalT クラス・テンプレートを渡す例

出力は次のとおりです。

```
**** BCD OBJECT RECEIVED IS: 12.1234567
The COBOL program was called and passed a bcd object
```
This program shows how to write _DecimalT class template constants to a file and scan them back again. Shows how to pass a _DecimalT class template array to a function.

```c
#include <bcd.h>
#include <iostream.h>
#include <stdlib.h>

#define N 3 // Array size for decimal declaration.

FILE *stream; // File pointer declaration.

__D("12.35")
__D("25.00")
__D("-19.58");

void write_num(_DecimalT<4,2> a[N]); // Declare function to write to a file.
void read_num(_DecimalT<4,2> b[N]); // Declare function to read from a file.

int main(void)
{
    FILE *stream; // File pointer declaration.

    int reposition=0; // Open the file. Must use fopen() to access a physical file.

    if ((stream = fopen("*CURLIB/OUTFILE","w+")) == NULL)
    {
        cout <<"Can not open file" << endl;
        exit(EXIT_FAILURE);
    }

    write_num(arr_1); // Call function to write values of the array to outfile with fprintf().

    reposition=fseek(stream, 0L, SEEK_SET);

    if (reposition!=0)
    {
        cout <<"FSEEK failed to position file pointer" << endl;
        exit(EXIT_FAILURE);
    }

    read_num(arr_2); // Call function to read values of the array from file using fscanf().

    fclose(stream); // Close the file.
}
```

write_num is passed a the array. These values are written to a text file with fprintf(). If the function is successful a 0 is returned, otherwise a negative value is returned (indicating an error.
```c
int i, j;
for (i=0;i < N;i++)
    { 
    j = fprintf(stream, "%D(4,2)\n", a[i]);
    if (j < 0)
        cout << "Number not written to file" << a[i] << endl;
    }
// read_num is passed a the array. The values are
// read from a text file with fscanf().
// If the function is successful 0 is returned, otherwise a
// negative value is returned (indicating an error).
void read_num(_DecimalT<4,2> b[N])
{
    int i, j;
    for (i=0;i < sizeof(b)/sizeof(b[0]);i++)
    { 
    j = fscanf(stream, "%D(4,2)\n", &b[i]);
    if (j < 0)
        cout << "Error when reading from file" << endl;
    cout << "b[0]=" << b[0] << endl;
    cout << "b[1]=" << b[1] << endl;
    }
    cout << "b[0]=12.35";
    cout << "b[1]=25.00";
}
```

以下の図に示すように、ofstream クラスを使用するようにこのプログラムを書き換えることができます。

// This program shows how to write _DecimalT template
// constants to a file
// and scan them back again. Shows how to pass a _DecimalT
// class template array to a function.

```c
#include <bcd.h>
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
#define N 3 // Array size
        // for decimal declaration.

    // Declare valid
        // array.

    _DecimalT<4,2> arr_1[] = {
        _D("12.35"),
        _D("25.00"),
        _D("-19.58"),
    };

    void write_num(_DecimalT<4,2> a[N]); //Declare function to
        // write to a file.

    void read_num(_DecimalT<4,2> b[N]); //Declare function to
        //read from a file.

    int main ( void )
```
```cpp
write_num(arr_1); // Call function to write
   // values of the
   // array to outf with fprintf
   // library function.

read_num(arr_2);  // Call function to read
   // values of the
   // array from file using
   // fscanf() function.

} // write_num is passed an array. These values are
   // written to a text file with the fstream class.
   // If the function is successful a 0 is returned, otherwise a
   // negative value is returned (indicating an error).

void write_num(_DecimalT<4,2> a[N])
{
    int i;
    ofstream outf("data",ios::trunc | ios::out,
               filebuf::openprot);
    if (!outf)
    {
        cerr << "Could not open file 'data' " <<endl;
        exit (EXIT_FAILURE);
    }
    for (i=0;i<N;i++) {
        outf << a[i] << endl;
    }
    outf.close();
}

// read_num is passed an array. The values are
// read from a text file with the fstream class.
// If the function is successful a 0 is returned, otherwise a
// negative value is returned (indicating an error).

void read_num(_DecimalT<4,2> b[N])
{
    int i;
    ifstream file("data");
    if (!file)
    {
        cerr << "Could not open file 'data' " <<endl;
        exit (EXIT_FAILURE);
    }
    for (i=0;i<N;i++)
    {
        file >> b[i];
        cout << "b["<< i <<"]=" <<b[i] <<endl;
    }
    if (file.eof())
    {
        cerr << "Unexpected EOF!" <<endl;
        exit (EXIT_FAILURE);
    }
    file.close();
}

出力は次のとおりです。

b[0]=12.35
b[1]=25.00
```
**C++ プログラムでのテンプレートの使用**

C++ において、以下の関連セットの宣言と定義にテンプレートを使用できます。
- クラス
- 関数
- テンプレート・クラスの静的データ・メンバー

C++ 言語では、各種類のテンプレートの構文と意味が記述されています。各コンバイラーで、テンプレートが展開されるタイミングと頻度を制御するメカニズムが決まります。

詳しくは、「ILE C/C++ 解説書」を参照してください。「Templates」のトピックに、以下に関する説明と例があります。
- テンプレート・パラメーター
- テンプレート・インスタンス化と特殊化 (明示的と部分的の両方)
- 静的データ・メンバーとテンプレート
- クラス・テンプレートと関数テンプレート
- 関数テンプレートの多重定義
- ネストされたテンプレート

このセクションには、以下が含まれています。
- 「テンプレートのインスタンス化の管理」
- 8-161 ページの「テンプレート・インスタンス化管理のオプション」
- 8-162 ページの「ILE C++ コンパイラによるテンプレート・インスタンス化的処理方法」
- 8-163 ページの「クラス・テンプレート・インスタンス化的例」
- 8-165 ページの「デフォルト・テンプレート・インスタンス化管理オプションの使用」
- 8-166 ページの「ILE テンプレート・レジストリー・オプションの使用」
- 8-167 ページの「ILE TEMPINC オプションの使用」

**テンプレートのインスタンス化の管理**
外部インクレージ・テンプレート・インスタンスの明示的特殊化は 1 つのみ使用できます。

ILE には、テンプレートの潜在的な複数のインスタンス化を処理するための方法がいくつか用意されています。以下にその方法を示します。
- デフォルトのインスタンス化オプションを使用する。
  - 8-165 ページの「デフォルト・テンプレート・インスタンス化管理オプションの使用」および、8-165 ページの「単一のインスタンス化のためにコードを手動で構築」を参照してください。
- テンプレート・レジストリー内のすべてのインスタンス化のリポジトリを管理する。
  - 8-166 ページの「ILE テンプレート・レジストリー・オプションの使用」を参照してください。
- ILE 自動テンプレート組み込み機能を使用して、テンプレートの単一のインスタンス化を保証する。
  - 8-167 ページの「ILE TEMPINC オプションの使用」を参照してください。
テンプレート・インスタンス化管理のオプション

テンプレート・インスタンス化の最適な方法は、プログラミング環境とプログラマーの能力によって異なります。表8-17では、テンプレート・インスタンス化オプションをリストし、それぞれの利点と欠点を説明します。

注: C++ モジュールの作成 (CRTCPPMOD) コマンドの TEMPLATE(*TEMPINC) オプションと TMPLREG オプションの両方を同時に使用しようとすると、コンパイルが停止します。

表8-17. テンプレート・インスタンス化管理のオプション

<table>
<thead>
<tr>
<th>CRTCPPMOD オプション</th>
<th>利点</th>
<th>欠点</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILE C++ のデフォルト: TEMPLATE(*NONE) TMPLREG(*NONE)</td>
<td>コードおよびファイル構造の再構築は不要です。</td>
<td>以下の理由により、アプリケーション全体のコンパイルとリンクに必要な時間が著しく増大する可能性があります。</td>
</tr>
<tr>
<td></td>
<td>ネットワーク環境で、 ILE C++ TEMPINC オプションに関連して発生する可能性があるファイル識別の問題を回避します。</td>
<td>• コンバイラーが、参照するすべてのインスタンスをインスタンス化する必要がある。</td>
</tr>
<tr>
<td></td>
<td>チーム環境で、 ILE C++ TEMPINC オプションに関連するファイル共有の問題を回避します。</td>
<td>• アプリケーション全体で複数のインスタンス化が検出される可能性がある。その場合、リンカーは、1つ以外のすべてを捨てなければなりません。</td>
</tr>
<tr>
<td></td>
<td>IFS/QSYS.LIB の混合環境で、 ILE C++ TEMPINC オプションに関連するファイル・システム依存関係の問題を回避します。</td>
<td>テンプレートの実装を変更した場合、そのテンプレートを使用するすべての単位を再コンパイルする必要があります。</td>
</tr>
<tr>
<td></td>
<td></td>
<td>複数のテンプレート定義を回避するために、各テンプレート・クラス関数または静的データ・メンバに1つの定義が生成されるように、コードを手動で構築する必要があります。これを行うには、</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ILE C++ コンバイラーがテンプレートに対してどのように動作するかを把握する必要があります。</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• プログラムで必要なすべてのテンプレート・インスタンス化を認識する必要があります。</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ソース・ファイルを再編成して、新しいコンパイル単位を作成する必要がある可能性があります。</td>
</tr>
<tr>
<td>ILE C++ のテンプレート・レジストリ: CRTCPPMOD TEMPLATE(*NONE) TMPLREG(*DFT) または CRTCPPMOD TEMPLATE (*NONE) TMPLREG('path-name')</td>
<td>アプリケーションごとに1つのテンプレート・インスタンス化が保証されます。</td>
<td>テンプレート・インスタンス化が除去されるたびに、依存ファイルを手動で再コンパイルする必要がある可能性があります。</td>
</tr>
<tr>
<td></td>
<td>自動インスタンス化のためにプログラムを構築する必要はありません。</td>
<td>必要な手動介入は最小限に抑えられます。</td>
</tr>
</tbody>
</table>
### ILE C++ コンパイラによるテンプレート・インスタンス化の処理方法

プログラムでテンプレートを使用すると、ILE C++ コンパイラは、次のすべての条件に一致する各 C++ テンプレートを自動的にインスタンス化します。

<table>
<thead>
<tr>
<th>アプリケーションごとに 1 つのテンプレート・インスタンス化が証明されます。</th>
<th>以下の理由により、このオプションは、チーム・プログラミング環境で実用的でない場合があります。</th>
<th>欠点</th>
</tr>
</thead>
<tbody>
<tr>
<td>デフォルトの使用によりコンパイル時間が長くなることを回避します。</td>
<td>他のユーザーがソース・ファイルを変更している間に、コンパイラがそのソース・ファイルを更新する可能性がある。</td>
<td>以下の実装を別々のファイルに分割する必要があります。</td>
</tr>
<tr>
<td>テンプレート実装が変更されるたびに、そのテンプレートを使用するすべての単位を再コンパイルする必要があります。</td>
<td>ソース・ファイルの変更がファイル・システムに依存する可能性がある。例えば、ネットワークで使用可能なヘッダー・ファイルではなく、ローカルで使用可能なヘッダー・ファイルがインクルードされる場合など。</td>
<td></td>
</tr>
</tbody>
</table>

### 静的メンバー定義の生成

ISO 標準に従うと、テンプレート・クラスの静的メンバーはデフォルトでは弱定義です。つまり、プログラム内で弱静的メンバーが複数回定義されても、その静的メンバーは 1 回しか初期化されないということです。

一部のプログラムは、他のモジュールにリンクされていると、強静的データ・メンバーを要求します。コンパイル時にデフォルトをオーバーライドするには、CRTCPPMOD コマンドに WEAKTMPL(*NO) パラメーターを追加します。

注: WEAKTMPL オプションについて詳しくは、「ILE C/C++ コンパイラ参照」を参照してください。

### 内部結合

個別にコンパイルされるいくつかの単位からプログラムが構成される場合、2 つ以上のコンパイル単位で指定のテンプレートが拡張される可能性があります。クラス、インライン関数、または静的非メンバー関数を定義するテンプレートの場合、これは必要な振る舞いです。これらのテンプレートは、それが使用される各コンパイル単位で定義する必要があります。
外部結合
外部リンケージを持つ他の関数および静的データ・メンバーの場合、複数のコンパイル単位で定義すると、プログラムのパンド時に、通常はエラーとなります。ILE C++ では、こうしたオブジェクトのテンプレート生成バージョンに特殊な扱いを行うことでこの問題が回避されます。パンド時、ILE C++ は、テンプレート生成のすべての関数および静的メンバー定義、それに加えて明示的特殊化を収集し、それらに対するすべての参照を解決します。

• 関数または静的メンバーの明示的特殊化が存在する場合、すべての参照にそれが使用されます。その関数または静的メンバーのテンプレート生成定義は、すべて破棄されます。
• 明示的特殊化が存在しない場合、テンプレート生成定義の 1 つがすべての参照に使用されます。その関数または静的メンバーの他のテンプレート生成定義は、すべて破棄されます。

注: 関数または静的メンバーに複数のテンプレート生成定義が存在すると、モジュールが大きくなり、しかもコンパイル時間が長くなります。テンプレートの重複したコードはパンド時に除去されるので、実行可能プログラムが大きくなりません。

クラス・テンプレート・インスタンス化の例
次の例では、クラス・テンプレート Stack は項目のスタックを実装します。多重定義した演算子 << と >> を使用して、スタックへの項目のプッシュと、スタックからの項目のポップを行います。

```cpp
template <class Item, int size> class Stack {
  public:
    int operator << (Item item); // push operator
    int operator >> (Item& item); // pop operator
    Stack() { top = 0; }  // constructor defined inline
  private:
    Item stack[size];  // stack of items
    int top;          // index to top of stack
};
```

図 8-143. クラス・テンプレート・インスタンス化の例

宣言と定義
図 8-143 では、Stack クラス・テンプレートの宣言はファイル stack.h に含まれ、関数定義はファイル stack.c に含まれます。

• Stack クラス・テンプレートを使ってデフォルトのインスタンス化管理オプションまたはテンプレート登録オプションのいずれかを使用するときには、必ず stack.h と stack.c の両方を、Stack クラスのインスタンスを使用する全てのコンパイル単位に含めてください。
• TEMPINC オプションを使用するときには、必ず stack.h ファイルのみを、Stack クラスのインスタンスを使用する全てのコンパイル単位に含めます。関数テンプレート定義が入っている stack.c ファイルは、stack.c の tempinc ファイルに自動的に含められます。Stack クラスのインスタンスを使用する他のすべてのコンパイル単位は、定義に対する参照を含みます。
• tempinc オプションが有効になっているときに、コンパイラによって定義されたマクロを使用して、実装ファイルをテンプレート・ヘッダー・ファイルに条件的に組み込むことによって、TEMPINC オプション付きまたは TEMPINC オプションなしのいずれかで機能するように、コードを構成することができます。以下の図に示すとおりです。

IBM i 機能を使用した処理 8-163
#ifndef __TEMPINC__
#include "stack.c"
#endif

図 8-144. TEMPINC オプション付きまたは TEMPINC オプションなしで機能する C++ コードの例

リンケージ
以下の図に示したように、テンプレートのコンストラクタ関数は、インラインで定義されます。他の関数は、ファイル stack.c 内の個別の関数テンプレートを使用して定義されていると想定してください。

```cpp
template <class Item, int size>
int Stack<Item, size>::operator << (Item item)
{
    if (top >= size) return 0;
    stack[top++] = item;
    return 1;
}

template <class Item, int size>
int Stack<Item, size>::operator >> (Item& item)
{
    if (top <= 0) return 0;
    item = stack[--top];
    return 1;
}
```

図 8-145. インラインで定義されるコンストラクター関数の例

注:
1. コンストラクターは、クラス・テンプレート宣言にインラインで定義されているので、内部リンケージを持っています。つまり、コンパイラがコンストラクター関数本体を、Stack クラスのインスタンスを使用する各コンパイル単位に生成することです。言い換えると、各単位はそれぞれコンストラクターの独自のコピーを持ち、使用します。
2. コンパイル単位における Stack クラスのインスタンスに、コンパイラは以下の関数の定義を生成します。

理由は、明示的特殊化があるからです。

クラス・テンプレートがソース・ファイル usrstack.cpp 内でインスタンス化されている場合、C++ プログラムには以下の図に示したようなコードが含まれます。
デフォルト・テンプレート・インスタンス化管理オプションの使用
デフォルト・テンプレート・インスタンス化管理オプションを使用する場合は常に、テンプレート・クラスのインスタンスを使用するすべてのコンパイル単位に、テンプレート宣言ヘッダー (.h ファイル) とアウトオブラインテンプレート定義 (.c ファイル) の両方を組み込む必要があります。コンパイラーは、関数テンプレート用に定義を生成します。例えば、図 8-163 ページの『クラス・テンプレート・インスタンス化の例』を参照してください。

単一のインスタンス化のためにコードを手動で構築
テンプレート定義を生成する ILE C++ 自動インスタンス化方法のどちらも使用したくない場合は、コンパイル単位にテンプレートを直接定義する方法でプログラムを構築することができます。このアプローチの利点として、モジュールが小さくなり、テンプレート定義をあらゆる場所にインクルードする場合と比べてコンパイル時間が短くなります。

テンプレート・インスタンス化のためにコードを手動で構築すると、自動インスタンス化に関連して発生する可能性がある問題を回避できます。発生する可能性がある問題のリストについては、図 8-161 ページの表 8-17 を参照してください。

単一のインスタンス化のためにコードを構築するには、以下のいずれかまたは両方の方法を使用してください。

・ 明示的インスタンス化を使用して、他のコンパイル単位で使用されるすべてのテンプレート・クラスに必要な定義をコンパイラーに生成させます。図 8-163 ページの『明示的インスタンス化の例』を参照してください。

・ 非メンバー関数テンプレートの明示的特殊化を使用して、コンパイラーにそれらを生成させます。

「ILE C/C++ 解説書」を参照してください。索引で「テンプレート」、「明示的特殊化」を調べてください。

注: これらの方法を使用する場合は、TEMPLATE(*NONE) コンパイラー・オプションを指定して、IFS tempinc ディレクトリー (または TEMPINC ファイル) を自動作成しないようにしてください。 ILE C/C++ コンパイラー・オプションについて詳しくは、「ILE C/C++ コンパイラー参照」を参照してください。

明示的インスタンス化
クラスの明示的インスタンス化により、クラスのオブジェクトを作成することなく、クラス特殊化の定義が強制されます。まだ明示的に特殊化されていないクラスのすべてのメンバーディニスタンス化が暗黙指定されます。プログラムですべてのクラス・メンバーが必要であるわけではない場合は、プログラムで必要な個別メンバーのみを明示的にインスタンス化できます。

IBM i 機能を使用した処理 8-165
明示的インスタンス化について詳しくは、「ILE C/C++ 解説書」を参照してください。索引で「テンプレート」を検索し、次に副見出し「インスタンス化」、「明示的」を検索してください。

図 8-147 において、以下のとおりです。

・ ヘッダー・ファイル stack.h は、Stack クラスのインスタンスを使用しているすべてのコンパイル単位に組み込まれます。
・ テンプレート実装ファイル stack.c は、1 つのファイルでのみ使用されます。
・ クラス Stack の 2 つのインスタンス化のみがプログラムで使用されています。これらには、テンプレート引数の < int, 20 > および < myClass, 100 > があります。

注: プログラムで使用されている Stack クラスのすべてのインスタンスが分かっている場合は、単一のコンパイル単位すべてのインスタンスを定義できます。

#include "stack.h"
#include "stack.c"
#include "myclass.h" // Definition of "myClass" class
template class Stack<int,20>;
template class Stack<myClass,100>;

図 8-147. クラスのすべてのインスタンスを単一のコンパイル単位で定義した例

**ILE テンプレート・レジストリー・オプションの使用**

ILE テンプレート・レジストリーは、バージョン 5、リリース 3 以降で使用できます。

注: ILE コンパイラによる再コンパイル・オプションの処理は、AIX コンパイラの場合と異なります。ILE C/C++ コンパイラは、再コンパイルが必要なときに警告を生成します。

ユーザーには、テンプレート・レジストリー・ファイルが生成されるディレクトリーへの読み取り/書き込み権限が必要です。ユーザーに読み取り/書き込み権限がないディレクトリーでテンプレート・レジストリーのアクセスまたは作成を試行すると、重大エラーが発生し、コンパイルは停止します。

注: ソース・ファイルにテンプレートがない場合、コンパイル中にエラーは生成されません。

**ILE テンプレート登録オプションの機能の仕方**

テンプレート登録では以下のことが行われます。

・ プログラムまたはサービス・プログラム内のあるすべてのテンプレート・インスタンス化のリポジトリが維持されます。
・ テンプレート・インスタンス化へのすべての参照が記録されます。
・ リンカーに定義が 1 つだけ提供されるようにします。

テンプレート登録は、テンプレート・インスタンス化の定義が含まれる単一モジュールを指定します。
・ テンプレート・インスタンス化を以前に含んでいたモジュールが再コンパイルされる場合、コンパイラは、そのインスタンス化がまだ存在するかを確認するためのチェックを行います。
・ 別のモジュール内のテンプレート・インスタンス化への依存関係があり、その定義が除去されている場合、コンパイラは、欠落インスタンス化を再生成するために従属モジュールを再コンパイルする必要があることを示す警告を出します。従属モジュールが再コンパイルされないと、リンクが失敗し、欠落定義があることを示すエラーが出されます。
初期コンパイル時、見つかったテンプレートの定義またはインスタンス化はすべて展開され、テンプレート登録がモジュール内の展開の場所で更新されます。それ以降のコンパイルでテンプレート・インスタンス化が必要な場合、コンパイラは、テンプレート登録をチェックして、別のモジュールにインスタンスが既に存在しているかどうかを確認します。存在する場合、コンパイラは新規インスタンス化を作成しませんが、新規参照を作成します。これにより、リンク・フェーズで定義が１つだけ提供されるようになります。

**TMPLREG パラメーターの値の指定**

TMPLREG パラメーターの値は、ソース・ファイルがストリーム・ファイルであるか、データ・メンバー・ファイルであるかによって決まります。

- *DFT が指定されている場合、以下のようになります。
  - ソース・ファイルがストリーム・ファイルであると、「templateregistry」ファイルがソース・ディレクトリーに作成されます。
  - ソース・ファイルがデータ・メンバー・ファイルであると、ソースが存在するライブラリーにメンバーや QTMPLEREG を持つファイル QTMPLEREG が作成されます。

- パス名が指定されている場合、以下のようになります。
  - ソース・ファイルがストリーム・ファイルであると、そのファイルがテンプレート登録として使用されます。

注: ファイルをテンプレート登録として明示的に使用したければ、「QSYS.LIB...」命名規則を使用する必要があります。

- ソース・ファイルがデータ・メンバー・ファイルであると、ソースが存在するライブラリーにソース・ファイルと同じ名前を持つファイルが作成されます。

**ILE TEMPINC オプションの使用**

ILE TEMPINC オプションは、コードが既にそこに構築されている場合に使用します。

ILE TEMPINC オプションを使用するには、以下の行を行います。

1. 1つ以上のヘッダー・ファイルでテンプレートをリストします。ただしそれらの引数は定義しないでください。
2. ソース・コードでヘッダー・ファイルのみをインクルードします。

注: 各ヘッダー・ファイルは、テンプレート実装ファイルの代わりに #pragma implementation ディレクトリーを使用してテンプレートを定義することができます。

3. モジュールをバイナードする際には、それらのコンパイルに使用したものと同じコンパイラ・オプションを使用します。以下に例を示します。

   CRTPGM PGM (MYLIB/MYPROG) MODULE(MYLIB/MYFILE)

このことは、ライブラリー、リンクージ、およびコード互換性を制御するオプションの場合に特に重要です。

**TEMPINC オプションの機能の仕方**

定義する必要がある関数テンプレートを持つヘッダー・ファイルごとに、コンパイラは IFS TEMPINC ディレクトリー (または tempinc ファイル) を生成します。IFS TEMPINC ディレクトリー (または tempinc ファイル) では、定義する必要がある関数テンプレート、クラス・テンプレート・メンバー関数、またはクラス・テンプレート静的データ・メンバーに #include ステートメントが生成されます。これらの関数およびメンバーは以下に存在します。
テンプレート宣言を持つヘッダー・ファイル
対応する IFS TEMPINC ディレクトリー（または tempinc ファイル）
テンプレート・パラメーターで使用される型を宣言する他のヘッダー・ファイル
コンバイラーは、バインダーの呼び出し前に、tempinc ファイルをコンバイルして、必要なテンプレート定義を生成します。テンプレートごとに生成される定義は 1 つだけです。

TEMPLATE パラメーターのデフォルトは "NONE" です。その他のオプションは "SRCDIR" であるか path-name です（path-name は IFS ディレクトリーやまたはファイル）。指定された IFS TEMPINC ディレクトリー（または tempinc ファイル）がまだ存在しない場合、コンバイラーが作成します。

注: 適用可能な xlc qshell コマンド・オプションは "-qtempinc=dir" です。この場合、dir はディレクトリー名です。完全修飾バス名を指定するか、現行ディレクトリーに対して相対的なパス名を指定できます。

**TEMPINC 管理インスタンス化用のプログラムの構成**

TEMPINC インスタンス化管理オプションを使用するときには、プログラムを構成する際に以下のガイドラインに従ってください。

- テンプレートの内部で使用されるが、テンプレート・パラメーターではない他の宣言がある場合、組み込まれたヘッダー・ファイルの 1 つまたは tempinc ファイルのいずれかにそれを配置するか、そのための #include を使用する必要があります。
- 以下のようにクラスを定義します。
  - テンプレート引数で使用される
  - ヘッダー・ファイルに関数テンプレートを生成するために必要である

注: 定義に他のヘッダー・ファイルが必要な場合、#include ディレクティブで組み込みます。これにより、単位のコンパイル時に定義が使用可能になります。

テンプレート実装ファイル内の関数定義は、明示的特殊化、テンプレート定義、またはその両方とすることができます。明示的特殊化はすべて、テンプレートによって生成された定義をオーバーライドします。

tempinc ファイルに異なるディレクトリーを指定する場合、バインド・ステップを含め、それをプログラムのすべてのコンパイルで整合したものとして指定する必要があります。以下に例を示します。

```cpp
crtpqm pgm (mylib/myprog) module(mylib/myfile)
```

関数インスタンス化を除去する場合、または tempinc ファイルが廃止されるようにプログラムを再編成する場合、これらのファイルの 1 つ以上を削除してから、プログラムを再コンパイルします。IFS TEMPINC ディレクトリー内のファイルについてエラー・メッセージが生成された場合、そのファイルを削除してから、再コンパイルします。すべての tempinc ファイルを削除する場合には、TEMPINC ディレクトリーおよびモジュールを削除してから、プログラムを再コンパイルします。

注:
1. コンパイラーは、IFS TEMPINC ディレクトリーまたはファイルの作成後、各単位がコンパイルされるときに、ファイルを更新します。コンパイラーは、個別のファイルから情報を除去することはありません。
2. モジュールを削除しないと、MAKEFILE 規則によりモジュールの再コンパイルが阻止され、IFS TEMPINC ファイルは、プログラムで使用されるすべてのコンパイル単位に必要なすべての行で更新することができなくなります。バインドは失敗します。

- テンプレート引数で使用されるクラスの定義をソース・コードに含めないでください。
注: 以下の例は、行わないことを示しています。

foo.h
    template<class T> void foo(T*);

hoo.h
    void hoo(A*);

foo.c
    template<class T> void foo(T* t)
        {t -> goo(); hoo(t);}

other.h
    class A {public: void goo() {}};

main.cpp
    #include "foo.h"
    #include "other.h"
    #include "hoo.h"
    int main() { A a; foo(&a); }

この場合、foo(T*) テンプレートをテンプレート型パラメーターとしての class A で拡張する必要があります。コンパイラは IFS tempinc ファイル TEMPINC\foo.cpp を作成します。ファイル内容 (単純化したもの) を以下に示します。

```
#include "foo.h" //the template declaration header
#include "other.h" //file defining template type parameter
#include "foo.c"  //corresponding template implementation
void foo(A*);  //triggers template instantiation
```

注: この例は適切にコンパイルできません。ヘッダー・ファイル hoo.h が組み込み条件を満たしていないが、foo(A*) の本体のコンパイルにはこのヘッダー・ファイルが必要であるからです。

図8-148. ソース・コードにも含まれるテンプレート引数で使用されるクラス定義の例 (コンパイルは適切に行われません)

1 つの解決策は、ステートメント #include "foo.h" をファイル foo.c に移動することです。

テンプレート実装ファイル: Stack のソースで、ファイル stack.c がテンプレート実装ファイルです。Stack クラス・テンプレートを使用してプログラムを作成するには、stack.h と stack.c の両方が同じディレクトリに存在する必要があります。このクラスのインスタンスを使用するソース・ファイルでは、stack.h をインクルードします。stack.c ファイルをソース・ファイルにインクルードする必要はありません。次に例を示します。

```
#include "stack.h"
void Swap(int i&, Stack<int,20>& s)
{
    int j;
    s >> j;
    s << i;
    i = j;
}
```

コンパイラは、関数 Stack<int,20>::operator<<(int) と Stack<int,20>::operator>>(int&) を自動的に生成します。
図8-149. テンプレート実装ファイルの例

テンプレート実装ファイルの名前変更または再配置: テンプレート実装ファイルの名前を変更するか、それを別のディレクトリに入れるには、#pragma implementation ディレクティブを使用します。

構文は、次のとおりです。

IBM i 機能を使用した処理 8-169
注:
1. path は、テンプレート実装ファイルのパス名を指定するために使用されます。部分パス名でしかない場合、ヘッダー・ファイルを含んでいるディレクトリに対して相対的なもののでなければなりません。
2. このパスは、ストリング・リテラルを作成する場合の通常規則を従った引用符付きストリングです。バックスラッシュは、二重引用符とする必要があります。

テンプレート実装ファイルの変更:
ファイル stack.def を stack.c の代わりにテンプレート実装ファイルとして使用するには、行 #pragma implementation("stack.def") を Stack クラス内の stack.h ファイルの任意の場所に追加します。これでコンパイラは、stack.h 同じディレクトリでテンプレート実装ファイル stack.def を検索します。

tempinc ファイル:
TEMPINC のテンプレート・インスタンス化管理オプションを使用すると、tempinc ファイルがコンパイラによって生成されます。

注: tempinc ファイルは編集しないでください。

図8-150 に、コンパイラが生成する一般的な tempinc ファイルを示します。

1. /*0000000000*/ #pragma sourcedir("c:\swearsee\src")
2. /*0698421265*/ #include "swearsee\src\list.h"
3. /*0000000000*/ #include "swearsee\src\list.c"
4. /*0698414046*/ #include "swearsee\src\mytype.h"
5. /*0698414046*/ #include ”/QIBM/include/iostream.h"
6. template void List<MyType>::push(MyType);
7. template MyType List<MyType>::pop();
8. ostream& operator<<(ostream&; List<MyType>);
9. #pragma undeclared
10. int count(List<MyType>);

図8-150. 一般的な tempinc ファイル

注:
1. この pragma により、コンパイラは、ILE C++ ファイル・インクルード規則で要求されるように、元のソース・ファイルを含むディレクトリで、ネストされたインクルード・ファイルを探ります。
2. tempinc ファイルに対応するヘッダー・ファイル。各 #include 行の先頭にあるコメント内の数字（この行の場合、/*0698421265*/）は、インクルードされたファイルのタイム・スタンプです。コンパイラは、この数字を使用して、tempinc ファイルが現行であるか、再コンパイルする必要があるかを判断します。3 行目のようにゼロ (0) のみを含むタイム・スタンプは、コンパイラがこのタイム・スタンプを無視することを意味します。
3. 2 行目のヘッダー・ファイルに対応するテンプレート実装ファイル。
4. コンパイラが tempinc ファイルのコンパイルに必要とする別のヘッダー・ファイル。コンパイラが tempinc ファイルのコンパイルに必要とする他のすべてのヘッダー・ファイルが、この時点で検出されます。
5. コンパイラが必要とする別のヘッダー・ファイル。6 から 7 行目の関数宣言で参照されます。
6. 明示的インスタンス化の最初のステートメント。このケースでは、クラス List<MyType> が定義されます。
7. 明示的インスタンス化の 2 つ目のステートメント。このケースでは、クラス List<MyType> のメンバー関数が生成されます。
8-170 IBM i: ILE C/C++ プログラマーの手引き
8. `operator<<` 関数は、`list.h` ヘッダー・ファイルのテンプレート宣言に一致した非メンバ関数です。コンパイラは、この宣言を挿入して、関数定義を強制的に生成します。

9. `#pragma undeclared` ディレクティブは、コンパイラのみが `tempinc` ファイルのみで使用します。少なくとも 1 つのコンパイラ単位で明示的に宣言された関数テンプレートはすべて、この行より前に出現在します。呼び出されたが宣言されていない関数テンプレートはすべて、この行より後に出現します。関数多重定義の解決の C++ 規則では、宣言済み関数テンプレートと宣言されていない関数テンプレートで扱いが異なるため、この分割が必要です。

10. `count` は、呼び出されたが宣言されていない関数テンプレートです。関数のテンプレート宣言は `list.h` に含まれていますが、インスタンス `count(List<MyType>)` は宣言されていません。

### ILE C および C++ プログラムでのテラスペースの使用

テラスペースは、IBM i ストレージ・モデルおよびランタイム環境の拡張機能です。

テラスペース・ストレージ・モデルでは、
- 最大で 2 GB から 244 バイトを引いた連続アドレス範囲をサポートします。小規模のランタイム環境でハイパフォーマンスのアプリケーションを作成できます。
- IBM i へのアプリケーションの移植を支援します。
- 既存 IBM i アプリケーションの継続的な発展をサポートします。

注: 「ILE 概念」に、ILE プログラムでテラスペースを使用する際の発生しうる問題と使用上のヒントが説明されています。

### サポートされるテラスペース環境

ILE C/C++ は、16 バイト・ポインター・ライブラリーを提供します。ILE C++ は、8 バイト・ポインタ・ライブラリー・ライブラリー (RTBND) も提供します。こうしたライブラリーは、バイナリ互換ではありません。テラスペースを使用する場合、バインドされたモジュールがバイナリ互換であるようにしてください。8-176 ページの「テラスペース環境でコードを移植する際のバイナリ互換性に関する考慮事項」を参照してください。

注: テラスペースおよび他のストレージ・モデルについて詳しくは、「ILE 概念」の「テラスペースおよび単一レベル保管」を参照してください。

### C/C++ のポインターのサポート

プログラムでテラスペースを使用可能にするために、C および C++ のコンパイラは、以下のポインターのサポートを提供します。
- 8 バイトまたは 16 バイトのポインターを明示的に宣言するため、以下の構文。
  - 8 バイト・ポインターを `char * __ptr64` として宣言する。
  - 16 バイト・ポインターを `char * __ptr128` として宣言する。
- C/C++ プログラミング環境に固有なデータ・モデルを指定し、オーバーライドするための構文。8-173 ページの「8 バイト実行時バインディング (RTBND) ライブラリー拡張」を参照してください。

### C/C++ ポインター型変換

IBM C および C++ コンパイラは、必要に応じて、関数および変数の宣言に基づいて、属性 `__ptr128` のポインターを属性 `__ptr64` のポインターに (また逆に) 变換します。

以下を考慮してください。
テラスベースを使用するためのバインド可能 API

IBM は、テラスベースの割り振りおよび廃棄用以下のバインド可能 API を提供しています。

_C_TS_malloc()
テラスベース内にストレージを割り振ります。

_C_TS_free()
テラスベースの直前の 1 つの割り振りを解除します。

_C_TS_realloc()
テラスベースの直前の割り振りのサイズを変更します。

_C_TS_calloc()
テラスベース内にストレージを割り振り、0 に設定します。

注: テラスベース・ストレージ・モデルが指定されていない場合、こうしたバインド可能 API は、その呼び出し側プログラムのストレージ・モデルに従って、単一レベル・ストレージまたはテラスベース・ストレージを割り振りまたは割り振り解除します。プロセス間通信 API および shmget() インターフェースについて詳しくは、IBM i Information Center Web サイト (http://www.ibm.com/systems/i/infocenter) の プログラミング カテゴリーの API セクションにある『UNIX-type APIs』のトピックを参照してください。
16 パイトの実行時バインディング・ライブラリー

RTBND コンバイラー・オプションを使用して実行時バインディング・ディレクトリーを指定しない限り、C++ 実行時環境ではニューラル・ストレージ・モデルが使用されます。

ニューラル・ストレージ・モデルは、16 パイトの内部ポインターを使用するため、以下のようになります。
- 単一レベル・ストレージ (SLS) モデルまたはテラスベース・ストレージ・モデルのいずれかによってビルトされたプログラムで使用できます。
- 実行時にさらにポインター・サイズ変換を必要とします。

注: _SYSPTR や_INVPTR などのシステム・ポインターは、16 パイトのサイズのままでです。

サービス・プログラムが 16 パイトの C++ 実行時ライブラリーを使用するには、RTBND オプションのデフォルト値が有効になっている必要があります。

注: テラスベース用のプログラムの有効化およびサービス・プログラムのバインディング規則について詳しくは、ILE 概念を参照してください。

8 パイト実行時バインディング (RTBND) ライブラリー拡張

最適な ISO 準拠が得られるようにするために、以下の場合に、8 パイトのポインター標準ライブラリーおよびオブジェクト・モデルを使用します。
- 8 パイトのポインターを使用して、テラスベース・ハイパフォーマンス・アプリケーションをビルドする
- 以前にビルトしたサービス・プログラムまたはランタイム・ライブラリーとの 2 進非互換性が重要でない

注: ILE C++ 8 パイト・ランタイム・ライブラリーは、バージョン 5.3 以上で使用可能です。

8 パイトのランタイム・ライブラリー拡張には、以下が含まれています。
- ライブラリー QSYS 内のバインディング・ディレクトリー QYPPLR510T
- サービス・プログラム QYPRT510T、QYPRL510T、および QYPWL530T (これらは、サービス・プログラム QYPRT370、QYPRL510、および QYPWL530 に置き換わる)
- 更新された C++ 標準ライブラリー・ヘッダー・ファイル

RTBND(*LLP64) が有効になっている場合、以下のようになります。
- (QSYS/QYPPLR510 ではなく) バインディング・ディレクトリー QSYS/QYPPLR510T が使用されます。
- C++ Itanium ABI データ・レイアウトが使用されます。C++ Itanium ABI の資料については、http://www.codesourcery.com/cxx-abi/abi.html を参照してください。
- 新規 POSIX 準拠 C シグナルが使用可能です。以下のようにになっています。
  - 同期 signal() 関数は、非同期 sigaction() 関数にマップされます。
  - 同期 raise() 関数は、非同期 kill() 関数にマップされます。
- 2 つの予約済みマクロ __LLP64_RTBND__ と __ASYNC_SIG__ が、値 1 を持つように定義されています。
RTBNDを使用したC++プログラムのパフォーマンスの最適化

C++ 一般的に、使用プランのサイズは、DTAMDLコンバイラー・オプションまたは
#pragma datamodelディレクティブのいずれかによって決定されます。

要件
8 バイトの実行時バインディング・ライブラリー拡張子を使用するC++プログラムをビルドするには、以下のコンバイラー・オプションを使用します。

- データ・モデル・オプション DTAMDL(*LLP64)
- ストレージ・モデル・オプション STGMDL(*TERASPACE)
- TERASPACE(*YES *TSIFC)
- 実行時バインディング・オプション RTBND(*LLP64)

注：これらのオプションの使用に特有な情報については、「ILE C/C++コンバイラー参照」を参照してください。

エラー状態
RTBND(*LLP64)オプションと両立しないオプションが使用された場合、診断エラーCZS2121が生成され、コンパイルは停止します。

RTBND(*LLP64)オプションがバージョン5.3より前のリリースで使用された場合、診断エラーCZS2120が生成され、コンパイルは停止します。

制限
RTBND(*LLP64)オプションが有効である場合、以下のようにになります。

- 8バイト・ポインタはテラスペースしか指すことができません。
- 8バイト・プロシージャー・ポインタは、常にテラスペースを介して活動状態のプロシージャーを参照します。
- 8バイト・ポインタ型はスペース・ポインタおよびプロシージャー・ポインタのみです。
- C++プログラムが以下の状態の場合、RTBNDオプションに*LLP64パラメーターを指定することができません。
  - デマンジングコードを使用している（8バイトと16バイトのポインタ名は異なる方法でマンジグルされています）。
  - 前のバージョンのコンパイラで作成されたオブジェクトを呼び出す（オブジェクトはバイナリー互換ではありません）。

各テラスペース・ストレージ・モデルの特性
以下の表では、2つのデータ・モデルの特性を比較します。

<table>
<thead>
<tr>
<th>RTBND(*DEFAULT)が有効になっている場合、以下のようにになります。</th>
<th>RTBND(*LLP64)が有効になっている場合、以下のようにになります。</th>
</tr>
</thead>
<tbody>
<tr>
<td>バインディング・ディレクトリー QSYS/QYPPLR510が使用されます。</td>
<td>バインディング・ディレクトリー QSYS/QYPPLR510Tが使用されます。</td>
</tr>
<tr>
<td>サービス・プログラム QYPRT370、QYPSSL510、およびQYPWL530が使用されます。</td>
<td>サービス・プログラム QYPRT510T、QYPSSL510TおよびQYPWL530Tが使用されます。</td>
</tr>
</tbody>
</table>

8-174 IBM i: ILE C/C++プログラマーの手引き
表8-18 デフォルトのテラベース・ストレージ・モデルと RTBND( LLP64) テラベース・ストレージ・モデルの特性の比較（続き）

<table>
<thead>
<tr>
<th>RTBND(*DEFAULT) が有効になっている場合、以下のようになります。</th>
<th>RTBND(*LLP64) が有効になっている場合、以下のようになります。</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>this</strong> ポインターのサイズは、クラス宣言の有効なデータ・モデルによって決まります。</td>
<td><strong>this</strong> ポインターのサイズは、データ・モデルのクラス宣言に関係なく 8 バイトです。内部コンパイラ生成構造は、すべて 8 バイト・ポインターとみなされます。これは、クラス宣言が #pragma datamodel(P128) で開まっていても該当します。</td>
</tr>
<tr>
<td><strong>C++</strong> 標準ライブラリー・ヘッダー・ファイルの 16 バイト互換セクションが使用されます。</td>
<td><strong>C++</strong> 標準ライブラリー・ヘッダー・ファイルの 8 バイト互換セクションが使用されます。</td>
</tr>
<tr>
<td>すべての内部構造（例えば、仮想関数テーブル）で 16 バイト・ポインターが使用されます。</td>
<td>すべての内部構造（例えば、仮想関数テーブル）で 8 バイト・ポインターが使用されます。</td>
</tr>
<tr>
<td>すべての C++ プログラムおよび C++ サービス・プログラムには、RTBND(*DEFAULT) オプションでビルドされたモジュールしか入れることができません。</td>
<td>すべての C++ プログラムおよび C++ サービス・プログラムには、RTBND(*LLP64) オプションでビルドされたモジュールしか入れることができません。</td>
</tr>
</tbody>
</table>

新規の ABI 名 Mangling 方式が代わりに使用されるので、16 バイト・ライブラリー用に書かれたコードのデータバインディングは機能しません。ABI 名 Mangling 方式によって、RTBND(*LLP64) でビルドされていないモジュールのパインディングは不可能になります。

注：1つ以上のモジュール内の名前が名前 Mangling を無効にするように明示的に作成されている場合、非互換が発生する可能性があります。

派生クラスは、基本クラスと同じデータ・モデルでなければなりません。

main() の宣言は、モジュールのコンパイル時に使用されたデータ・モデル・オプションと整合性がなければならないません。8-177ページの「一貫した引数宣言の維持」を参照してください。

変数引数リストで宣言された関数は、引数リストの宣言時に有効になっているデータ・モデルによって管理されます。引数リストの変数は、ポインター変数のサイズに関して常に整合している必要があります。ILE C/C++ コンパイラは、変数リスト内のポインターのポインター型変換の1つのレベルを提供します。

クラスまたは構造では、そのクラスまたは構造が完全に宣言されたときに有効になっているデータ・モデルが使用されます。このデータ・モデルは、この同じクラスまたは構造がフォワード宣言されたときに有効になっているデータ・モデルと異なることがあります。8-177ページの「例：データ・モデルでのフォワード宣言の効果」を参照してください。

プロトタイプ化されていない関数は、その最初の参照の時点で有効になっているデータ・モデルによってそのシンタッチャーが推測されます。ヘッダー・ファイルが組み込まれていないと、C 実行時関数で予測不能な状況が生じる可能性があります。

派生クラスが 8 バイトであるのでは、データ・モデルが LLP64 の場合のみです。

**C++** テンプレートは、その宣言時に有効になっているデータ・モデルを採用し、そのテンプレートの依頼のインスタンス化にそのデータ・モデルを適用します。派生クラスはすべて、基本クラスと同じデータ・モデルでなければなりません。8-179ページの「例：テンプレートがデータ・モデルを採用する方法」を参照してください。

**C++** 関数シグニチャーは、クラス関数宣言を管理するデータ・モデルの影響を受けます。8-180ページの「例：関数の多重定義」を参照してください。

(&) 演算子のアドレスは 8 バイト結果に戻します。

ソース・コードでポインター・サイズを混合すると、構造および関数プロトタイプでまだ 16 バイト・ポインターを使用する多くのシステム API の使用が許可されるようになります。
テラスベース環境でコードを移植する際のバイナリ互換性に関する考慮事項
すべてのバインド済みモジュール内のクラスの内部表現は、バイナリ互換でなければなりません。
RTBND(*LLP64) を指定してビルドされたモジュールと指定せずにビルドされたモジュールのバインドイングを防止する明示的なメカニズムはありません。

以下の場面には常に、予期しない実行時の動作が発生することがあります。
- RTBND(*LLP64) オプションを指定してビルドされた C++ プログラムが、このオプションを指定せずにビルドされた C++ サービス・プログラムにバインドされた場合
- RTBND(*LLP64) オプションを指定せずにビルドされた C++ プログラムが、このオプションを指定してビルドされたサービス・プログラムにバインドされた場合

テラスベース環境の指定
ストレージ・モデル (STGMDL) コンバイラー・オプションは、プログラム全体のテラスベース環境を決定します。
データ・モデル (DTAMDL) コンバイラー・オプションは、適用可能なモジュールのポインター・サイズ宣言を決定します。8 バイトの実行時ライブラリを使用するには、DTAMDL(*LLP64) を指定します。

ソースの 1 つのセクションでデフォルトのデータ・モデル (DTAMDL) 設定の効果をオーバーライドするには、DTAMDL(*LLP64) または #pragma datamodel ディレクティブのいずれかを使用できます。8-177 ページの『例: データ・モデルでのフォワード宣言の効果』および、8-179 ページの『例: データ・モデルを採用する方法』を参照してください。

注: #pragma datamodel ディレクティブは、DTAMDL オプション設定をオーバーライドします。
特定の変数のセクション設定をオーバーライドするには、ポインターに属性を指定します。8-177 ページの『一貫した引数宣言の維持』および、8-180 ページの『例: 関数の多重定義』を参照してください。

特定のポインターのサイズの決定
特定のポインターのサイズを決定するには、以下のようにします。
1. ポインター属性を確認します。
2. ポインター属性が存在しない場合、#pragma datamodel 設定を確認します。
3. #pragma datamodel ディレクティブがソース・セクションに含まれていない場合、C または C++ モジュールの作成またはバインド時に指定されたデータ・モデル (DTAMDL) コンバイラー・オプションを確認します。
一貫した引数宣言の維持

main() で宣言された引数リストの妥当性は、モジュールが作成またはバインドされる際に有効なデータ・モデルによって異なります。

表 8-19. 一貫した引数宣言の維持

<table>
<thead>
<tr>
<th>宣言</th>
<th>有効なデータ・モデル</th>
</tr>
</thead>
<tbody>
<tr>
<td>main(int argc, char ** argv)</td>
<td>P128, LLP64</td>
</tr>
<tr>
<td>main(int argc, char * argv[])</td>
<td>P128, LLP64</td>
</tr>
<tr>
<td>main(int argc, char *__ptr128 *__ptr128 argv)</td>
<td>P128, LLP64</td>
</tr>
<tr>
<td>main(int argc, char *__ptr128 argv[])</td>
<td>P128</td>
</tr>
<tr>
<td>main(int argc, char *__ptr128 argv)</td>
<td>P128</td>
</tr>
<tr>
<td>main(int argc, char *__ptr64 *__ptr64 argv)</td>
<td>P128, LLP64</td>
</tr>
<tr>
<td>main(int argc, char *__ptr64 argv[])</td>
<td>LLP64</td>
</tr>
<tr>
<td>main(int argc, char *__ptr64 argv)</td>
<td>LLP64</td>
</tr>
<tr>
<td>main(int argc, char *__ptr128 *__ptr128 argv)</td>
<td>(どのデータ・モデルでも無効)</td>
</tr>
<tr>
<td>main(int argc, char *__ptr128 *__ptr64 argv)</td>
<td>(どのデータ・モデルでも無効)</td>
</tr>
</tbody>
</table>

注: 同じ規則が、main() の 3 番目のパラメーター envp にも適用されます。さらに、argv および envp の結果のポインター・サイズも同じでなければなりません。

ソース・コードのサンプル

以下のソース・サンプルは、テラスペースを使用する際にプログラムをコーディングする方法を示しています。

例: データ・モデルでのフォワード宣言の効果

以下の図では、struct Foo は、フォワード宣言時に有効な P128 データ・モデルを使用しません。代わりに、struct Foo は、struct Foo が完全に宣言されたときに有効な LLP64 データ・モデルを使用します。

```c
#pragma datamodel(P128)
struct Foo; //forward declaration
#pragma datamodel(LLP64)
struct Foo {
    char* string; // this pointer is 8-byte because LLP64 datamodel
    // was in effect for the struct definition.
};
#pragma datamodel(pop)
#pragma datamodel(pop)
```

図 8-15I. LLP64 データ・モデルに基づいたフォワード宣言の例

例: new または delete 演算子の再定義

C++ テラスペースからストレージを動的に割り振るモジュールを作成するには、CRTCPPMOD および CRTBNDCPP コマンドに TERASPACE(*YES *TSIFC) を指定します。これにより、C 実行時間数 (malloc, calloc, free など) が新規テラスペースの同等のものに再マップされます。これは、new および
delete 演算子を使用して動的に割り振られたストレージにも影響します。new および delete 演算子により他の方法でストレージを割り振りたい場合、これらの演算子をオーバーライドすることができます。

以下のコードは、関数 (malloc や free など) を呼び出すグローバルな new および delete 演算子を再定義します。

注: _set_mt_new_handler の宣言を検出するには、以下のソースを DEFINE("_MULTI_") でコンパイルする必要があります。
例: テンプレートがデータ・モデルを採用する方法
次の例では、
- FooT のインスタンス化では、P128 データ・モデルが使用されます
・FooTZ のインスタンス化では、LLP64 データ・モデルが使用されます

```cpp
#pragma datamodel(LLP64)
template <class T>
class FooTZ {
    public:
        T bar(const char * a, T x) { return x; }
};
#pragma datamodel(pop)

#pragma datamodel(P128)
template <class T>
class FooT {
    public:
        T bar(const char * a, T x) { return x; }
};
#pragma datamodel(pop)
```

図 8-153. テンプレートの宣言時に有効なデータ・モデルを採用するテンプレートの例

### 例: 関数の多重定義

関数シグニチャーは、クラス/関数宣言を管理するデータ・モデルの影響を受けます。

例えば、int Bar::foo(const char *) は以下にマングルされます。

- foo_3BarFPCc (データ・モデルが P128 の場合)
- foo_3BarZFPcc (データ・モデルが LLP64 の場合)
- _ZN3Bar3fooEPKc (実行時バイニングが LLP64 の場合)

ポインタ引数のサイズを除いてすべての点で同一である多重定義メソッドを作成できます。

例:

```cpp
class Bar {
    int foo(const char* __ptr128);
    int foo(const char* __ptr64);
};
```

### 実行時型情報を使用したキャスト

実行時型情報 (RTTI) は、ISO 規格委員会によって行われた C++ 言語に対する重要な拡張です。

このトピックでは以下について説明します。

- **RTTI 言語拡張**
- **C++ 言語で定義された RTTI の使用**
- **コンストラクターおよびデストラクターにおける RTTI の使用**
- RTTI に対する ILE C++ の拡張
RTTI 言語拡張

C++ 言語では、コードの再使用と、パーセからプログラムのビルドの両方をサポートします。さまざまな C++ ライブラリによって内部的に使用される RTTI メカニズムの間で不適切がある場合があります。RTTI 言語拡張では、そうした不適切を解決します。

RTTI の C++ 言語サポートには、以下が含まれます。

dynamic_cast 演算子
この演算子では、型変換とキャストを 1 つの操作に結合します。これは、要求されたキャストが有効であるかどうかを検査し、有効な場合にのみ、そのキャストを実行します。

typeid 演算子
この演算子は、オブジェクトの実行時型を戻します。typeid 演算子で戻される型の定義に使用されます。このクラスにより、ユーザーが、自分のニーズに合わせて、RTTI のシェービングと拡張を行えるようになります。デバッガーやデータベース・システムなどのオブジェクト入出力システムの実装者にとって、これは非常に有効な機能となります。

C++ 言語で定義された RTTI の使用

RTTI を使用するには、dynamic_cast 演算子と typeid 演算子について熟知する必要があります。

dynamic_cast 演算子
動的キャスト式は、基底クラス・ポインタを派生クラス・ポインタにキャストする場合に使用します。これをダウンキャストと呼びます。

dynamic_cast 演算子は、以下を行います。

- 標準的な静的キャストよりもはるかに安全なダウンキャストを実行する
- オブジェクトの基底クラスへのポインターを渡して、派生クラスのそのオブジェクトへのポインターを取得する
- 特定の派生クラスが実際に存在する場合にのみポインターを返す

注: 指定した派生クラスが存在しない場合は、ゼロが返されます。

動的キャストの形式は、以下のとおりです。

```cpp
 dynamic_cast<type_name>(expression)
```

この演算子は、式を目的のタイプ type_name に変換します。type_name は、クラス・タイプへのポインタまたは参照にすることができます。type_name へのキャストが失敗した場合、式の値はゼロになります。

ポインタによる動的キャスト:
動的キャストは、他の方法では使用できない、派生クラスの詳細を使用できるように、派生クラスを指すポインターを取得するために使用されます。例については、8-182ページの図8-154を参照してください。
class employee {
public:
    virtual int salary();
};

class manager : public employee {
public:
    int salary();
    virtual int bonus();
};

図8-154. 他の方法では使用できない詳細を使用するために派生クラスを指すポインターをキャストするための ILE ソース

図8-155 で使用されたクラス階層では、動的キャストは、manager::bonus() 関数を管理職の給与計算に含め、一般従業員の給与計算には含めないようにする場合に使用できます。dynamic_cast 演算子は、bonus() メンバー関数を使用するために、基本クラス employee を指すポインターを使用して、派生クラス manager を指すポインターを取得します。

図8-155 では、基本クラス employee およびその派生クラスを (ソース・コードの変更が望ましくないライブラリーの一部として) ユーザーが使用できない場合にのみ、動的キャストが必要になります。それ以外の場合、新規仮想関数を追加して、これらの関数の特殊化された定義を派生クラスに提供するほうが、この問題を解決するには望ましい方法です。

void payroll::calc (employee *pe) {
    // employee salary calculation
    if (manager *pm = dynamic_cast<manager*>(pe)) {
        // use manager::bonus()
    } else {
        // use employee's member functions
    }
}

図8-155. 指定した計算でのみメンバー関数を使用するために派生クラスを指すポインターを取得するための ILE ソース

図8-155 では以下のようになっています。
- pe が実行時に実際に manager オブジェクトを指している場合、動的キャストが成功し、pm が manager を指すポインターに初期化され、bonus() 関数が使用されます。
- pe が実行時に manager オブジェクトを指していない場合、pm がゼロに初期化され、employee 基本クラスの関数のみが使用されます。

参照による動的キャスト：

C++ dynamic_cast 演算子は、参照型にキャストする場合に使用できます。C++ 参照キャストはポインター・キャストと似ています。参照キャストは、from参照から派生クラス・オブジェクトへの基本クラス・オブジェクト to参照にキャストする場合に使用できます。

参照型への動的キャストでは、type_name は型を表し、expression は参照を表します。この演算子は、式を所要の型 type_name & に変換します。

ゼロ参照のようなものがないため、結果 (動的キャストの結果の参照) をゼロと比較することでは、参照型を使用する動的キャストの成功を検証することはできません。参照型への動的キャストが失敗すると、bad_cast 例外がスローされます。

参照による動的キャストは、コーディング前提をテストするにはよい方法です。8-183 ページの図8-156では、図8-154で使用された例が、参照キャストを使用するように変更されています。
void payroll::calc (employee &re) {
  // employee salary calculation
  try {
    manager &rm = dynamic_cast<manager &>(re);
    // use manager::bonus()
  }
  catch (bad_cast) {
    // use employee's member functions
  }
}

注: 図 8-156 は、テストとして使用される dynamic_cast 演算子を示すためのものにすぎません。この例は、実行フローの制御に例外を使用しているため、適切なプログラミング・スタイルを示すものではありません。8-182 ページの図 8-155 に示したポインタによる dynamic_cast を使用するほうが良い方法です。

### typeid 演算子

**typeid 演算子**

typeid 演算子は、基底クラスへのポインタを与えられたオブジェクトの正確な型を識別します。これは通常、システムが操作する全オブジェクトの共通インタフェースを想定できない操作について、その実行に必要な情報をアクセスするために使用されます。オブジェクト入出力システムやデータベース・システムは、使用可能な仮想関数がないオブジェクトにサービスを実行する必要があることがよくあります。

**typeid 演算子により、これが可能になります。**

**typeid 演算の形式は、次のとおりです。**

```cpp
typeid(type_name expression)
```

**typeid 演算の結果:** typeid 演算の結果には型 const type_info & があります。

表 8-20 では、さまざまな typeid 演算の結果を要約しています。

<table>
<thead>
<tr>
<th>ポリマフック型への参照</th>
<th>参照された完全なオブジェクトの<strong>type_info</strong>。</th>
</tr>
</thead>
<tbody>
<tr>
<td>ポリマフック型を指すポインタ</td>
<td>指された完全なオブジェクトの動的型。注: このポインタがゼロの場合、typeid 式では bad_typeid 例外がスローされます。</td>
</tr>
</tbody>
</table>
| 非ポリマフック型 |expression の静的型を表す**type_info**。
注: expression は評価されません。|

式での typeid 演算子の使用: 8-184 ページの図 8-157 の例では、employee クラス階層内のオブジェクトの実行時型を比較する式で typeid 演算子を使用しています。
//...
employee *pe = new manager;
employee& re = *pe;
//...
typeid(pe) == typeid(employee*) // 1. True - not a polymorphic type
typeid(&re) == typeid(employee*) // 2. True - not a polymorphic type
typeid(*pe) == typeid(manager) // 3. True - *pe represents a polymorphic type
typeid(re) == typeid(manager) // 4. True - re represents the object manager

typeid(pe) == typeid(manager*) // 5. False - static type of pe returned
typeid(pe) == typeid(employee) // 6. False - static type of pe returned
typeid(pe) == typeid(manager) // 7. False - static type of pe returned

typeid(*pe) == typeid(employee) // 8. False - dynamic type of expression is manager
typeid(re) == typeid(employee) // 9. False - re actually represents manager
typeid(&re) == typeid(manager*) //10. False - manager* not the static type of re
//...

図8-157. 式での typeid 演算子の例：

注：
1. 最初の比較では、pe はポインタです (すなわち、ポリモアフィック型ではありません)。したがって、式 typeid(pe) は、employee* と同等である pe の静的型を戻します。
2. 2番目の式では、re は、参照されるオブジェクトのアドレスを表します (すなわち、ポリモアフィック型ではありません)。したがって、式 typeid(re) は、employee を指すポインタである re の静的型を戻します。
3. 3番目と4番目の比較では、typeid によって戻される型は、式がポリモアフィック型を表す場合のみ式の動的型を表します。
4. 比較 5、6、および 7 は false です。調べられる式の型 (pe) であり、pe で指されるオブジェクトの型ではないからです。

これらの例では、type_info オブジェクトを直接扱っていません。組み込み型で typeid 演算子を使用するには、type_info オブジェクトとの相互作用が必要です。

int i;
//...
typeid(i) == typeid(int) // True
typeid(8) == typeid(int) // True
//...

図8-158. typeid 演算子の例

type_info クラス：実行時型情報 (RTTI) で typeid 演算子を使用するには、C++ の標準ヘッダー <typeinfo.h> をインクルードする必要があります。このヘッダーでは、以下のクラスを定義します。

type_info
実装に使用可能な RTTI を記述します。これは、比較演算子と照合演算子を提供するポリモアフィックなタイプで、表されるタイプの名前を戻すメンバ関数を提供します。

クラス type_info のコピー・コンストラクターと代入演算子は private です。このタイプのオブジェクトをコピーすることはできません。

bad_cast
失敗した動的キャストを報告するために例外としてスローされるオブジェクトのタイプを定義します。
bad_typeid

typeid 式内の NULL ポインターを報告するために例外としてスローされるオブジェクトのタイプを定義します。

コンストラクターおよびデストラクターにおける RTTI の使用
コンストラクターまたはデストラクター、あるいはそれらから呼び出された関数で、typeid 演算子と
dynamic_cast 演算子を使用できます。

使用された dynamic_cast のオペランドが構築または消滅するオブジェクトを参照する場合、typeid は、
コンストラクターまたはデストラクターのクラスを表す type_info を戻します。

dynamic_cast のオペランドが構築または消滅するオブジェクトを参照する場合、このオブジェクトは、コンストラクターまたはデストラクターのクラスの型を持つ完全なオブジェクトと見なされます。

オペランドが構築または消滅するオブジェクトを参照し、オペランドの静的タイプが、コンストラクターまたはデストラクターのクラス、またはそのベースの 1 つのオブジェクトでない場合、typeid 演算と
dynamic_cast 演算の結果は未定義です。

RTTI に対する ILE C++ の拡張
ILE C++ の extended_type_info クラスは、永続オブジェクト・ストアの実装サポートを提供するために
設計されました。サポートされる必要がある基本操作は以下のとおりです。

・オブジェクトのメモリーの割り振り
・オブジェクト配列のメモリーの割り振り
・オブジェクトの初期構築
・オブジェクト配列の初期構築
・オブジェクトのコピー構築
・オブジェクト配列のコピー構築

構築中に発生する例外を検出するために、メモリーの破棄または割り振り解除を行う追加操作も提供されま
す。これらの演算は、次のとおりです。

・オブジェクトの消滅
・オブジェクト配列の消滅
・オブジェクトのメモリーの消滅
・オブジェクト配列のメモリーの割り振り解除

extended_type_info クラス

8-187 ページの図 8-159 は、extended_type_info クラスを使用したコード・サンプルです。

extended_type_info クラス定義は以下のとおりです。

size() extended_type_info オブジェクトによって表されるタイプのサイズ。

create(void*)
この関数は、at によって指されたストレージ・ロケーションに extended_type_info オブジェク
トで表されたタイプのオブジェクトを作成するために呼び出します。

create(void*, size_t)
この関数は、at によって指されたストレージ・ロケーションに extended_type_info オブジェク
copy(void* to, const void* from)
この関数は、from で参照されたオブジェクトの値を使用して、extended_type_info オブジェクトによって表されたタイプのオブジェクトを、to によって指されたストレージ・ロケーションにコピーするために呼び出します。

copy(void* to, const void* from, size_t)
この関数は、from で参照されたオブジェクトの値を使用して、extended_type_info オブジェクトによって表されたタイプのオブジェクトの配列を、to によって指されたストレージ・ロケーションにコピーするために呼び出します。構成時に例外がスローされた場合、copy() は、既に構成されている配列エレメントを破棄してから、その例外を再スローします。

destroy(void*)
この関数は、at によって指されたストレージ・ロケーションで、extended_type_info オブジェクトで表されたタイプのオブジェクトを破棄するために呼び出します。

destroy(void*, size_t)
この関数は、at によって指されたストレージ・ロケーションで、extended_type_info オブジェクトで表されたタイプのオブジェクトの配列を破棄するために呼び出します。破棄時に例外がスローされた場合、destroy() は、既に構成されている残っている配列エレメントを破棄してから、その例外を再スローします。残っているエレメントの破棄時に例外がもう 1 つ発生した場合、destroy() は terminate() を呼び出します。

allocObject()
この関数は、extended_type_info オブジェクトによって表されたタイプのオブジェクト用のメモリーを割り振るために呼び出します。初期化は実行されません。ユーザーは、create(void*) または copy(void*, const void*) 関数を使用して新規メモリーを初期化する必要があります。

allocArray(size_t)
この関数は、extended_type_info オブジェクトによって表されたタイプのオブジェクトの配列用のメモリーを割り振るために呼び出します。初期化は実行されません。ユーザーは、create(void*, size_t) または copy(void*, const void*, size_t) 関数を使用して新規メモリーを初期化する必要があります。

deallocObject(void*)
この関数は、extended_type_info オブジェクトによって表されたタイプのオブジェクトの割り振りを解除するために呼び出します。前もって破棄は実行されません。ユーザーは、メモリーの割り振りを解除する前に、destroy(void*) を使用してオブジェクトを終了する必要があります。

deallocArray(void*, size_t)
この関数は、extended_type_info オブジェクトによって表されたタイプのオブジェクト配列の割り振りを解除するために呼び出します。前もって破棄は実行されません。ユーザーは、メモリーの割り振りを解除する前に、destroy(void*, size_t) を使用して配列を終了する必要があります。

linkageInfo()
この関数は、クラス・タイプのマングル名を返します。
class extended_type_info : public type_info {
    public:
~extended_type_info();

    virtual size_t size() const=0;
    virtual void* create(void* at) const=0; //object
    virtual void* create(void* at, size_t count) const=0; // array
    virtual void* copy (void* to, const void* from) const=0; //object
    virtual void* copy (void* to, const void* from, size_t count) const=0; //array
    virtual void* destroy(void* at) const=0; //object
    virtual void* destroy(void* at, size_t count) const=0; //array
    virtual void* allocObject() const=0; //object
    virtual void* allocArray(size_t count) const=0; //array
    virtual void* deallocObject(void* at) const=0; //object
    virtual void* deallocArray(void* at, size_t count) const=0; //array
};
国際ロケールおよびコード化文字セットの使用

このトピックでは、以下を行うための方法について説明します。
• 代替コード化文字セットの処理
• 国際ロケールの使用

プログラムの国際化

このトピックでは、以下を行うための方法について説明します。
• 特定のコード化文字セット ID (CCSID) を使用してソース物理ファイルを作成する
• ソース物理ファイル内のメンバーの CCSID を、別のソース物理ファイル内の別のメンバーの CCSID に変更する
• メンバーの特定のソース・ステートメントの CCSID を変換する

Integrated Language Environment C/C++ コンパイラーは、ほとんどの 1 バイト EBCDIC CCSID で作成されたソース・コードを認識します。

CCSID 905 および 1026 で作成されたソース・コードは、" 文字がこれらの CCSID で異なるため、認識できないことがありますが。

CRTCMMOD/CRTCPMMOD および CRTBNDC/CRTBNDCPP コマンドでは、混合バイト環境で SRCSTMF パラメーターはサポートされません。

2 バイト文字セット (DBCS) ソース・コードでは、特殊なプログラミングの考慮事項に注意する必要があります。

注: CCSID (1 次言語で決定) が CCSID 037 (米国英語) 以外の場合は、CCSID 値番号を使用してソース物理ファイルをタグ付けする必要があります。

コード化文字セット ID

コード化文字セット ID (CCSID) は、エンコード・スキーム (EBCDIC, ASCII, または 8 ビット ASCII), 文字セット ID, コード・ページ ID, および使用されるコード化グラフィック文字表現を固有に識別する追加のコーディング関連情報の特定のセットから構成されます。

文字セット は、グラフィック文字の集まりです。

グラフィック文字 は、文字、数字、句読記号などの記号です。

コード・ページ は、1 グループのグラフィック文字のバイナリー ID のセットです。

コード・ポイント は、各グラフィック文字に割り当てられ、情報の入力、保管、変更、表示、または印刷に使用されるバイナリー値です。

文字データ処理体系 (CDRA) は、文字の表現に使用されるコード・ポイントを識別し、意味を保存するために必要に応じて文字データを変換する CCSID 値を定義します。
CCSID へのソース・ファイル変換

Integrated Language Environment C/C++ ソース・プログラムは、複数のソース・ファイルで構成できます。1つのルート・ソース・メンバーおよび複数の 2 次ソース・ファイル (組み込みファイルや DDS ファイルなど) を使用できます。

ルート・ソース・メンバーとは異なる CCSID でタグ付けされている 2 次ソース・ファイルがある場合、その内容は、Integrated Language Environment C/C++ コンパイラーによって読み取られる際に、ルート・ソース・メンバーの CCSID に自動的に変換されます。

1 次ソース物理ファイルに CCSID 65535 が含まれている場合、ソース物理ファイルにジョブ CCSID が想定されます。ソース物理ファイルに CCSID 65535 が含まれていて、ジョブが CCSID 65535 で、システムに非 65535 が含まれている場合、ソース物理ファイルにシステム CCSID 値が想定されます。1 次ソース物理ファイル、ジョブ、およびシステムに CCSID 65535 が含まれている場合、CCSID 037 が想定されます。2 次ファイル、ジョブ、およびシステムの CCSID が 65535 の場合、1 次ソース物理ファイルの CCSID が想定され、変換は行われません。

コード化文字セット ID を使用したソース物理ファイルの作成

ソース物理ファイルの作成時に、CCSID パラメーターで使用したい文字セットを指定します。CCSID パラメーターのデフォルトは、ジョブの CCSID です。

以下の図は、CCSID 273 のルート・ソース・メンバー、および CCSID がそれぞれ異なる複数の組み込みファイルを持つプログラム・オブジェクトを作成するときに何が起こるかを示しています。Integrated Language Environment C コンパイラーは、組み込みファイルを CCSID 273 に変換します。プログラム・オブジェクトは、ルート・ソース・メンバーと同じ CCSID で作成されます。

図 9-1. ソース・ファイルの CCSID 変換

注: ルート・ソース・メンバー CCSID と組み込みファイル CCSID の一部の組み合わせはサポートされていません。

例:

9-2 IBM i: ILE C/C++ プログラマーの手引き
以下の例は、ライブラリー MYLIB 内のソース物理ファイル QCSRC の CCSID 273 の指定方法を示しています。

CCSID 273 のソース物理ファイルを作成するには、以下を入力します。

CRTSRCFILE(MYLIB/QCSRC) CCSID(273)

コード化文字セット ID (CCSID) の変更
ソース物理メンバーの CCSID を別の CCSID に変更するには、パラメーター FMTOPT(*MAP) を指定したコマンド CPYF を使用して、別の CCSID のソース物理メンバーのコピーを入手します。

以下の例は、ソース・ファイル内の CCSID 037 のメンバーを CCSID 273 に変更する方法を示しています。

例:

CRTSRCFILE(MYLIB/NEWCCSID) CCSID(273)
CPYF FROMFILE(MYLIB/QCPPSRC) TOFILE(MYLIB/NEWCCSID) FROMMBR(HELLO) TOMBR(HELLO)
MBROPT(*ADD) FMTOPT(*MAP)

注:
1. 最初のコマンドは CCSID 273 を作成します。
2. 有効な変換が定義されている限り、ファイル・コピー操作の間、FROM メンバーの文字データが「FROM ファイル」フィールドの CCSID と「TO ファイル」フィールドの CCSID との間で変換されます。
3. ファイル NEWCCSID 内の HELLO メンバーが CCSID 273 で QCSRCC にコピーされます。CCSID 65535 または *HEX が使用された場合、フィールドの文字データがビット・データとして扱われ、変換されないことを示します。

ソース・ファイル内のストリング・リテラルの変換
ソース・プログラム内のストリング・リテラルは、#pragma convert ディレクティブが指定されたポイントからプログラムの終わりまで変換することができます。#pragma convert ディレクティブは、プログラム内で該当ポイントから前進方向のストリング・リテラルの変換に使用する CCSID を指定します。変換は、ソースの終わりまで、または別の #pragma convert ディレクティブが指定されるまで続行されます。

値 65535 の CCSID が指定されている場合、ルート・ソース・メンバーの CCSID とみなされます。ソース・ファイルの CCSID 値が 65535 の場合、CCSID 037 とみなされます。変換前のストリング・リテラルの CCSID は、ルート・ソース・メンバーと同じ CCSID です。CCSID は EBCDIC または ASCII のいずれかを使用します。

例:

以下の例は、ソース物理ファイルの CCSID が EBCDIC であっても、T1520CCS のストリング・リテラルを ASCII CCSID 850 に変換する方法を示しています。

注: この例では、TGTCCSID パラメーターにデフォルト値 *SOURCE が使用されます。
1. 以下のとおり入力してください。

CRTCNSDC PGM(MYLIB/T1520CCS) SRCFILE(QCPPLE/QCSRC)

次のソースを使用してプログラム T1520CCS を作成する場合:
/* This program uses the #pragma convert directive to convert */
/* string literals. */
#include <stdio.h>

char EBCDIC_str[20] = "Hello World";
#pragma convert(850) /* Use the #pragma convert */
/* directive to convert the */
/* string to ASCII, CCSID 850. */

char ASCII_str[20] = "Hello World";
#pragma convert(0) /* Stop string conversion. */

int main(void)
{
    int i;
    printf("EBCDIC_str(hex) = "); /* Print hex value of EBCDIC */
    for (i = 0; i < 11; ++i) /* string. */
        printf("%X ",EBCDIC_str[i]);
    printf(\\n\\n); /* Print hex value of ASCII */
    for (i = 0; i < 11; ++i) /* string. */
        printf("%X ",ASCII_str[i]);
}

図9-2. T1520CCS — ストリングおよびリテラルを変換するための ILE C ソース

CRTBNDC コマンドは、ライブラリー MYLIB 内にプログラム T1520CCS を作成します。プログラム T1520CCS は EBCDIC ストリング「Hello World」を ASCII CCSID 850 に変換します。

2. プログラム T1520CCS を実行するには、以下を入力します。

CALL PGM(MYLIB/T1520CCS)

出力は以下のようになります。

```
EBCDIC_str(hex) = C8 85 93 93 96 40 E6 96 99 93 84
ASCII_str(hex) = 48 65 6C 6C 6F 20 57 6F 72 6C 64
実行キーを押して端末セッションを終了してください。
```

ワイド文字リテラルのユニコード・サポートの使用

ユニコード文字ストレージを使用すると、複数の文字セットの文字の処理がデータ保全性を失うことなく可能になります。ワイド文字のリテラルおよびストリングは、UCS-2 文字（ユニコード CCSID 13488）として保管できます。これにより、国際使用のためのアプリケーションを開発する際のコード・ベース変換の必要性が最小限になります。ILE C/C++ コンパイラは、ユニコード文字ストレージをサポートします。

ワイド文字リテラルの表現

ワイド文字リテラルは、さまざまな方法で表現することができます。これらの表現およびその処理の仕方を以下に示します。

Trigraphs

リテラルとして使用される 3 文字表記文字は、その対応するユニコード文字に変換されます。次に例を示します。

```c
wchar_t *wcs = L" ??(";
```

は、以下と同等です。

```c
wchar_t wcs[] = {0x0020, 0x005B, 0x0000};
```

3 文字表記について詳しくは、「ILE C/C++ 解説書」を参照してください。
文字エスケープ・コード (Ya, Yb, Yf, Yn, Yr, Yt, Yv, Y', Y")
文字エスケープ・コードは、その対応するユニコード・エスケープ・コードに変換されます。次に例を示します。

```c
wchar_t* wcs = L"\t \n";
```

は、以下と同等です。

```c
wchar_t wcs[] = {0x0020, 0x0009, 0x0020, 0x000A, 0x0000};
```

数値エスケープ・コード (Yxnnn, Yooo)
数値エスケープ・コードは、ユニコードに変換されません。代わりに、リテラルの数値部分は保存され、16 進数形式または 8 進数形式で表現されるユニコード文字とみなされます。次に例を示します。

```c
wchar_t* wcs = L"\x4145";
```

は、以下と同等です。

```c
wchar_t wcs[] = {0x0020, 0x4145, 0x0000};
```

`wchar_t` ストリング・リテラルで `Yxnn` を指定することは、`Yx00nn` を指定することと同等です。

`0xFF` より大きい 16 進定数値は、通常無効とみなされます。*LOCALUCS2 オプションを設定すると、これが変更されて、`wchar_t` 型の 2 バイトの 16 進数初期化のみが可能になります。次に例を示します。

```c
wchar_t wc = L'\x4145'; /* Valid only with *LOCALUCS2 option, */
/* otherwise an out of bounds error */
/* will result. */
```

```c
char c = '\x4145'; /* Not valid due to size restriction. */
/* Error will result with or without */
/* specifying *LOCALUCS2 option. */
```

注: 数値の 16 進エスケープ・コードは、型のサイズ制限を超えているかどうか以外には検証されません。

**DBCS 文字**
16 進エスケープ・シーケンスとして入力された DBCS 文字は、ユニコードに変換されません。受け取られたたとおりに保管されます。

**ユニコード文字セット・サポートの有効化**
ユニコード文字セット・サポートを有効にするには、CTCMod/CRTCpMod または CRTBnDc/CRTBndCp コマンド行に LOCALETYPE(*LOCALUCS2) を指定します。このオプションを選択すると、__UCS2__ マクロが定義されます。ワイド文字リテラルは、ルート・ソース・ファイルの CCSID を使用して解釈されてから、保険時にユニコード CCSID (13488) に変換されます。

**#pragma convert() 演算でのユニコードの効果**
LOCALETYPE(*LOCALUTF) を指定すると、ワイド文字リテラルは、ルート・ソース・ファイルで使用される CCSID に関係なく、常に UTF-32 形式で表現されます。さらに、#pragma convert() は、ワイド文字リテラルに対しては効果がなくなります。

LOCALETYPE(*LOCALUCS2) を指定すると、ワイド文字リテラルは、ルート・ソース・ファイルで使用される CCSID に関係なく、常に UCS-2 文字リテラルを表現します。さらに、文字を別のコード・ページに変換するときには、#pragma convert() はワイド文字リテラルを無視します。
例:
この例では、CCSID 37 ソースを想定します。

#include <stdio.h>
#include <wchar.h>

void main () {
    #pragma convert (500)
    wchar_t wcs1[] = L"[]";
    char str1[] = "[]";
    #pragma convert (0)
    wchar_t wcs2[] = L"[]";
    char str2[] = "[]";
    printf("str1 = %x %x\n", str1[0], str1[1]);
    printf("str2 = %x %x\n", str2[0], str2[1]);
    printf("wcs1 = %04x %04x\n", wcs1[0], wcs1[1]);
    printf("wcs2 = %04x %04x\n", wcs2[0], wcs2[1]);
}

プログラムを実行すると、結果の出力は以下に示したようなものになります。

str1 = 4a 5a
str2 = ba bb
wcs1 = 005b 005d
wcs2 = 005b 005d

GB18030 コード・ページ・サポート
中国政府は、中国で販売されるすべての製品で GB18030 コード・ページのサポートを要求します。
GB18030 は、ユニコードに対するコード・ページおよびマッピング・テーブルを指定する中国語版標準です。

この標準の要点は以下のとおりです。
・ コード・ページはマルチバイト・エンコード方式を使用します。
・ ユニコードへの直接マッピングがあります (GB18030 で使用可能なすべてのコード・ポイントはユニコードにあります)。
・ 64 K を超える文字があります。

このことは、GB18030 文字は、ユニコードを使用して表現できるということを意味します。ここでは以下のもを使用します。
・ ナロー文字およびストリング・リテラルを表現する UTF-8
・ ワイド文字およびストリング・リテラル用の UTF-32
コンパイラが処理する基本文字セットは UTF-8 です。

ユーザー・ソースは UTF-8 でエンコードする必要はありません。コンパイラがこれを内部処理用に UTF-8 に変換します。

UTF-32 でのワイド文字およびストリング・リテラルの生成: ワイド文字およびストリング・リテラルを UTF-32 で生成するには、以下のもので LOCALETYPE(*LOCALEUTF) オプションを使用します。

・ C モジュール作成 (CRTCMOD) またはバインド済み C 作成 (CRTBNDC) のいずれかのコマンド
・ C++ モジュールの作成 (CRTCPPMOD) またはバインド済み C++ 作成 (CRTCNDCCPP) のいずれかのコマンド

9-6 IBM i: ILE C/C++ プログラマーの手引き
考慮事項：LOCALETYPE(*LOCALEUTF) を指定すると、ワイド文字リテラルは、ルート・ソース・ファイルで使用される CCSID に関係なく、常に UTF-32 形式で表記されます。さらに、#pragma convert() は、ワイド文字リテラルに対しては効果がありません。

LOCALETYPE(*LOCALEUTF) オプションの場合、ターゲット CCSID が 1208 でなければなりません。デフォルトまたは指定されたターゲット CCSID が 1208 にマップされない場合、以下のようなになります。
- オーバーライドでコンパイルを続行できます。
- 診断警告 C02218 が出されます。

改行文字 (Yn) は、SYSIFCOPT オプション（*IFS64IO || *IFSIO || *NOIFSIO）に関係なく、値 0x0a に変換されます。

ナロー文字の変換には、基本文字セットを超える値が含まれます。こうした文字の例に「〜」があります。これは、UTF-8 では 2 バイト値 0xC2AC を持つ。

LOCALETYPE(*LOCALEUTF) オプションを指定すると、コンパイラは「wchar_t」を 4 バイトのサイズで、位置合わせのある符号なし整数として事前定義します（それ以外の場合、「wchar-t」は 2 バイトのサイズで位置合わせのある符号なし短整数のままです）。

LOCALETYPE(*LOCALEUTF) オプションを指定すると、4 バイトのサイズで位置合わせのある符号なし整数の定義が使用されます。この定義は、<stdlib.h> で提供されます。

**CCSID のターゲット指定**

TGTCSSID パラメーターにより、コンパイラは以下のことを行うことができます。
- さまざまな CCSID またはコード・ページからソース・ファイルを処理する（ソース・ストリーム・ファイルの場合）。
- ルート・ソース・ファイルとは異なるモジュール CCSID をターゲットとする。ただし、ソース文字セットとターゲット・モジュール CCSID の間の変換がオペレーティング・システムにインストールされている必要があります。

ターゲット CCSID (TGTCSSID) は、以下の ILE C/C++ コマンドで使用されるパラメーターです。
- C モジュールの作成 (CRTCMOD)
- C++ モジュールの作成 (CRTCPPMOD)
- パインド済み C プログラムの作成 (CRTBNDC)
- パインド済み C++ プログラムの作成 (CRTBNDCPP)

**ILE C/C++ コンバイラーによるソース・ファイルのターゲット CCSID への変換方法**

TGTCSSID がソース・ファイルの CCSID と異なっている場合、ILE C コンパイラは、ソース・ファイルを TGTCSSID に変換してから、ファイルを処理します。これにより、ターゲット・モジュールおよびそのすべての文字データ・コンポーネント（リスト、ストリング・プールなど）が所要の TGTCSSID に入ります。そのため、1つの文字セットで開発し、別の文字セットをターゲットとすることができます。デフォルトの振る舞いが後方互換を持つように、引数はデフォルトでソース・ファイルの文字セットになります (290, 930, および 5026 を除く）。

注意：C++ は、ストリング・リテラルのみ（ソースではなく）を TGTCSSID に変換します。

さらに多くのソース文字セットをサポートするようにすると、コンパイラの NLS ユーザビリティが増します。現在 CCSID の 290, 930, および 5026 がサポートされています。TGTCSSID パラメーター
は、より複雑な NLS プログラミングの問題に対する解決策となります。例えば、異なる TGTCCSID 値を持つソースを再コンパイルするだけで、異なるモジュール CCSID を持ついくつかのモジュールを、同じソースからコンパイルできます。

リテラル、コメント、および ID

TGTCCSID パラメーターでは、結果のモジュールの CCSID を選択することができます。モジュールの CCSID は、モジュールの文字データが保管されるコード化文字セット ID を識別します。その文字データには、ソースによって記述されたリテラル、コメント、および ID 名を記述するために使用される文字データが含まれます (CCSID の 5026、930、および 290 の ID 名を除く)。

例えば、ルート・ソース・ファイルが 500 という CCSID を持ち、コンパイラ・パラメーター TGTCCSID のデフォルト値が変更されない (すなわち *SOURCE である) 場合、揺る舞いは 500 という結果のモジュール CCSID による場合と変わりません。すべてのストリング・リテラルおよび文字リテラル (単一文字とワイルド文字の両方) は、ソース・ファイルの CCSID で説明したとおりです。リテラル、コメント、および ID の場合も、変換は同様に行われます。

ただし、TGTCCSID パラメーターが 37 に設定されていて、同じソースが再コンパイルされると、結果のモジュール CCSID は 37 です。つまりすべてのリテラル、コメント、および ID は、モジュール内で 37 が必要でしかもそのように保管される場合は 37 に変換されます。

ルート・ソースおよび含まれるヘッダーの CCSID に関係なく、結果のモジュールは TGTCCSID によって定義され、そのリテラル、コメント、および ID はすべてこの CCSID に保管されます。

制限

デバッグ・リスト・ビュー

TGTCCSID パラメーターの導入により、CCSID の 5026、930、または 290 を持つソースのコンパイルを妨げている制約が取り除かれ、しかもリテラルおよびコメントで DBSC 文字が失われることはありませ ん。ただし、こうした CCSID には厳やかな制約が取り入れられています。リスト・ビューを使用して、CCSID 5026、930、または 290 に等しい TGTCCSID によってコンパイルされたモジュールをデバッグする際には、CCSID 37 と互換性がないすべての文字に対して置換文字が表示されます。

フォーマット・ストリング

C 実行時出力関数 (printf "%d\n", 1234); など）のフォーマット・ストリングをコーディングする場合、フォーマット・ストリングは CCSID 037 と互換性がなければなりません。 CCSID 037 互換でない CCSID の 290、930、5026 をターゲットとする場合、実行時関数がフォーマット・ストリングを正しく処理できるように、フォーマット・ストリング・リテラルの前後に #pragma convert(37) が必要です。

有効なターゲットのエンコード・スキーム

TGTCCSID 値は、エンコード・スキームが 1100 または 1301 の CCSID に制限されています。他の値が入力されると、コマンドによってエラー・メッセージが出されます。

1100 = EBCDIC、1 バイト、コード拡張は許可されない、状態番号 = 1。

1301 = EBCDIC、1 バイトと 2 バイトの混合、シフトイン (SI) とシフトアウト (SO) コード拡張メソッドを使用、状態番号 = 2。
国際ロケールのサポート

国際ロケールのサポートにより、プログラムは、ユーザーの言語環境に従って動作を変更できます。このサポートには、以下の3つのキー・コンポーネントがあります。

- 言語固有のデータを作成するプログラミング・ツール
- このデータへのアクセスを可能にするプログラミング・インタフェース（関数）
- 実行されている言語環境に自動的に依存したプログラムを作成するメソッド

言語環境のエレメント

言語環境の典型的なエレメントは以下のとおりです。

- ネイティブ言語:
  ユーザーの自然言語。
- 文字セットおよびコード化文字セット:
  コード化文字セットは、文字セットの文字をコード・ポイント（16進数値）のセットにマップすることによって作成されます。コード化文字セットおよびCCSIDについて詳しくは、「9.1ページの『プログラマムの国際化』」を参照してください。
- 照合と順序付け:
  ソートに使用される文字の相対順序。
- 文字種別:
  文字セット内の文字の型（英字、数字など）。
- 文字の大/小文字変換:
  文字セット内の大文字と小文字の間のマッピング。
- 日時形式:
  日付および時刻データの形式（月の順序や曜日名）。
- 数値および通貨数量の形式:
  数値および通貨数量の形式、数値グループ化、小数点文字、通貨記号など。
- 肯定および否定のシステム応答の形式:
  肯定および否定のシステム応答の形式。

ロケール

ロケールは、言語固有のデータがどのように処理、印刷、および表示されるかを指定するシステム・オプジェクトです。ロケールは、それが使用される言語環境の文字セット、照合シーケンス、日時表記、および通貨表記を記述するカテゴリから構成されます。ロケールおよびロケール依存関数を使用すると、言語、文化的データ、または文字セットに依存しないアプリケーションを作成できますが、ユーザーの言語環境には依存します。

ロケール用の ILE C/C++ サポート

ILE C のコンパイラおよび実行時環境は、*CLD、*LOCALC、*LOCALUCS2、および*LOCALUTF というタイプのロケールをサポートします。

国際ロケールおよびコード化文字セットの使用 9-9
ILE C++ のコンバイラーおよび実行時環境は、*LOCALE、*LOALEUCS2、および *LOCALEUTF というタイプのロケールをサポートします。


POSIX 標準は、アプリケーション国際化対応のために、*CLD ロケールに使用可能なものと比べて関数およびロケール・データの極めて包含的なセットを定義します。ILE C/C++ 実行時ライブラリ内のロケールに対して POSIX 仕様をサポートし、XPG4、POSIX、および ISO/IEC 標準に準拠する新規関数を導入すると、タイプ *LOCALE のロケールを使用する ILE C/C++ プログラムは、他のオペレーティングシステムとの間のポートビリティが増します。

注：CLD は ILE C 専用です。

*CLD および *LOCALE オブジェクト・タイプ用の ILE C/C++ サポート

V3R7 以前にコンバイルされたプログラムは、*CLD ロケール・サポートを使用します。オプション LOCALETERROR(*CLD) を指定した CRTCMOD または CRTBNDC コマンドでコンバイルされたプログラムは、*CLD オブジェクト用に ILE C/C++ によって提供されるロケール・サポートを使用します。オプション LOCALETERROR(*LOCALE) を指定した CRTCMOD/CRTCPPMOD または CRTBNDC/CRTBNDCP コマンドでコンバイルされたプログラムは、タイプ *LOCALE ロケール用に ILE C/C++ によって提供されるロケール・サポートを使用します。

注：*C++ は *CLD オブジェクトをサポートしません。

タイプ *CLD のロケールを使用するアプリケーションをタイプ *LOCALE のロケールを使用するものに変換したい場合、C ソース・コードでは setlocale() の呼び出しを変更するだけで済みます。ただし、*CLD オブジェクトと *LOCALE オブジェクトのロケール定義ソースの間には多くの相違があります。多くの言語環境の *LOCALE 定義ソース・メンバーは、オプションでインストール可能なライブラリ QSYSLOCALE 内にシステムによって提供されます。既存の*CLD ロケール・ソースを *LOCALE ソース定義に変換することもできます。*CLD オブジェクトを作成するためのソース・ファイル内のコマンドを、*LOCALE オブジェクトを作成するためのソース・ファイル内の対応するキーワードにマップすることについては、「9-11 ページの表 9-11」を参照してください。

アプリケーションは *CLD か *LOCALE のいずれかのタイプのロケールを使用できますが、両方は使用できません。ILE C プログラムが両方のタイプのロケールを使用しようとすると、結果は未定義のものになります。ILE C++ は *CLD を使用しません。またロケール依存関数の中には、タイプ *LOCALE のロケールが使用される場合にしかサポートされないものもあります。ロケール依存関数のリストについては、「9-16 ページの表 9-16」を参照してください。

C ロケール・マイグレーション・テーブル

ILE C ランタイムは、setlocale() 関数およびロケール依存関数の 2 つの実装環境をサポートします。オリジナルの実装環境ではタイプ *CLD のロケール・オブジェクトが使用され、第 2 の実装環境ではタイプ *LOCALE のロケール・オブジェクトが使用されます。以下の表では、タイプ *CLD とタイプ *LOCALE のロケールの間のロケール・ソース・キーワードにおける相違点を要約しています。
表9-1. C ロケール・マイグレーション・テーブル

<table>
<thead>
<tr>
<th>カテゴリー</th>
<th>*CLD コマンド</th>
<th>説明と形式</th>
<th>*LOCALE キーワード</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_TIME</td>
<td>AM</td>
<td>AM のロケール同等の表現に使用される文字ストリング。</td>
<td>am_pm</td>
</tr>
<tr>
<td>LC_CTYPE</td>
<td>CHARTYP</td>
<td>文字セット・タイプ。</td>
<td>ロケールの CCSID により決定</td>
</tr>
<tr>
<td>LC_COLLATE</td>
<td>CPYSYSCOL</td>
<td>システム照合シーケンス表。</td>
<td>cpysycol</td>
</tr>
<tr>
<td>LC_COLLATE</td>
<td>COLLSTR</td>
<td>ストリング変換テーブル</td>
<td>collating-element</td>
</tr>
<tr>
<td>LC_COLLATE</td>
<td>COLLTAB</td>
<td>関数の文字の重み再割り当て</td>
<td>collating-element</td>
</tr>
<tr>
<td>LC_CTYPE</td>
<td>CTYPE</td>
<td>特定の文字の属性を設定します。</td>
<td>upper lower alpha digit space cntrl punct graph print xdigit blank</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>CURR</td>
<td>通貨記号を表す文字ストリング</td>
<td>currency_symbol</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>DATFMT</td>
<td>このロケールで日付の形式を指定するために使用される文字ストリング。</td>
<td>d_fmt</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>DATTIM</td>
<td>このロケールで日時の形式を指定するために使用される文字ストリング。</td>
<td>d_t_fmt</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>DEC</td>
<td>定義の非通貨数の小数点文字。</td>
<td>decimal_point</td>
</tr>
<tr>
<td>LC_TOD</td>
<td>DSTEND</td>
<td>夏時間調整時間が終了する時点。(日、時間)</td>
<td>dstend</td>
</tr>
<tr>
<td>LC_TOD</td>
<td>DSTNAME</td>
<td>夏時間調整時間が有効なときの時間帯の名前。</td>
<td>dstname</td>
</tr>
<tr>
<td>LC_TOD</td>
<td>DSTSHIFT</td>
<td>夏時間調整時間が有効になるときにロケールの時間がシフトされる秒数。</td>
<td>dstshift</td>
</tr>
<tr>
<td>LC_TOD</td>
<td>DSTSTART</td>
<td>夏時間調整時間が有効になる時点。</td>
<td>dststart</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>GROUP</td>
<td>定義の非通貨数の小数点の左側の数字の桁区切り。</td>
<td>grouping</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>ICURR</td>
<td>国際的定義の通貨数で通貨記号の表現に使用される文字ストリング。</td>
<td>int_curr_symbol</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>LDAVS</td>
<td>週の各曜日の長形式。</td>
<td>day</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>LMONNS</td>
<td>年の各月の長形式。</td>
<td>mon</td>
</tr>
<tr>
<td>LC_CTYPE</td>
<td>LOWER</td>
<td>tolower ライブラリー関数で指定の文字に小文字が戻されるように設定します。</td>
<td>tolower</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>MDEC</td>
<td>定義の通貨数の小数点に使用される文字ストリング。</td>
<td>mon_decimal_point</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>MFDIGIT</td>
<td>定義の通貨数で表示される小数桁数。</td>
<td>frac_digits</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>MGROUP</td>
<td>定義の通貨数での小数点の左側の数字の桁区切り。</td>
<td>mon_grouping</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>MFDIGIT</td>
<td>国際的定義の通貨数で表示される小数桁数。</td>
<td>int_frac_digits</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>MMINUS</td>
<td>負の定義の通貨数で負の値の表現に使用される文字ストリング。</td>
<td>negative_sign</td>
</tr>
</tbody>
</table>
### 表9-1. C ロケール・マイグレーション・テーブル (続き)

<table>
<thead>
<tr>
<th>カテゴリー</th>
<th>*CLD コマンド</th>
<th>説明と形式</th>
<th>*LOCALE キーワード</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_MONETARY</td>
<td>MMINUSPOS</td>
<td>負の定義される通貨数量で負符号の位置を</td>
<td>n_sign_posn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>表現に使用されるエンコード値。</td>
<td></td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>MNCSP</td>
<td>負の定義される通貨数量で値に通貨記号が</td>
<td>n_cs_precedes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>先行するかどうかの判定に使用されるtrue</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>またはfalse値。値がfalseの場合、記号は値の後にきます。</td>
<td></td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>MNSBYS</td>
<td>負の定義される通貨数量で通貨記号がスペース区切りであるかどうかの判定に使</td>
<td>n_sep_by_space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>用されるtrueまたはfalse値。値がfalseの場合、記号は値の後にきます。</td>
<td></td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>MPLUS</td>
<td>正の定義される通貨数量で正の値の表現に</td>
<td>positive_sign</td>
</tr>
<tr>
<td></td>
<td></td>
<td>使用される文字ストリング。</td>
<td></td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>MPLUSPOS</td>
<td>正の定義される通貨数量で正符号の位置を</td>
<td>p_sign_posn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>表現に使用されるエンコード値。</td>
<td></td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>MPSBYS</td>
<td>正の定義される通貨数量で通貨記号がスペース区切りであるかどうかの判定に使</td>
<td>p_sep_by_space</td>
</tr>
<tr>
<td></td>
<td></td>
<td>用されるtrueまたはfalse値。値がfalseの場合、記号は値の後にきます。</td>
<td></td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>MSEP</td>
<td>定義される通貨数量で桁を区切るために使用される文字。</td>
<td>mon_thousands_sep</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>PM</td>
<td>PMのロケール同様の表現に使用される文字ストリング。</td>
<td>am_pm</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>SDAWS</td>
<td>週の各曜日の略形式。</td>
<td>abday</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>SEP</td>
<td>定義される通貨数量で桁を区切るために使用される文字。</td>
<td>decimal_point</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>SMONS</td>
<td>年の各月の略形式。</td>
<td>abmon</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>TIMFMT</td>
<td>このロケールで時刻の形式を指定するために使用される文字ストリング。</td>
<td>t_fmt</td>
</tr>
<tr>
<td>LC_TOD</td>
<td>TNAME</td>
<td>ロケールの時間帯名の表現に使用される文字ストリング。</td>
<td>tname</td>
</tr>
<tr>
<td>LC_TOD</td>
<td>TDIFF</td>
<td>ロケールの時間帯のグリニッジ標準時からの時差を示す分数。</td>
<td>tdiff</td>
</tr>
<tr>
<td>LC_CTYPE</td>
<td>UPPER</td>
<td>toupperライブラリ関数で指定の文字</td>
<td>toupper</td>
</tr>
</tbody>
</table>

### POSIX ロケール定義および *LOCALE サポート

IEEE POSIX P1003.2 標準に準拠しているロケール定義ソース・ファイルが、システムとともに送付され、オプションでインストール可能なライブラリ QSYSLOCALE 内にあります。1つの *LOCALE オブジェクト (POSIX 標準によって定義された C ロケール) がシステムによって提供されます。タイプ *LOCALE のその他のロケールは、CRTLOCALE コマンドを使用して、QSYSLOCALE ライブラリ内のロケール・ソース定義メンバーから作成できます。
**LOCALETYPE コンバイラー・オプション**

CRTCMOD/CRTCPMPMOD または CRTBNDC/CRTBNDCPP コマンドの LOCALETYPE オプションにより、プログラムはそのコンバイルの際に使用されるロケール・オブジェクトのタイプを指定できます。
LOCALETYPE オプションのキーワード・オプションは *CLD、*LOCALE、*LOCALEUCS2、および *LOCALEUTF で、デフォルトは *LOCALE です。キーワード *CLD では *CLD ロケール・サポートが有効になり、他のキーワードではタイプ *LOCALE のロケールのサポートが有効になります。

タイプ *LOCALE のロケールをサポートする実行時環境を有労にするためのコマンド形式は以下のとおりです。

```
CRTMOD MODULE(MYLIB/MYMOD) SRCFILE(MYLIB/QCSRC) LOCALETYPE(*LOCALE)
CRTBNDC PGM(MYLIB/MYPGM) SRCFILE(MYLIB/QCSRC) LOCALETYPE(*LOCALE)
```

LOCALETYPE オプションに *CLD キーワードを指定しないと、ILE C/C++ コンバイラーはマクロ __POSIX_LOCALE__ を定義します。 __POSIX_LOCALE__ が定義されると、ロケール依存 C 実行時間数は、*LOCALE オブジェクトに定義されているロケールに依存する関数をリマップされます。さらに、一定の ILE C/C++ 実行時間数は、タイプ *LOCALE のロケールでしか使用できず、*CLD ロケールでは機能しません。こうした関数は、V3R7 以降のリリースの ILE C/C++ ランタイムでしか使用できません。

9-16 ページの表 9.3 のロケール依存関数のリストは、タイプ *LOCALE のロケールにのみ依存する関数を示しています。

注: デフォルトは変更されました。 V5R1 以前では、*CLD が ILE C のデフォルト値でした。V5R1 現在、デフォルトは *LOCALE に変更されています。

**ロケールの作成**

IBM i プラットフォームでは、*LOCALE オブジェクトは、ロケールの定義ソースを含むファイル・メンバーの名前を指定し、さらにロケールの文字セットの文字をそれぞれの 16 進値にマップするために使用される CCSID を指定して、CRTLOCAL コマンドで作成されます。

ロケール定義ソース・メンバーには、言語環境に関する情報が含まれます。この情報は、次のセクションで説明する多種のカテゴリに分けられます。1 つのロケール定義ソース・メンバーが 1 つの言語環境の特性を表します。

文字は、ロケール定義ソース・メンバー内ではそれぞれのシンボル名で表されます。シンボル名、それが表す文字、およびそれぞれの関連付けられた 16 進値の間のマッピングは、CRTLOCAL コマンドに指定された CCSID 値に基づいて行われます。

タイプ *LOCALE のロケールがどのように作成されるかのモデルを以下に示します。
図9-3 タイプ *LOCALE のロケールの作成

注: 非 EBCDIC の CCSID で作成されたロケールに対するサポートはありません。

LOCALETYP (*LOCALE) を使用したモジュールの作成
LOCALETYP (*LOCALE) オプションでモジュールを作成すると、MB_CUR_MAX は以下の値になります。

- 1 (00037 などの 1 バイト CCSID でビルト出されたロケールの場合)
- 4 (00939 などの混合バイト CCSID でビルト出されたロケールの場合)

MB_CUR_MAX は、現行ロケールの LC_CTYPE カテゴリーに依存します。

ロケールで使用されるカテゴリ
ロケールおよびその定義ソース・メンバーには、以下のカテゴリが含まれています。

表9-2 ロケールで使用されるカテゴリ

<table>
<thead>
<tr>
<th>カテゴリー</th>
<th>目的</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_COLLATE</td>
<td>文字間の照合関係を定義します。照合関数 <code>strcoll()</code>、<code>strxfrm()</code>、<code>wcsoll()</code>、および <code>wcsxfrm()</code> の動作に影響します。</td>
</tr>
<tr>
<td>LC_CTYPE</td>
<td>大文字、小文字、スペース、数字、句読点といった文字タイプを定義します。文字処理関数の動作に影響します。</td>
</tr>
<tr>
<td>LC_MESSAGES</td>
<td>アプリケーションからの応答のフォーマットおよび値を定義します。</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>通貨名、記号、句読点、その他の詳細を定義します。 <code>localeconv()</code> で返される通貨情報に影響します。</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>フォーマット設定された入出力およびストリーミング変換関数の小数点 (radix) 文字、および <code>localeconv()</code> によって返される非通貨フォーマット設定情報および通貨情報に影響します。</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>使用されるカレンダー、時間帯、および曜日といった日付と時刻の規則を定義します。時間表示関数の動作に影響します。</td>
</tr>
<tr>
<td>LC_TOD</td>
<td>時間帯の差、時間帯名、および夏時間調整時間開始/終了を定義します。</td>
</tr>
</tbody>
</table>

アプリケーションのアクティブ・ロケールの設定
タイプ *LOCALE のロケールを使用するすべての C および C++ アプリケーションは、そのプログラムの活動状態を有効範囲とするアクティブ・ロケールを持っています。アクティブ・ロケールは、C ライブラリー内のロケール依存関数の振る舞いを決定します。アクティブ・ロケールは、`setlocale()` の呼び
出しによって明示的に設定することができます。 setlocale() の使用について詳しくは、「ILE C/C++ ランタイム・ライブラリ関数」を参照してください。

アクティブ・ロケールは、setlocale() の呼び出しによって明示的に設定されていない場合、プログラムを動活化するときに C 実行時環境によって暗黙的に設定されます。以下にプログラムが活動化されるときに、実行時環境によってアクティブ・ロケールがどのように設定されるかを示します。

・ユーザー・プロファイルの LOCALE バラメーターが *NONE (デフォルト) または *SYSVAL 以外の値である場合、アプリケーションのアクティブ・ロケールにその値が使用されます。
・ユーザー・プロファイルの LOCALE バラメーターの値が *NONE である場合、デフォルトの「C」ロケールがアクティブ・ロケールになります。
・ユーザー・プロファイルの LOCALE バラメーターの値が *SYSVAL である場合、システム値 QLOCALE に関連付けられたロケールがプログラムのアクティブ・ロケールとして使用されます。
・QLOCALE の値が *NONE の場合、デフォルトの「C」ロケールがアクティブ・ロケールになります。

アクティブ・ロケールを設定するための環境変数の使用
プログラムのアクティブ・ロケールは、プログラムの開始時に暗黙的に設定されるか (既に説明済み)、setlocale() の呼び出しによって明示的に設定されます。 setlocale() 関数は 2 つの引数を取ります。1 つは、アクティブ・ロケールに必要な値を持つロケール・カテゴリを表す数値で、1 つはその値が取り出されるロケールの名前です。ロケールの名前は、以下のいずれかとすることが可能です。

・C
・POSIX
・タイプ *LOCALE のロケール・オブジェクトの完全修飾パス名
・ヌル・ストリング ("")

setlocale() のロケール引数がヌル・ストリング (""") として指定されている場合、setlocale() は、プログラムが実行中であるジョブに定義された環境変数に従ってアクティブ・ロケールを設定します。ロケール・カテゴリと同じ名前の環境変数を作成して、各環境変数に関連付けられるロケールを指定することができます。ロケール・パス名をユーザー・プロファイルの LOCALE バラメーターに指定するか、または QLOCALE システム値に指定すると、LANG 環境変数が、ジョブの開始時に自動的に作成されます。

プログラムが setlocale(category, ")") を呼び出すと、指定されたカテゴリに使用される 1 つ以上のロケール名を見つけるために、現行ジョブに定義されたロケール関連の環境変数がチェックされます。ロケール名は、適用される次の条件の最初のものに従って選択されます。

1. ヌルでない環境変数 LC_ALL が定義されている場合、LC_ALL の値が指定されたカテゴリに使用されます。指定されたカテゴリが LC_ALL の場合、その値がすべてのカテゴリに適用されます。
2. カテゴリーにヌルでない環境変数が定義されている場合、その環境変数に指定された値が使用されます。 LC_ALL カテゴリーの場合、ヌルでない個別の環境変数 (LC_CTYPE や LC_MONETARY など) が定義されていると、環境変数に対応するカテゴリにその値が使用されます。これにより、別のロケール・オブジェクトから検索される、各カテゴリのロケール情報が作成されることになります。
3. ヌルでない環境変数 LANG が定義されている場合、その LANG 環境変数の値が使用されます。
4. ロケール値を提供する非ヌルの環境変数が存在しない場合、デフォルトの C ロケールが使用されます。

環境変数に指定されたロケールが無効であるか、存在しないと判明した場合、setlocale() はヌルを戻し、プログラムのアクティブ・ロケールは変更されません。
setlocale(LC_ALL, "") の場合、見つかったロケール名がシステム上の有効なロケールを示していると、
setlocale() は、各ロケール・カテゴリーに関連したロケールの名前を示すストリングを戻します。それ以外の場合、setlocale() はヌルを戻し、プログラムのロケールは変更されません。

SAA および POSIX *LOCALE 定義

ILE C プログラムが LOCALETYP イ *LOCALE でコンパイルされた場合、setlocale() が呼び出されないか、またはロケール名 C または POSIX で呼び出されると、使用されるデフォルトの C 環境は、
QSYSLOCALE ライブラリー内の POSIX ロケール定義ソースに指定されたものです。このロケール定義は、タイプ *CLD のデフォルトの C ロケールとは若干異なります。デフォルトのタイプ *CLD の C ロケールとの互換性維持のために、QSYSLOCALE ライブラリーには SAA と呼ばれる別のロケール定義ソース・メンバーが提供されています。

アプリケーションをタイプ *CLD のロケールからタイプ *LOCALE のロケールにマイグレーションするが、デフォルトのタイプ *CLD の C ロケールと互換性があるようにしたい場合、QSYSLOCALE ライブラリー内の SAA ロケール定義ソース・メンバーを使用し、CRTLOCALE コマンドでロケールを作成します。次に、アプリケーションで setlocale() を呼び出すときに、このロケールの名前を使用します。

SAA と POSIX の間のロケール定義の相違は、以下のとおりです。

- LC_CTYPE カテゴリーの場合、SAA ロケールには、cntrl クラスに定義されたすべての EBCDIC 制御文字があります。これに対し、POSIX ロケールには ASCII 制御文字しか含まれていません。また SAA には句読文字としてセント文字および縦破線がありますが、これに対し、POSIX ではこれらの 2 つの文字は句読文字に含まれません。

- LC_COLLATE カテゴリーの場合、SAA のデフォルトの照合シーケンスは EBCDIC シーケンスです。これに対し、POSIX では ASCII シーケンスが使用されます。これは、文字セットの CCSID マッピングには関係ありません。POSIX ロケールの場合、最初の 128 個の ASCII 文字が照合シーケンスに定義され、残りの EBCDIC 文字は照合シーケンスの終わりに置かれます。また、SAA ロケール定義では、小文字の照合は大文字の前です。これに対し、POSIX ロケール定義では小文字の照合は大文字の後です。

- LC_TIME カテゴリーの場合、SAA ロケールは日時形式 (d_t_fmt) を "%Y/%M/%D %X" として指定します。これに対し、POSIX ロケールは "%a %b %d %H %M %S %Y" を使用します。

ロケール依存実行時関数

以下の ILE C/C++ 実行時関数は、ロケール依存です。

表9-3. ロケール依存実行時関数

<table>
<thead>
<tr>
<th>処理</th>
<th>海外</th>
<th>変換</th>
<th>機能</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime()</td>
<td>asctime_r()</td>
<td>btowc()¹</td>
<td></td>
</tr>
<tr>
<td>ctime()</td>
<td>ctime_r()</td>
<td>fprintf()</td>
<td></td>
</tr>
<tr>
<td>fgetwc()¹</td>
<td>fgetws()¹</td>
<td>fputwc()¹</td>
<td></td>
</tr>
<tr>
<td>fputws()¹</td>
<td>fwprintf()¹</td>
<td>fwscanf()¹</td>
<td></td>
</tr>
<tr>
<td>getwc()¹</td>
<td>getwchar()¹</td>
<td>gtime()</td>
<td></td>
</tr>
<tr>
<td>gmtime_r()</td>
<td>isalnum() to isxdigit()</td>
<td>isascii()</td>
<td></td>
</tr>
<tr>
<td>iswalnum() から iswxdigit()¹</td>
<td>localeconv()</td>
<td>localtime()</td>
<td></td>
</tr>
<tr>
<td>localtime_r()</td>
<td>mblen()</td>
<td>mbtowc()¹</td>
<td></td>
</tr>
<tr>
<td>mbsinit()¹</td>
<td>mbtowcs()¹</td>
<td>mbsrtowcs()¹</td>
<td></td>
</tr>
<tr>
<td>mbstowcs()¹</td>
<td>mbtowc()¹</td>
<td>mktime()</td>
<td></td>
</tr>
<tr>
<td>nl_langinfo()¹</td>
<td>printf()</td>
<td>putwc()¹</td>
<td></td>
</tr>
</tbody>
</table>
表9-3. ロケール依存実行時間数 (続き)

<table>
<thead>
<tr>
<th>処理</th>
<th>処理</th>
<th>処理</th>
</tr>
</thead>
<tbody>
<tr>
<td>putwchar()</td>
<td>regcomp()</td>
<td>regerror()</td>
</tr>
<tr>
<td>regexec()</td>
<td>regfree()</td>
<td>setlocale()</td>
</tr>
<tr>
<td>sprintf()</td>
<td>strcoll()</td>
<td>strfmon()</td>
</tr>
<tr>
<td>strftime()</td>
<td>strptime()</td>
<td>strxfrm()</td>
</tr>
<tr>
<td>swprintf()</td>
<td>swscanf()</td>
<td>time()</td>
</tr>
<tr>
<td>toascii()</td>
<td>tolower()</td>
<td>toupper()</td>
</tr>
<tr>
<td>towlower()</td>
<td>towtrant()</td>
<td>towupper()</td>
</tr>
<tr>
<td>vfprintf()</td>
<td>vfwprint()</td>
<td>vprintf()</td>
</tr>
<tr>
<td>vsprintf()</td>
<td>vswprintf()</td>
<td>vswprintf()</td>
</tr>
<tr>
<td>wcstod()</td>
<td>wcstol()</td>
<td>wcstoul()</td>
</tr>
<tr>
<td>wcstombs()</td>
<td>wcstombs()</td>
<td>wcswidth()</td>
</tr>
<tr>
<td>wcxfrm()</td>
<td>wctob()</td>
<td>wctomb()</td>
</tr>
<tr>
<td>wcxfrm()</td>
<td>wctype()</td>
<td>wcwidth()</td>
</tr>
<tr>
<td>wscanf()</td>
<td>wprintf()</td>
<td></td>
</tr>
</tbody>
</table>

注: 1 *LOCALE でオブジェクトのみに依存
GENCSRC ユーティリティーと #pragma mapinc ディレクティブ

C/C++ ソースの生成 (GENCSRC) ユーティリティーを使用してデータベース・ヘッダー・ファイルを生成することで、以下を行えます。
- データ記述仕様 (DDS) 情報の抽出
- プログラムで使用する宣言を含むヘッダー・ファイルの作成

GENCSRC では、C/C++ プログラムで使用する外部記述ファイル情報を取得する手段を提供します。このユーティリティーでは、インクルード・ファイルの型定義構造を含む C/C++ ヘッダー・ファイルを作成します。

#pragma mapinc ディレクティブは、GENCSRC コマンドを使用して、DDS ファイルをインクルード・ファイルに直接変換する機能を提供します。

GENCSRC ユーティリティーと #pragma mapinc ディレクティブを比較すると、以下のようになります。
- GENCSRC は、IFS ファイル・システムまたはデータベース・ファイル・システムでインクルード・ファイルを生成できます。一方、#pragma mapinc は、データベース・ファイル・システムのみでヘッダーを生成します。
- GENCSRC のインクルード・ファイルは、どこにでも永久に配置することができます。一方、#pragma mapinc は、生成されたインクルード・ファイルを QTEMP ライブラリーに書き込みます。

次の表は、#pragma mapinc のオプションと GENCSRC のキーワードの比較を示しています。個々のオプションの詳細については、「ILE C/C++ コンパイラー参考」 内の #pragma mapinc の説明を参照してください。

<table>
<thead>
<tr>
<th>GENCSRC のキーワード</th>
<th>#pragma mapinc のオプション</th>
<th>説明</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRCFILE</td>
<td>member_name</td>
<td>ソース・プログラムの #include ディレクティブで参照するファイルの名前。出力ファイルはファイル・システムに生成されます。</td>
</tr>
<tr>
<td>SRCMBR</td>
<td>member_name</td>
<td>ヘッダー情報を含むメンバーの名前。これは、IBM i の命名規則に従います。出力ファイルはファイル・システムに生成されます。</td>
</tr>
<tr>
<td>OBJ</td>
<td>file_name</td>
<td>QSYS ファイル・システムでマップするオブジェクトのベース名。</td>
</tr>
<tr>
<td>SRCSTMF</td>
<td>member_name</td>
<td>出力ファイルは IFS ファイル・システムに生成されます。</td>
</tr>
<tr>
<td>RCDFMT</td>
<td>format_name</td>
<td>プログラムにインクルードされる DDS レコード様式を示します。デフォルトは *ALL です。</td>
</tr>
</tbody>
</table>
表10-1. GENCSRC のキーワードと #pragma mapinc のオプションの比較 (続き)

<table>
<thead>
<tr>
<th>GENCSRC のキーワード</th>
<th>#pragma mapinc のオプション</th>
<th>説明</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLTFLD</td>
<td>オプション</td>
<td>以下の値の組み合わせに制限されます。</td>
</tr>
<tr>
<td></td>
<td></td>
<td>具体的に</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PKDDECFLD</td>
<td>d または p</td>
<td>*DECIMAL または *CHAR</td>
</tr>
<tr>
<td>STRUCTURE</td>
<td>_P</td>
<td>*PACKED または *NONPACKED</td>
</tr>
<tr>
<td>ONEBYTE</td>
<td>1BYTE_CHAR</td>
<td>*ARRAY または *CHAR</td>
</tr>
<tr>
<td>UNIONDFN</td>
<td>union_name</td>
<td>*OBJ、NONE、または union_name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>注: *OBJ がデフォルトです。デフォルトの型定義共用体名は FILE_t で、 #pragma mapinc ディレクティブで説明された LIBRARY_FILE_FMT_both_t ではありません。</td>
</tr>
<tr>
<td>TYPEDEFPFX</td>
<td>prefix_name</td>
<td>*OBJ または <em>NONE</em></td>
</tr>
</tbody>
</table>

注: DDS と C/C++ のデータ・タイプ・マッピングについては、「ILE C/C++ コンパイラー参照」を参照してください。

注: DDS と C/C++ のデータ・タイプ・マッピングについては、「ILE C/C++ コンパイラー参照」を参照してください。
言語間のデータ・タイプ互換性

各高水準言語 (HLL) には、異なるデータ・タイプがあります。複数の言語で作成されているプログラム間でデータを受け渡す場合は、それらの違いについて熟知している必要があります。

ILE C++ プログラミング言語の一部のデータ・タイプは、他の言語に直接相当するものがあります。
ILE C++ データ・タイプを使用して、他の言語のデータ・タイプをシミュレートできます。

注:
1. C のデータ・タイプ互換性表は示されていません。C と C++ のデータ・タイプはパック 10 進数データ・タイプを除いて同じであるため、C++ の表を使用できます。ILE C/C++ でのパック 10 進数データの使用について詳しくは、以下を参照してください。
   * 8-126 ページの「C プログラムでのパック 10 進数データの使用」
   * 8-139 ページの「C++ プログラムでのパック 10 進数データの使用」
2. C++ で、パック 10 進数データ・タイプは、bcd クラスとして実装されています。ILE C のパック 10 進数データ・タイプと C++ の 2 進化 10 進数クラスは、バイナリー互換です。

以下の表を示します。
* 表11-1 は、ILE CL に対する ILE C++ データ・タイプの互換性を示しています。
* 11-2 ページの表11-2 は、CL に対する ILE C++ データ・タイプの互換性を示しています。
* 11-3 ページの表11-3 は、ILE RPG に対する ILE C++ データ・タイプの互換性を示しています。
* 11-5 ページの表11-4 は、OPM RPG に対する ILE C++ データ・タイプの互換性を示しています。
* 11-7 ページの表11-5 は、ILE COBOL に対する ILE C++ データ・タイプの互換性を示しています。
* 11-8 ページの表11-6 は、OPM COBOL に対する ILE C++ データ・タイプの互換性を示しています。

表11-1 は、ILE CL に対する ILE C++ データ・タイプの互換性を示しています。

<table>
<thead>
<tr>
<th>プロトタイプにおける ILE C++ 宣言</th>
<th>CL</th>
<th>長さ</th>
<th>コメント</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char[n]</code></td>
<td>*CHAR LEN(&amp;N)</td>
<td>n</td>
<td>文字の配列 (n は、1 から 32766 まで)。ヌル終了ストリング。CHGVAR &amp;V VALUE(&amp;V *TCAT X'00')。ここで、&amp;V1 は、&amp;V より 1 バイト大きい値です。</td>
</tr>
<tr>
<td><code>char *</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_Packed struct (short i; char[n])</td>
<td>*LGL</td>
<td>1</td>
<td>'1' または '0' を保持します。</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n+2</td>
<td>可変長フィールド。i は指定した長さで、n は最大長です。</td>
</tr>
<tr>
<td>整数型</td>
<td>*INT LEN(&amp;N)</td>
<td>2, 4, 8</td>
<td>2, 4、または 8 バイトの符号付き整数。(CL では、1 バイトの整数タイプはサポートされません)</td>
</tr>
<tr>
<td></td>
<td>*UINT LEN(&amp;N)</td>
<td>2, 4, 8</td>
<td>2, 4、または 8 バイトの符号なし整数。(CL では、1 バイトの符号なし整数タイプはサポートされません)</td>
</tr>
<tr>
<td>浮動小数点数</td>
<td>CL 定数のみ</td>
<td>4</td>
<td>4 バイトまたは 8 バイトの浮動小数点</td>
</tr>
</tbody>
</table>
表11-1. ILE CL に対する ILE C++ データ・タイプの互換性 (続き)

<table>
<thead>
<tr>
<th>プロトタイプにおける ILE C++ 宣言</th>
<th>CL</th>
<th>長さ</th>
<th>コメント</th>
</tr>
</thead>
</table>
| _DecimalT<n,p>                | *DEC | n/2+1 | バック 10 進数。 n の限度は 15 で、p の限度は 9 です。 C++ では、これは、2 進化 10 進数クラスであり、データ・タイプではありません。
| union.element                 | サポートなし。 | | 最長共用体メンバーよの長さ共用体のエレメント |

構造体またはクラス

| サポートなし。 | n | 構造体、構造体はバックする必要があります。
| | | 1 構造体の内容を変更する場合、受け渡す構造体はポインタとして構造体に渡す必要があります。 |

関数のポインター

| サポートなし。 | 16 | 16 バイトのポインター |

表11-2 は、CL に対する ILE C++ データ・タイプの互換性を示しています。

表11-2. CL に対する ILE C++ データ・タイプの互換性

<table>
<thead>
<tr>
<th>プロトタイプにおける ILE C++ 宣言</th>
<th>CL</th>
<th>長さ</th>
<th>コメント</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n]</td>
<td>*CHAR LEN(&amp;N)</td>
<td>n</td>
<td>文字の配列 (n は、1 から 32766 まで)。ヌル終了ストリング。CHGVAR &amp;V1 VALUE(&amp;V *TCAT X'00')。ここで、&amp;V1 は、&amp;V より 1 バイト大きい値です。n の限度は 9999。</td>
</tr>
<tr>
<td>char *</td>
<td>*LGL</td>
<td>1</td>
<td>'1' または '0' を保持します。</td>
</tr>
<tr>
<td>_Packed struct</td>
<td>サポートなし。</td>
<td>n+2</td>
<td>可変長フィールド。i は指定した長さで、n は最大長です。</td>
</tr>
<tr>
<td>(short i; char[n])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>整数型</td>
<td>*INT LEN(&amp;N)</td>
<td>2, 4</td>
<td>2 または 4 バイトの符号付き整数。(CL では、1 バイトの整数タイプはサポートされません)</td>
</tr>
<tr>
<td>*UINT LEN(&amp;N)</td>
<td>2, 4</td>
<td>2 または 4 バイトの符号なし整数。(CL では、1 バイトの符号なし整数タイプはサポートされません)</td>
<td></td>
</tr>
<tr>
<td>浮動小数点定数</td>
<td>CL 定数のみ</td>
<td>4</td>
<td>4 バイトまたは 8 バイトの浮動小数点</td>
</tr>
<tr>
<td>_DecimalT&lt;n,p&gt;</td>
<td>*DEC</td>
<td>n/2+1</td>
<td>バック 10 進数。 n の限度は 15 で、p の限度は 9 です。C++ では、これは、2 進化 10 進数クラスであり、データ・タイプではありません。</td>
</tr>
<tr>
<td>union.element</td>
<td>サポートなし。</td>
<td></td>
<td>最長共用体メンバーよの長さ共用体のエレメント</td>
</tr>
</tbody>
</table>

構造体またはクラス

| サポートなし。 | n | 構造体、構造体はバックする必要があります。
| | | 1 構造体の内容を変更する場合、受け渡す構造体はポインタとして構造体に渡す必要があります。 |

関数のポインター

| サポートなし。 | 16 | 16 バイトのポインター |
表11-3 は、ILE RPG に対する ILE C++ データ・タイプの互換性を示しています。

<table>
<thead>
<tr>
<th>プロトタイプにおける ILE C++ 宣言</th>
<th>フリー・フォーム ILE RPG 様式</th>
<th>ILE RPG D 仕様、列33から39まで</th>
<th>長さ</th>
<th>コメント</th>
</tr>
</thead>
</table>
| char[n] char * | CHAR(n) | nA | n | 文字の配列 (n は、1 から 16773104 まで)。
ヌル終了ストリング・パラメーターの場合は、VALUE および OPTIONS(+STRING) キーワードを使用してポインタとしてプロトタイプ・パラメーターをコーディングします。 |
| char[n] | IND | 1N | 1 | 標識 |
| char[n] | ZONED(n) | nS 0 | n | ソーン 10 進数 |
| _Packed struct (unsigned short i; char[n]) | VARCHAR(n) | nA VARYING | n+2 | 可変長フィールド (i は指定した長さ、n は最大長)。
注: n が 65535 より大きい場合は、構造体の最初のエレメントは符号なし int であり、長さは n+4 です。 |
| _Packed struct (unsigned int i; char[n]) | VARCHAR(n : 4) | nA VARYING(4) | n+4 | 可変長フィールド (i は指定した長さ、n は最大長) |
| wchar_t[n] | UCS2(n) | nC | 2n | UCS-2 文字の配列。
注: RPG UCS-2 タイプでは、CCSID(1200) を指定することで UTF-16 データもサポートされます。 |
| _Packed struct (unsigned short i; wchar_t[n]) | VARUCS2(n) | nC VARYING | 2n+2 | 可変長 UCS-2 フィールド (i は指定した長さ、n は最大長)。
注: n が 65535 より大きい場合は、構造体の最初のエレメントは符号なし int であり、長さは 2n+4 です。
注: RPG UCS-2 タイプでは、CCSID(1200) を指定することで UTF-16 データもサポートされます。 |
<table>
<thead>
<tr>
<th>フリー・フォーム ILE RPG 構文</th>
<th>ILE RPG D 仕様、列 33 から 39 まで</th>
<th>長さ</th>
<th>コメント</th>
</tr>
</thead>
</table>
| _Packed struct (unsigned int i; wchar_t[n]) | VARUCSZ(n : 4) | nC VARYING(4) | 2n+4 | 可変長 UCS-2 フィールド (i は指定した長さ、n は最大長)。
注: RPG UCS-2 タイプでは、CCSID(1200) を指定することで UTF-16 データもサポートされます。 |
| wchar_t[n] | GRAPH(n) | nG | 2n | グラフィック文字の配列。 |
| _Packed struct (unsigned short i; wchar_t[n]) | VARGRAPH(n) | nG VARYING | 2n+2 | 可変長グラフィック・フィールド。
注: n が 65535 より大きい場合は、構造体の初期のエレメントは符号なし int であり、長さは 2n+4 です。 |
| wchar_t[n] | GRAPH(n) | nG | 2n | グラフィック文字の配列。 |
| char[n] | DATE | D | 8, 10 | 日付フィールド |
| char[n] | TIME | T | 8 | 時刻フィールド |
| char[n] | TIMESTAMP | Z | 20-32 | タイム・スタンプ・フィールド |
| short int | INT(5) | 5I 0 | 2 | 整数フィールド |
| short unsigned int | UNS(5) | 5U 0 | 2 | 符号なし整数フィールド |
| int | INT(10) | 10I 0 | 4 | 整数フィールド |
| unsigned int | UNS(10) | 10U 0 | 4 | 符号なし整数フィールド |
| long int | INT(10) | 10I 0 | 4 | 整数フィールド |
| long unsigned int | UNS(10) | 10U 0 | 4 | 符号なし整数フィールド |
| long long int | INT(20) | 20I 0 | 8 | 8 バイトの整数フィールド。 |
| long long unsigned int | UNS(20) | 20U 0 | 8 | 8 バイトの符号なし整数フィールド |
| struct (unsigned int : n)x; | サポートなし。 | サポートなし。 | 4 | 4 バイトの符号なし整数 bitfield |
| float | FLOAT(4) | 4F | 4 | 4 バイトの浮動小数点 |
| double | FLOAT(8) | 8F | 8 | 8 バイトの浮動小数点 |
| long double | FLOAT(8) | 8F | 8 | 8 バイトの浮動小数点 |
| _Decimal32 | サポートなし。 | サポートなし。 | 4 | 4 バイトの 10 進浮動小数点 |
| _Decimal64 | サポートなし。 | サポートなし。 | 8 | 8 バイトの 10 進浮動小数点 |
| _Decimal128 | サポートなし。 | サポートなし。 | 16 | 16 バイトの 10 進浮動小数点 |
表 11-3. ILE RPG に対する ILE C++ データ・タイプの互換性 (続き)

<table>
<thead>
<tr>
<th>プロトタイプにおける ILE C++ 宣言</th>
<th>フリー・フォーム ILE RPG 構文</th>
<th>ILE RPG D 仕様、列 33 から 39 まで</th>
<th>長さ</th>
<th>コメント</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum</td>
<td>INT(p)</td>
<td>P1</td>
<td>1, 2, 4</td>
<td>列挙型 n = 1, 2, 4 の場合、p = 3, 5, 10。列挙値を持つ ILE RPG 名前付き定数を定義します。</td>
</tr>
<tr>
<td>*</td>
<td>POINTER</td>
<td>*</td>
<td>16</td>
<td>ポインター。</td>
</tr>
<tr>
<td>_DecimalT&lt;n,p&gt;</td>
<td>PACKED(n:p)</td>
<td>nP p</td>
<td>n/2+1</td>
<td>バック 10 進数。n は、必ず 30 以下。C++ では、これは、2 進化 10 進数クラスであり、データ・タイプではありません。</td>
</tr>
<tr>
<td>union.element</td>
<td>Keyword POS(1)</td>
<td>OVERLAY(data structure name) の &lt;type&gt;</td>
<td>最長共用体メンバーデータのエレメント</td>
<td></td>
</tr>
<tr>
<td>data_type[n]</td>
<td>keyword DIM(n) の &lt;type&gt;</td>
<td>keyword DIM(n) の &lt;type&gt;</td>
<td>16</td>
<td>C++ がポインターを渡す配列</td>
</tr>
<tr>
<td>構造体またはクラス</td>
<td>データ構造</td>
<td>データ構造</td>
<td>n</td>
<td>構造体。</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RPG データ構造体に ALIGN キーワードがない場合は、構造体で _Packed 修飾子を使用します。</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C 構造体に _Packed 修飾子がない場合は、RPG データ構造体で ALIGN キーワードを使用します。</td>
</tr>
<tr>
<td>関数のポインター</td>
<td>POINTER(*PROC)</td>
<td>マイナリープロシージャ [PROC] が指定された *</td>
<td>16</td>
<td>16 バイトのポインター</td>
</tr>
</tbody>
</table>

注: *virtual* 関数を持つクラスは、他の言語で直接表現することはできません。ネストされた構造体は、明示的にバックする必要があります。C++ では、ネストされたすべての構造体はバックされます。

表 11-4. OPM RPG に対する ILE C++ データ・タイプの互換性を示しています。

<table>
<thead>
<tr>
<th>プロトタイプにおける ILE C++ 宣言</th>
<th>OPM RPG 仕様、DS サブフィールド列の仕様</th>
<th>長さ</th>
<th>コメント</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n]</td>
<td>1 10</td>
<td>n</td>
<td>文字の配列 (n は、1 から 32766 まで)</td>
</tr>
<tr>
<td>char</td>
<td>*INxxxx</td>
<td>1</td>
<td>*IN で始まる変数である標識</td>
</tr>
<tr>
<td>char[n]</td>
<td>1 nd (d=0)</td>
<td>n</td>
<td>ソーン 10 進数、n の値は 30 です</td>
</tr>
<tr>
<td>_Packed struct [unsigned short i; char[n]]</td>
<td>サポートなし。</td>
<td>n+2</td>
<td>可変長フィールド、i は指定された長さで、n は最大長です。</td>
</tr>
<tr>
<td>_Packed struct [unsigned int i; char[n]]</td>
<td>サポートなし。</td>
<td>n+4</td>
<td>可変長フィールド、i は指定された長さで、n は最大長です。</td>
</tr>
<tr>
<td>wchar_t[n]</td>
<td>サポートなし。</td>
<td>2n</td>
<td>UCS-2 文字の配列。</td>
</tr>
<tr>
<td>プロトタイプにおける ILE C++ 宣言</td>
<td>OPM RPG 仕様、DS サブフィールド列の仕様</td>
<td>長さ</td>
<td>コメント</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>_Packed struct</td>
<td>仕様なし。</td>
<td>2n+2</td>
<td>可変長 UCS-2 フィールド。i は指定した長さで、n は最大長です。</td>
</tr>
<tr>
<td>_Packed struct</td>
<td>仕様なし。</td>
<td>2n+4</td>
<td>可変長 UCS-2 フィールド。i は指定した長さで、n は最大長です。</td>
</tr>
<tr>
<td>wchar_t[n]</td>
<td>仕様なし。</td>
<td>2n</td>
<td>グラフィック文字の配列。</td>
</tr>
<tr>
<td>_Packed struct</td>
<td>仕様なし。</td>
<td>2n+2</td>
<td>可変長グラフィック・フィールド。i は指定した長さで、n は最大長です。</td>
</tr>
<tr>
<td>wchar_t[n]</td>
<td>仕様なし。</td>
<td>2n+4</td>
<td>可変長グラフィック・フィールド。i は指定した長さで、n は最大長です。</td>
</tr>
<tr>
<td>char[n]</td>
<td>char</td>
<td>6, 8, 10</td>
<td>日付フィールド</td>
</tr>
<tr>
<td>char[n]</td>
<td>char</td>
<td>8</td>
<td>時刻フィールド</td>
</tr>
<tr>
<td>char[n]</td>
<td>char</td>
<td>26</td>
<td>タイム・スタンプ・フィールド</td>
</tr>
<tr>
<td>short int</td>
<td>8 1 20</td>
<td>2</td>
<td>-9999 から +9999 の範囲の 2 バイト符号付き整数</td>
</tr>
<tr>
<td>int</td>
<td>8 1 40</td>
<td>4</td>
<td>-9999999999 から +9999999999 の範囲の 4 バイト符号付き整数</td>
</tr>
<tr>
<td>long int</td>
<td>8 1 40</td>
<td>4</td>
<td>-9999999999 から +9999999999 の範囲の 4 バイト符号付き整数</td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>8</td>
<td>バイトの整数フィールド。</td>
</tr>
<tr>
<td>struct (unsigned int : n)x;</td>
<td>仕様なし。</td>
<td>1, 2, 4</td>
<td>4 バイトの符号なし整数 bitfield</td>
</tr>
<tr>
<td>float</td>
<td>仕様なし。</td>
<td>4</td>
<td>4 バイトの浮動小数点</td>
</tr>
<tr>
<td>double</td>
<td>仕様なし。</td>
<td>8</td>
<td>8 バイトの浮動小数点</td>
</tr>
<tr>
<td>long double</td>
<td>仕様なし。</td>
<td>8</td>
<td>8 バイトの浮動小数点</td>
</tr>
<tr>
<td>_Decimal32</td>
<td>仕様なし。</td>
<td>4</td>
<td>4 バイトの 10 進浮動小数点</td>
</tr>
<tr>
<td>_Decimal64</td>
<td>仕様なし。</td>
<td>8</td>
<td>8 バイトの 10 進浮動小数点</td>
</tr>
<tr>
<td>_Decimal128</td>
<td>仕様なし。</td>
<td>16</td>
<td>16 バイトの 10 進浮動小数点</td>
</tr>
<tr>
<td>enum</td>
<td>仕様なし。</td>
<td>1, 2, 4</td>
<td>列挙</td>
</tr>
<tr>
<td>*</td>
<td>仕様なし。</td>
<td>16</td>
<td>ポインター</td>
</tr>
</tbody>
</table>
| _DecimalT<n,p>             | P 1 n/2+1d                      | n/2+1 | バック 10 進数。n は、30 以下でなければなりません。C++ では、これは、2 進化 10 進数クラスであり、データ・タイプではありません。
| union.element              | 位置 1 で開始されるデータ構造体サブフィールド | 最長共用体メンバーの長さ | 共用体のエレメント |
| data_type[n]               | E-SPEC 配列                     | 16   | C++ がポインターを渡す配列 |
### 表11-4. OPM RPG に対する ILE C++ データ・タイプの互換性（続き）

<table>
<thead>
<tr>
<th>プロトタイプにおける ILE C++ 宣言</th>
<th>OPM RPG Ⅰ仕様、DS サブフィールド列の仕様</th>
<th>長さ</th>
<th>コメント</th>
</tr>
</thead>
<tbody>
<tr>
<td>構造体またはクラス</td>
<td>データ構造</td>
<td>n</td>
<td>構造体。構造体はバックする必要があります。構造体の内容を変更する場合、受け渡す構造体はポインターとして構造体に渡す必要があります。</td>
</tr>
<tr>
<td>関数のポインター</td>
<td>サポートなし。</td>
<td>16</td>
<td>16 バイトのポインター</td>
</tr>
</tbody>
</table>

注: すべての構造体はパックする必要があります。virtual 関数を持つクラスは、他の言語で直接表現することはできません。ネストされた構造体は、明示的にパックする必要があります。C では、ネストされたすべての構造体はパックされます。

### 表11-5 は、ILE COBOL に対する ILE C++ データ・タイプの互換性を示しています。

### 表11-5. ILE COBOL に対する ILE C++ データ・タイプの互換性

<table>
<thead>
<tr>
<th>プロトタイプにおける ILE C++ 宣言</th>
<th>ILE COBOL の LINKAGE SECTION</th>
<th>長さ</th>
<th>コメント</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n] char *</td>
<td>PIC X(n).</td>
<td>n</td>
<td>文字の配列 (n は、1 から 3,000,000 まで)。</td>
</tr>
<tr>
<td>char</td>
<td>PIC 1 INDIC ..</td>
<td>1</td>
<td>標識</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC S9(n) DISPLAY.</td>
<td>n</td>
<td>ソーン 10 進数</td>
</tr>
<tr>
<td>wchar_t[n]</td>
<td>PIC G(n) DISPLAY.</td>
<td>2n</td>
<td>グラフィック・データ・タイプ</td>
</tr>
<tr>
<td>_Packed struct {short i; char[n]}</td>
<td>05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).</td>
<td>n+2</td>
<td>可変長フィールド。i は指定した長さで、n は最大長です。</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>6</td>
<td>日付フィールド</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>5</td>
<td>日付フィールド</td>
</tr>
<tr>
<td>char</td>
<td>PIC X</td>
<td>1</td>
<td>曜日フィールド</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>8</td>
<td>時刻フィールド</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>26</td>
<td>タイム・スタンプ・フィールド</td>
</tr>
<tr>
<td>short int</td>
<td>PIC S9(4) COMP-4</td>
<td>2</td>
<td>-9999 から +9999 の範囲の 2 バイト符号付き整数</td>
</tr>
<tr>
<td>short int</td>
<td>PIC S9(4) BINARY</td>
<td>2</td>
<td>-9999 から +9999 の範囲の 2 バイト符号付き整数</td>
</tr>
<tr>
<td>int</td>
<td>PIC S9(9) COMP-4</td>
<td>4</td>
<td>-999999999 から +999999999 の範囲の 4 バイト符号付き整数</td>
</tr>
<tr>
<td>int</td>
<td>PIC S9(9) BINARY</td>
<td>4</td>
<td>-999999999 から +999999999 の範囲の 4 バイト符号付き整数</td>
</tr>
<tr>
<td>int</td>
<td>USAGE IS INDEX</td>
<td>4</td>
<td>4 バイトの整数</td>
</tr>
<tr>
<td>long int</td>
<td>PIC S9(9) COMP-4</td>
<td>4</td>
<td>-999999999 から +999999999 の範囲の 4 バイト符号付き整数</td>
</tr>
<tr>
<td>long int</td>
<td>PIC S9(9) BINARY</td>
<td>4</td>
<td>-999999999 から +999999999 の範囲の 4 バイト符号付き整数</td>
</tr>
<tr>
<td>struct {unsigned int : n]x;</td>
<td>PIC S9(9) COMP-4 PIC X(4)</td>
<td>1, 2, 4</td>
<td>ビット・フィールドは、16 進数リテラルを使用して操作できます。</td>
</tr>
</tbody>
</table>
表11-5. ILE COBOL に対する ILE C++ データ・タイプの互換性（続き）

<table>
<thead>
<tr>
<th>プロトタイプにおける ILE C++ 宣言</th>
<th>ILE COBOL の LINKAGE SECTION</th>
<th>長さ</th>
<th>コメント</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>USAGE IS COMP-1</td>
<td>4</td>
<td>4 バイトの浮動小数点</td>
</tr>
<tr>
<td>double</td>
<td>USAGE IS COMP-2</td>
<td>8</td>
<td>8 バイトの浮動小数点</td>
</tr>
<tr>
<td>long double</td>
<td>サポートなし。</td>
<td>8</td>
<td>8 バイトの浮動小数点</td>
</tr>
<tr>
<td>enum</td>
<td>サポートなし。</td>
<td>1, 2, 4</td>
<td>列挙</td>
</tr>
<tr>
<td>*</td>
<td>USAGE IS POINTER</td>
<td>16</td>
<td>ポインター</td>
</tr>
<tr>
<td>_DecimalT&lt;n,p&gt;</td>
<td>PIC S9(n-p) 9(p) COMP-3.</td>
<td>n/2+1</td>
<td>バック 10 進数。 C++ では、これは、2 進化10 進数クラスであり、データ・タイプではありません。</td>
</tr>
<tr>
<td>_DecimalT&lt;n,p&gt;</td>
<td>PIC S9(n-p) 9(p) PACKED-DECIMAL.</td>
<td>n/2+1</td>
<td>バック 10 進数。 C++ では、これは、2 進化10 進数クラスであり、データ・タイプではありません。</td>
</tr>
<tr>
<td>union.element</td>
<td>REDEFINES</td>
<td>最長共用体メンバーヨンの長さ</td>
<td>共用体のエレメント</td>
</tr>
<tr>
<td>data_type[n]</td>
<td>OCCURS</td>
<td>データ・タイプの長さの n 倍</td>
<td>C++ がポインターを渡す配列</td>
</tr>
<tr>
<td>構造体またはクラス</td>
<td>OCCURS ...DEPENDING ON</td>
<td>変数</td>
<td>構造体、構造体はバックする必要がありません。1 構造体の内容を変更する場合、受け渡す構造体はポインターとして構造体に渡す必要があります。</td>
</tr>
<tr>
<td>構造体またはクラス</td>
<td>01 record</td>
<td>n</td>
<td>構造体、構造体はバックする必要がありません。1 構造体の内容を変更する場合、受け渡す構造体はポインターとして構造体に渡す必要があります。</td>
</tr>
<tr>
<td>関数のポインター</td>
<td>PROCEDURE-POINTER</td>
<td>16</td>
<td>プロシージャーを指す 16 バイトのポインター</td>
</tr>
<tr>
<td>サポートなし。</td>
<td>PIC S9(18) COMP-4</td>
<td>8</td>
<td>8 バイトの整数</td>
</tr>
<tr>
<td>サポートなし。</td>
<td>PIC S9(18) BINARY</td>
<td>8</td>
<td>8 バイトの整数</td>
</tr>
</tbody>
</table>

注: 1 すべての構造体はバックする必要があります。virtual 関数を持つクラスは、他の言語で直接表現することはできません。ネストされた構造体は、明示的にバックする必要があります。C では、ネストされたすべての構造体はバックされます。

表11-6 は、OPM COBOL に対する ILE C++ データ・タイプの互換性を示しています。

表11-6. OPM COBOL に対する ILE C++ データ・タイプの互換性

<table>
<thead>
<tr>
<th>プロトタイプにおける ILE C++ 宣言</th>
<th>OPM COBOL の LINKAGE SECTION</th>
<th>長さ</th>
<th>コメント</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>n</td>
<td>文字の配列 (n は、1 から 3,000,000 まで)。</td>
</tr>
<tr>
<td>char</td>
<td>PIC 1 INDIC ..</td>
<td>1</td>
<td>標識</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC S9(n) USAGE IS DISPLAY</td>
<td>n</td>
<td>ズーン 10 進数。n の限度は 18 です。</td>
</tr>
</tbody>
</table>
表 11-6. OPM COBOL に対する ILE C++ データ・タイプの互換性 (続き)

<table>
<thead>
<tr>
<th>プロトタイプにおける ILE C++ 宣言</th>
<th>OPM COBOL の LINKAGE SECTION</th>
<th>長さ</th>
<th>コメント</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Packed struct (short i; char[n])</td>
<td>05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).</td>
<td>n+2</td>
<td>可変長フィールド。i は指定した長さで、n は最大長です</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>6, 8, 10</td>
<td>日付フィールド</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>8</td>
<td>時刻フィールド</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>26</td>
<td>タイム・スタンプ・フィールド</td>
</tr>
<tr>
<td>short int</td>
<td>PIC S9(4) COMP-4</td>
<td>2</td>
<td>-9999 から +9999 の範囲の 2 バイト符号付き整数</td>
</tr>
<tr>
<td>int</td>
<td>PIC S9(9) COMP-4</td>
<td>4</td>
<td>-999999999 から +999999999 の範囲の 4 バイト符号付き整数</td>
</tr>
<tr>
<td>long int</td>
<td>PIC S9(9) COMP-4</td>
<td>4</td>
<td>-999999999 から +999999999 の範囲の 4 バイト符号付き整数</td>
</tr>
<tr>
<td>struct {unsigned int : n}x;</td>
<td>PIC 9(9) COMP-4 PIC X(4)</td>
<td>1, 2, 4</td>
<td>ビット・フィールドは、16 進数リテラルを使用して操作できます</td>
</tr>
<tr>
<td>float</td>
<td>サポートなし。</td>
<td>4</td>
<td>4 バイトの浮動小数点</td>
</tr>
<tr>
<td>double</td>
<td>サポートなし。</td>
<td>8</td>
<td>8 バイトの浮動小数点</td>
</tr>
<tr>
<td>long double</td>
<td>サポートなし。</td>
<td>8</td>
<td>8 バイトの浮動小数点</td>
</tr>
<tr>
<td>enum</td>
<td>サポートなし。</td>
<td>1, 2, 4</td>
<td>列挙</td>
</tr>
<tr>
<td>*</td>
<td>USAGE IS POINTER</td>
<td>16</td>
<td>ポインター</td>
</tr>
<tr>
<td>_DecimalT&lt;n,p&gt;</td>
<td>PIC S9(n-p)V9(p) COMP-3.</td>
<td>n/2+1</td>
<td>バック 10 進数。n および p の限度は 18 です。C++ では、これは、2 進化 10 進数クラスであり、データ・タイプではありません</td>
</tr>
<tr>
<td>union.element</td>
<td>REDEFINES</td>
<td>最長共用体メンバーの長さ</td>
<td>共用体のエレメント</td>
</tr>
<tr>
<td>data_type[n]</td>
<td>OCCURS</td>
<td>データ・タイプの長さの n 倍</td>
<td>C++ がポインターを渡す配列</td>
</tr>
<tr>
<td>構造体またはクラス</td>
<td>OCCURS ...DEPENDING ON</td>
<td>変数</td>
<td>構造体。構造体はバックする必要があります。1 構造体の内容を変更する場合、受け渡す構造体はポインターとして構造体に渡す必要があります。</td>
</tr>
<tr>
<td>構造体またはクラス</td>
<td>01 record</td>
<td>n</td>
<td>構造体。構造体はバックする必要があります。1 構造体の内容を変更する場合、受け渡す構造体はポインターとして構造体に渡す必要があります。</td>
</tr>
<tr>
<td>頃数のポインター</td>
<td>サポートなし。</td>
<td>16</td>
<td>16 バイトのポインター</td>
</tr>
<tr>
<td>サポートなし。</td>
<td>PIC S9(18) COMP-4</td>
<td>8</td>
<td>8 バイトの整数</td>
</tr>
</tbody>
</table>

注: 1 すべての構造体はバックする必要があります。virtual 頃数を持つクラスは、他の言語で直接表現することはできません。ネストされた構造体は、明示的にバックする必要があります。

C では、ネストされたすべての構造体はバックされます。
関連情報

ILE C プログラミングに関連するトピックの詳細は、以下の IBM 資料を参照してください。

・ IBM i Information Center では、適用業務開発管理機能/400 の機能用に定義されたプロジェクトの作成と管理、およびこのプログラムを使用したアプリケーションの開発について説明しています。プログラマー情報、ADTS をご確認ください。

・ 『適用業務開発ツールセット AS/400 用 規格情報 』には、適用業務開発ツールセット/400 の規格情報についての情報が記載されています。この規格情報は、SEU の規格情報の参照を含むもので、これに基づいてアプリケーションの開発および他の CICS インターフェースの既存のアプリケーションのマイグレーションに関する一般情報が記載されています。

・ 『CICS for iSeries® Application Programming Guide (SC41-5454-02)』は、CICS/400 のアプリケーション・プログラミングに関する情報を提供しています。この資料には、CICS® アプリケーション・プログラミング・インターフェースとシステム・プログラミング・インターフェース・コマンドに関する手引きおよび参照情報、さらに、新しいアプリケーションの開発や他の CICS インターフェースからの既存のアプリケーションのマイグレーションに関する一般情報が記載されています。

・ 『ILE C/C++ for AS/400 MI Library Reference (SC09-2418-00)』には、システム・レベルのプログラミング機能を提供する ILE C コンパイラで使用可能なマシン・インターフェース命令に関する情報を記載しています。

・ 『CL プログラミング (SD88-5038-06)』には、オブジェクトとライブラリー、CL プログラミング、プログラム内の制御フローと連絡、制御言語プログラムでのオブジェクトの処理、および制御言語プログラムの作成についての概要説明を含むプログラムに関するトピックを広範囲にわたって説明しています。その他のトピックとしては、事前定義メッセージと即時メッセージおよびメッセージの処理、ユーザー定義のコマンドとメニューの定義と作成、デバッグ・モード、ブレークポイント、追跡、および表示機能を含むアプリケーションのテストなどが入っています。

・ 『Communications Management (SC41-5406-02)』には、通信環境における実行管理機能、通信状況、通信障害の追跡および診断、エラーの処理と回復手順、パフォーマンス、特定の回線速度、およびサブシステム記憶域情報に関する説明が記載されています。

・ 『Experience RPG IV Multimedia Tutorial』(SK2T-2700) は、RPG III と RPG IV の違いおよび新しい ILE 環境での作業方法を説明する、対話式の専門プログラムです。付属のワークブックには追加の練習
問題が載せられていて、学習終了後には参照資料として役立ちます。ILE RPG のコーディング例が、
自習用プログラムとともに配布され、それはシステム上で直接実行できます。

-『GDDM Programming Guide (SC41-0536-00)』では、图形データ表示管理プログラム (GDDM) を使っ
て图形データ・アプリケーション・プログラムを作成する方法を説明します。多くのプログラム例およ
びこのプロダクトをデータ処理システムに適合させる方法を理解するのに役立つ情報が含まれていま
す。

-『GDDM Reference (SC41-3718-00)』では、图形データ表示管理プログラム (GDDM) を使って图形デー
タ・アプリケーション・プログラムを作成する方法を説明します。この資料には、GDDM で使用可能な
すべてのグラフィックス機能の詳細説明が入っています。また、GDDM への高水準言語インターフェー
スについての説明も入っています。

-『ICF Programming (SC41-5442-00)』では、システム通信およびシステム間通信機能 (ICF) を使用する
アプリケーション・プログラムを作成するのに必要な情報を提供します。また、データ記述仕様書
(DDS) キーワード、システム提供の様式、戻りコード、ファイル転送サポートの説明、およびプログラ
ム例も入っています。

-『IDDU Use (SC41-5704-00)』では、対話式データ定義ユーティリティー (IDDU) を使用してデータ・デ
ィクショナリー、ファイル、およびレコードをシステムに対して記述する方法を説明します。次の項目
が含まれています。

- コンピューター・ファイルおよびデータ定義の概念の紹介
- 照会および文書で使用するデータを記述するための IDDU の使用法の紹介
- データ・ディクショナリー、ファイル、レコード様式、およびフィールドの作成、保守、および使用
に関する代表的な作業
- 他のシステムで作成されたファイルを処理するための IDDU の使用法、エラー回復と問題の防止策に
関する詳細説明

-『IBM Rational® Development Studio for i: ILE C/C++ コンパイラ参照 (SC88-4025-02)』には、ILE
C/C++ コンパイラに関する参照情報が記載されています。この情報には、コンパイラ・オプショ
ン、ILE C/C++ マクロ、プリプロセッサー・ディレクティブ、およびプログラムが含まれています。

-『IBM Rational Development Studio for i: ILE C/C++ 解説書 (SC88-4026-02)』では、言語のエレメン
ト、ステートメント、およびプリプロセッサー・ディレクティブなど、ILE C/C++ コンパイラに関する
参照情報を提供しています。例が示されており、プログラミングの考慮事項も説明されています。

-『Standard C/C++ Library Reference (SC09-4949-01)』では、標準的な C/C++ 言語、ステートメント、お
よびプリプロセッサー・ディレクティブに関する参照情報を提供しています。

-『ILE C/C++ ランタイムライブラリー関数 (SC88-4701-01)』では、標準 C ライブラリー関数および
ILE C ライブラリー拡張機能を含む、ILE C ライブラリー関数に関する参照情報を提供しています。例
が示されており、プログラミングの考慮事項も説明されています。

-『IBM Rational Development Studio for i: ILE COBOL プログラマーの手引き (SD88-5045-07)』では、シ
ステムにおける ILE COBOL プログラムの作成、コンパイル、バインド、実行、デバッグ、および保守
の方法について説明しています。ここには、他の ILE COBOL および非 ILE COBOL プログラムの呼び
出し方法、他のプログラムとのデータの共用方法、ポインタの使用法、および例外の処理方法について
のプログラミング情報が含まれています。また、外部接続装置、データベース・ファイル、表示装置
ファイル、および ICF ファイルに対する入出力操作の実行方法が説明されています。

-『IBM Rational Development Studio for i: ILE COBOL 解説書 (SD88-5044-07)』では、ILE COBOL プロ
グラミング言語について説明しています。ILE COBOL プログラム言語の構造および ILE COBOL ソー
ス・プログラムの構造についての情報を提供します。さらにこの資料には、見出し部の段落、環境部の
文節、データ部の文節、手続き部のステートメント、およびコンパイラ指示ステートメントのすべて
についても説明されています。
IBM Rational Development Studio for i: ILE RPG プログラマの手引き (SD88-5042-07) で、システムで統合化言語環境 (ILE) に RPG IV 言語をインプリメントしたものである、RPG IV プログラム言語に関する情報も提供しています。この資料には、プログラムの作成と実行、およびプロシージャ呼び出すと音楽の演出の調節に関する情報が含まれています。また、デバッグと例外処理および、RPG プログラムそのシステム・ファイルとシステムの使用方法を説明しています。また、RPG IV への移植に関する情報およびコンバイラー・リストの例も含まれています。この資料は、データ処理の概念および RPG 言語の基礎を理解している方を対象としています。

IBM Rational Development Studio for i: ILE RPG 言語の解説書 (SD88-5043-07) では、ILE RPG プログラミング言語に関する情報も提供しています。この解説書は位置ごとおよびキーワードごとに、すべての RPG IV 仕様書に有効な項目を説明し、またすべての演算コードおよび組み込み関数を詳細に説明しています。またこの解説書には、RPG の論理サクル、配列とテーブル、編集機能、および知識の説明があります。

ローカル装置の構成 (SD88-5003-00) では、システム上のローカル装置の構成に関する情報も提供しています。これには、次のものを構成する方法が含まれます。
- ローカル・ワークステーション制御装置（平衡型制御装置を含む）
- テープ制御装置
- ローカル接続装置（平衡型装置を含む）

マシン・インターフェース機能解説書 (Machine Interface Functional Reference) (SC41-5810-00) では、マシン・インターフェース命令セットについて説明しています。また、各命令により実行できる機能について説明し、各命令をコード化するのに必要な情報も提供しています。

印刷装置プログラミング (SD88-5073-03) では、印刷について理解し、印刷を制御するための情報を提供しています。システムの印刷要素およびその概念の特定情報、印刷操作に対するプリンター・ファイルと印刷スプーリング・サポート、およびプリンターの接続性について説明します。パーソナル・コンピューターの使用についての考慮事項、ビジネス・グラフィックス・ユーティリティ（BGU）、拡張機能の印刷（AFP）などの他の印刷機能、スプール出力ファイルある出力待ち行列から別の出力待ち行列へ移動するといったシステムの印刷要素の処理例が入っています。さらに、印刷作業負荷の管理のために使用する制御言語（CL）コマンドのリストも含まれています。また、システムで使用できるフォントも示されています。フォント配置テーブルによって、接続されたプリンターがアプリケーション指定のフォントをサポートしていない場合の、置き換えフォントの相互参照が提供されます。

REXX/400 Programmer's Guide (SC41-5728-00) では、システム上での REXX を使用したプログラミングについて広範囲にわたって説明しています。この資料の主な目的は、REXX の初心者に役立つプログラミング情報と例を提供することと、他のコンピューター環境で REXX の使用経験をもつ方に REXX インプリメンテーションに関する情報を提供することです。

IBM Rational Development Studio for i: ILE RPG プログラマの手引き (SD88-5042-07) では、システムにおける RPG プログラムの設計、コーディング、コンパイラーおよびテストに必要な情報を提供しています。この資料では、データ構造、データ形式、ファイル処理、複数のファイル処理、報告書簡易作成機能、RPG コマンド・ステートメント、テスト機能およびデバッグ機能、アプリケーション設計技法、問題分析、およびコンパイラのサービスについての情報を提供しています。コンパイラ用 RPG、システム/38 環境 RPG III コンパイラ、およびシステム/36 と互換性のある RPG II コンパイラの違いについても説明しています。

システムセキュリティ 機密保護解説書 (SD88-5027-11) では、システム・セキュリティ・サポートを使用して、システムおよびデータが適切な権限のないユーザーによって使用されるのを防ぐ方法、データを
故意または偶発的な損傷または消滅から保護する方法、セキュリティー情報を最新の状態に保つ方法、およびシステムにセキュリティーをセットアップする方法について説明します。

- 『ローカル装置の構成 (SD88-5003-00)』では、初期インストールと、IBM 提供のライセンス・プログラム、プログラム一時修正 (PTF)、および 2 次言語のインストール手順をステップごとに示します。またこの解説書は、既にあるリリースがインストールされたシステムに新しいリリースをインストールするユーザーにも役立ちます。

- IBM i Information Center には、IBM i アプリケーション・プログラミング・インターフェース (API)を使用したい経験豊富なアプリケーション・プログラマーおよびシステム・プログラマー向けの情報、およびプログラマーが API を使用する上で役立つ例が記載されています。
特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒103-8510
東京都中央区日本橋箱崎町19番21号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス係

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報は、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信じる方法で、使用もしくは配布をすることができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.
本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストを行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確認できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書はプランニング目的としてのみ記述されています。記述内容は製品が使用可能になる前に変更になる場合があります。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名前はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インタフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。これらのサンプル・プログラムは特定物として現存するままの状態で提供されるものであり、いかなる保証も提供されません。IBM は、お客様の当該サンプル・プログラムの使用から生ずるいかなる損害に対しても一切の責任を負いません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年)。このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。

© Copyright IBM Corp. _年を入れる_

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。
プログラミング・インターフェース情報
この「ILE C/C++ プログラマーの手引き」の資料には、プログラムを作成するユーザーが ILE C/C++ コンパイラのサービスを使用するためのプログラミング・インターフェースが記述されています。

商標

Adobe、Adobe ロゴ、PostScript、PostScript ロゴは、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

Intel、Intel ロゴ、Intel Inside、Intel Inside ロゴ、Intel Centrino、Intel Centrino ロゴ、Celeron、Intel Xeon、Intel SpeedStep、Itanium、および Pentium は、Intel Corporation または子会社の米国およびその他の国における商標または登録商標です。

Linux は、Linus Torvalds の米国およびその他の国における登録商標です。

Microsoft、Windows、Windows NT および Windows ロゴは、Microsoft Corporation の米国およびその他の国における商標です。

UNIX は The Open Group の米国およびその他の国における登録商標です。

Java™ およびすべての Java 関連の商標およびロゴは Oracle やその関連会社の米国およびその他の国における商標または登録商標です。

他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。

使用条件
これらの資料は、以下の条件に同意していただける場合に限りご使用いただけます。

個人使用：これらの資料は、すべての著作権表示その他の所有権表示をしていたいただくことを条件に、非商業的な個人による使用目的に限り複製することができます。ただし、IBM の明示的な承諾をえずに、これらの資料またはその一部について、二次的著作物を作成したり、配布（頒布、送信を含む）または表示（上映を含む）することはできません。

商業的使用：これらの資料は、すべての著作権表示その他の所有権表示をしていたいただくことを条件に、お客様の企業内に限り、複製、配布、および表示することができます。ただし、IBM の明示的な承諾をえずにこれらの資料の二次的著作物を作成したり、お客様の企業外で資料またはその一部を複製、配布、または表示することはできません。

ここに明示的に許可されているもの以外に、資料や資料内に含まれる情報、データ、ソフトウェア、またはその他の知的所有権に対するいかなる許可、ライセンス、または権利を明示的にも黙示的にも付与するものではありません。

資料の使用が IBM の利益を損なうと判断された場合や、上記の条件が適切に守られていないと判断された場合、IBM はいつでも自らの判断により、ここで与えた許可を撤回できるものとさせていただきます。
お客様がこの情報をダウンロード、輸出、または再輸出する際には、米国のすべての輸出入関連法規を含む、すべての関連法規を遵守るものとします。

IBM は、これらの資料の内容についていかなる保証もしません。これらの資料は、特定物として現存するままの状態で提供され、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任なしで提供されます。
索引

日本語、数字、英字、特殊文字の順に配列されています。なお、拗音と半拗音は清音と同等に扱われています。

【ア行】
アドレス・オーバーフロー 4-70
位置指定モード 4-65
インクルード・ソース・ビュー 5-13
インポート
定義 4-9
未解決要求 4-27
従来参照 4-28
処理 4-31
プログラム作成順序 4-32
要求、解決 4-11
インライナ化 4-61
インライナ化 (C++) 4-62
ウォッチ
活动状態の表示 5-29
ウォッチ条件
および条件付きブレークポイント 5-25
除去 5-28
設定と除去 5-26, 5-29
設定例 5-28
受け渡し
関数へのバック 10 進変数を指すポインタ 8-131
異なる方法 8-76
スタイル 8-76
操作記述子の使用によるパラメターの 8-86
パック 10 進数値変数に 8-130
パック 10 進数データの関数への 8-131
パック 10 進引数 8-132
パラメーター 4-43, 8-85
パラメーター (C++) 8-81
引数 8-76
引数としてのポインタ 8-43
C++ から別の HLL へのパラメターの 8-80
エクスポート
決定 4-22
定義 4-9
の処理 4-22
モジュールの表示コマンドの使用 4-23
エクスポート (続き)
リストの更新 4-25, 4-34
エスケープ・コード 9-5
エラー
言語固有 8-2
コンパイラ時 8-136, 8-137, 8-138
実行時 8-137
パック 10 進数データ・タイプ 8-136
メッセージング・サポート 4-15
エラー・マクロ
EIOERROR 8-6
EIORECERR 8-6
オーバーフローの振る舞い 8-130
オープン
データベース・ファイルをバイナリーストリーム・ファイルとして 7-29
表示装置ファイルをレコード・ファイルとして 7-57
DDM ファイルをバイナリーストリーム・ファイルとして 7-29
オープン・システム (QOpenSys) 6-26
オープン・データ・バス 7-26
オープン・フィードバック 6-20
応答標識 7-41
オブジェクトの圧縮 (CPROBJ) コマンド 4-17, 4-73
オブジェクトの圧縮解除 (DCPOBJ) コマンド 4-73
オブジェクトの割り振り (ALCOBJ) コマンド 7-26
オプション
速度対サイズ 4-74
パフォーマンス測定を使用可能にする 4-58
パフォーマンス用 4-75
INLINE 4-61
OPTIMIZE 4-74
rtncodeexec 4-59
オプション標識 7-41
外部記述ファイル (続き)
フィールド・レベル記述 7-23
物理データベース・ファイル 7-17
レコード様式名 7-3
論理データベース・ファイル 7-17
pragma mapinc ディレクティブ 7-1
外部記述物理データベース・ファイル 7-17
外部プログラム、宣言 8-82
活動化
グループ化 4-40
説明 4-14, 4-40
プロセス 4-40
活動化、説明 3-3
活動化グループ
作成 4-39
識別 4-10
指定 4-49
使用する理由 4-14
処理 4-40
説明 4-49
デフォルト・ヒープ 4-54
動的記憶域 4-54
名前を付いた削除 4-53
パフォーマンスへの影響 4-73
プログラムの異常終了 4-48
命名 4-40
*CALLER 4-52
*NEW 4-50
活動化グループ、説明 3-3
活動化グループ、名前を付いた
および呼び出しスタック 4-52
削除 4-53
作成 4-49
実行時の動作 4-50
パフォーマンスへの影響 4-73
ISO に準拠しない振る舞い 4-51
環境変数
ロケール 9-15
関数
ステップイン 5-34
ストリーム 4-66
バイナリーストリーム、データベース・ファイルおよび DDM ファイル用 7-30
パック 10 進データ型のライブラリ関数 8-133
発信されたシグナル 8-25
評価 5-36
戻り値の検査 8-5
レコード出力 4-63

© Copyright IBM Corp. 1993, 2013
関数 (続き)
ロケール依存ランタイム 9-16
fclose() 7-57
fopen() 6-2, 6-12, 6-32, 6-33, 7-57, 7-66
fopen() フォーマット 6-5
fread() 7-57
freopen() 6-4
frew() 7-57
fwrite() 7-57, 7-66
ILE リンケージの指定 8-82
lecl パラメーター 6-2
open() メンバー 6-13
raise 8-25
setbuf() 6-2
setvbuf() 6-2
signal 8-25
_GetExcData 8-25
_Racquirer() 7-58, 7-59
_Rclose() 7-59
_Rdevat() 7-59
_Rformat() 7-58, 7-59
_Rindara 7-42
_Rindarra 7-59
_Riofbk() 7-51, 7-59
_Ropen() 7-57, 7-58
_Rpgmdev() 7-49, 7-58, 7-59
_Rreaddd() 7-59
_Rreadindnc() 7-59
_Rreadindn() 7-59
_Rreadindv() 7-58, 7-59
_Rreadnc() 7-58
_Rreleas() 7-58, 7-59
_Rrupdata() 7-59
_Rrup() 7-59
_Rwritten() 7-59
_Rwritten() 7-59
_Rwrread() 7-59
閲覧結果を戻す 8-79
閲覧結果を戻す共通メカニズム 8-79
関数の ILE リンケージ 8-82
関数の戻り値、検査 8-5
関数ポインタ 8-40
関数呼び出し 5-2
記憶域解放 (CEEFRST) バインド可能 API 4-54
記憶域再割り振り (CEECZST) バインド可能 API 4-54
規則、呼び出し 8-45, 8-75
共通インタフェース 4-16
共有ファイル、使用削除 4-66
クラス
評価 5-36
クラス (続き)
C 構造体への C++ クラスのマップ 8-54
C プログラムからの C++ クラスへのアクセス 8-55
extended_type_info 8-185
ILE C++ 8-54
ILE C++ での使用のための作成 8-54
type_info 8-184
グラフィック文字 9-1
gローバル変数
_EXCP_MSGID 8-8
言語環境 9-9, 9-13
言語環境のレベルメント 9-9
言語間の呼び出し 8-60
原始ステートメント入力ユーティリティ
(SEU) 4-24
原始ステートメント入力ユーティリティ
の開始 (STRSEU) コマンド 4-3
コード化文字セット 9-9
コード化文字セット ID (CCSID)
グラフィック文字 9-1
cコード・ページ 9-1
cコード・ポイント 9-1
c混合バイリット CCSID 9-14
c指定 9-2
cソース・ファイル変換 9-2
cターゲット CCSID (TGTCSSID) バラメーター 9-7, 9-8
c定義 9-1
c変更 9-3
c文字データ処理体系 (CDRA) 9-1
c例外 4-4
ワイド文字リテラリク 9-4
1 バイト CCSID 9-14
C 文字の代わりの 3 文字表記 4-4
CCSID 037 9-2, 9-3
CCSID 037 以外 9-1
CCSID 1026 9-1
CCSID 1100 9-8
CCSID 1301 9-8
CCSID 13488 9-4
CCSID 273 9-2
CCSID 290 9-1, 9-8
CCSID 5026 9-8
CCSID 65535 9-2, 9-3
CCSID 905 9-1
CCSID 930 9-8
ILE C/C++ コンパイラ認識 9-1
コード・ページ 9-1
更新
サービス・プログラム・エクスポート・リスト 4-34
構造体
評価 5-36, 5-45
構造体、表示 5-40
国際ロケールのサポート
ILE C サポート 9-9
異なる受け渡し方法 8-76
コマンド行、デバッグ 5-3
コマンド作成 (CRTCMD) コマンド 4-46
コマンド機能変更 (CHGMDDFT) コマンド 4-6, 4-39
コマンド変更プログラム (CPP) 8-88
コミットメント制御 7-36
コミットメント制御の開始
(STRMTMTCTL) コマンド 7-36
コミットメント制御の終了
(ENDMTMTCTL) コマンド 7-36
混合バイリット環境 9-1
コンパイル時エラー 8-136

【役行者】
サービス・プログラム
エクスポート 4-22
エクスポート・リストの更新 4-25, 4-34
関連コマンド 4-18
共通インタフェース 4-16
更新 4-17
作成 4-16
作成の理由 4-9, 4-15
サンプル・プログラム SEARCH 4-18
自動でデバッグ・モードに入る 5-34
条件ハンドラ 8-18
対プログラム 4-15
バインダー言語を使用して作成 4-27
プレースホルダーの使用 4-33
変更 4-17
サービス・プログラムの更新
(UDPSRVPGM) コマンド 4-17
サービス・プログラムの作成
(CRTSRVPGM) コマンド 4-16, 4-17
サービス・プログラム変更
(CHGSRVPGM) コマンド 4-17
最適化プロセス 4-74
最適化レベル
変更 4-74, 5-54
*FULL 5-54
*NONE 5-54
作成
一時モジュール 4-5
インクルード・ソース・ビュー 5-13
サービス・プログラム 4-9
新規コマンド 4-46
デバッグのためのリスト・ビュー 5-12
バインディング・ディレクトリ 4-9
ビュー 5-13
プログラム 4-1, 4-9
リスト・ビュー 5-13
データ記述仕様 (DDS) (続き)

【夕行】

タイプ・キャスト 5-2
ダウンキャスト 8-181
直接モニター・ハンドラー
使用 8-8
有効範囲 8-9
データ記述仕様 (DDS) 7-24
テラスベース (C++) 8-171
演算子 delete 4-56, 8-177
演算子 delete、多重定義 4-56
演算子 new 4-56, 8-177
演算子 new または演算子 delete の多重定義 4-56
演算子 new、多重定義 4-56
環境の決定 8-171
関数の多重定義 8-180
サポートされる環境 8-171
置換関数 (C++) 4-56
置換関数を多重定義する際の C++ 制限 4-56
定義 8-171
バインド可能な API 8-172
ポインタ型変換 8-171
16 バイトのポインタ 8-171
C++ 置換関数 4-56
C/C++ のポインタのサポート 8-171
テラスベースを使用するためのバインド可能 API 8-172
テンプレート
インクルード・ファイル 8-170
インスタンス化 8-160, 8-165
インスタンス化、自動 8-162, 8-165
関数定義 8-168
実装ファイル 8-169
使用 8-160
明示的特殊化 8-163
テナプレート、表示 5-47
統合ファイル・システム 6-30
統合ファイル・システム (IFS) 6-24
オープン・システム (QOpenSys) 6-26
コンパイル 6-36
ストリームとしてのデータの保管 6-33
ストリーム・ファイル 6-30
ストリーム・ファイル、編集 6-35
ストリーム・ファイルとデータベース
ス・ファイル 6-31
テキスト・ストリーム 6-32
バイナリー・ストリーム 6-32
光ディスクサポート (QOPT) 6-28
ファイル・サーバー
(QFilesrv:400) 6-29
[ナ行]
名前、名前 4-26
名前、名前 4-26
出力に関する考慮事項
バイナリー・ストリーム ICF ファイル
7-60
バイナリー・ストリーム保持ファイル
7-77
バイナリー・ストリーム・サブファイル
7-74
バイナリー・ストリーム・データベース
7-57
バイナリー・ストリーム・データベース
7-29
バイナリー・ストリーム・データベース
ファイル 7-70
レコード・サブファイル 7-55
レコード・ディスケット・ファイル
7-74
DDM ファイル 7-29
入出力フィードバック 6-20
ヌル可能フィールド 7-27
ヌル終了文字配列、表示 5-41
ヘーザー、ヘッダー 4-23
ネイティブ言語 9-9
ネストされた例外 8-32

[ハ行]
パーコレーショ 例 4-60
バイナリー・ストリーム 6-33
バイナリー・ストリーム、記述 6-32
バイナリー・ストリーム・ファイル
オブジェクト、ファイル 6-11
オブジェクト、ファイル (C++) 6-13
オブジェクト、レコード単位 6-17
オブジェクト、レコード単位 (C++) 6-18
書き込み、文字単位 6-13
書き込み、レコード単位 6-18
更新、文字単位 6-15
バイナリー・ストリーム ICF ファイル
出力に関する考慮事項 7-60
プログラム装置 7-60
バイナリー・ストリーム・ファイル (続き)
バイナリー・ストリーム表示装置ファイル
プログラム装置 7-57
バイナリー・ストリーム保持ファイル
入出力に関する考慮事項 7-77
バイナリー・ストリーム・サブファイル
入出力に関する考慮事項 7-57
バイナリー・ストリーム・データベース・ファイル
入出力に関する考慮事項 7-29
バイナリー・ストリーム・データベース・ファイル
入出力に関する考慮事項 7-70
読み取り、文字単位 6-14
読み取り、レコード単位 6-19
配列、評価 5-36
バイナダー、使用 4-9
バイナダー用語
サービス・プログラムを作成する
4-27
作成 4-24, 4-30
使用する理由 4-15
ソース・ファイル 4-22
ソース・ファイルの作成 4-23, 4-24
EXPORT キーワード 4-25
EXPORT シンボル 4-79
バイナダー・ソース検索 (RTVBNDSRC)
コマンド 4-24
バイナダー・リスト、使用 4-11
バイナダー・ディレクトリの作成 4-9
バイナダー・ディレクトリの理由 4-9
バイナダー・プログラムの作成
(CRTBNDDIR) コマンド 4-9
パイナダー・プログラムの作成
(CRTBNDDIR) コマンド 5-13
パイナダー可能な API
記憶域の解放または再割り振り 4-54
デフォルト・ヒープの管理 4-54
パイナダー・ディレクトリの作成
(CRTBNDDIR) コマンド 4-9
パイナダー・プログラム言語
シンボルのエクスポート (EXPORT)
コマンド 4-34
バック 10 進数データ
使用 7-21
変換関数 7-22
バック 10 進数データ型
互換性のある 2 進化 10 進数の作成 8-62
バック 10 進数データ・タイプ
#pragma nostigtrunc ディレクトリ 8-138
バック 10 進数データ・タイプ、C と
C++ サポートの比較 8-127
バック 10 進数データ・タイプ表現
8-127, 8-139
バック 10 進数データ・タイプ・エラーの
理解 8-137
パフォーマンス
改善 4-56
関数呼び出しのインライン化 4-61
入出力に関する考慮事項 4-63
関数呼び出し 4-61
考上
埋め込みに使用されるスペースの削
減 4-71
仮想関数の回避 4-74
スペース所要量の削減 4-70
データ型の選択 4-70
動的メモリ・配列呼び出しの削
減 4-70
プログラム開始処理時間の短縮
4-74
コンパイル時
改善 4-74
最少のコードの変更での改善 4-58
実行時のパフォーマンス 4-56
測定 4-58
データ・タイプによる改善 4-57
バック 4-58
パフォーマンスの改善 4-63
パラメータマッキング
受け渡し 4-42
受け渡し (C++) 8-81
操作記述子を使用した受け渡し 8-86
デフォルト受け渡しスタイル 8-85
C++ から別の HLL への受け渡し
8-80
ハンドラーのタイプ 8-8
ハンドリング、エラーの 8-34
ヒープ記憶域取得 (CEEGTST) 3-9
可能 API 4-54
ヒープ・ストレージ
管理 4-53
タイプ 4-54
デフォルト 4-54
ユーザー作成の 4-54
光ディスクリートサポ (QOPT) 6-28
引数
受け渡し
制御言語プログラムから ILE C++
プログラムへの 8-87
操作記述子 8-86, 8-100
データ・タイプ一致要件 8-100
デフォルト・スタイル 8-85
バック 10 進数 8-131
ILE、CL、COBOL、または RPG
リング・サポ 8-82
表示装置ファイル作成（CRTDSPF）コマンド 7-57
表示装置ファイル作成（CRTRPTF）コマンド 7-66
表示装置ファイル作成（CRTPRF）コマンド 7-66

プリンター・ファイル作成（CRTPRF）コマンド 7-66
プリンター・ファイル作成（CRTPRTF）コマンド 7-66

プレースポイント
ジョブ 5-21
スレッド 5-21

プログラム
更新 4-14
サイズに対するデバッグ・データの影響 5-1
作成 4-9
実行 4-38
実動の準備 5-54
終了 4-48
ステップイン 5-31
ステップオーバー 5-30
ステップスルー 5-30

装置
入出力フィードバック域 7-51
対サービス・プログラム 4-15
デバッグ 5-13, 5-20, 5-25
デバッグの準備 5-12, 5-54
名前変更 8-47
ユーザーが作成した CL コマンドによる実行 4-46
呼び出し 4-41, 8-46
呼び出し (TRFRCLI) 4-44
呼び出し、ライブラリ修飾の使用 8-48
呼び出し、リモートで指定の使用 8-80, 8-126
呼び出しスタック項目 8-46
ILE C++ の呼び出し 8-95
ILE C++ への移植 8-58
ILE の呼び出し 8-46, 8-88

OPM の呼び出し 8-119

プログラム入力リプロシージャー（PEP）
識別 4-7
プログラム・オブジェクトの構成要素
として 4-8

プログラム入力リプロシージャー、説明 8-46

プログラム記述ファイル 7-23
プログラム除去（RMVPGM）コマンド 5-16

プログラム追加（ADDPGM）コマンド 5-16

プログラムのデバッグ
開始前 5-1, 5-12, 5-13
制約 5-2
ILE デバッグ・セッションへの OPM
プログラムの追加 5-18

プログラムへのモジュールのバインド
4-5, 4-7

プログラム・エクスポート開始
(STRPGMEXP) コマンド 4-79
プログラム・ソース、表示 5-18
プロシージャー
再帰呼び出し 8-47
ステップイン 5-34
ステップオーバー 5-33
名前変更 8-47
非再帰呼び出し 8-47
呼び出し 6-21, 8-46, 8-99
呼び出し、リネッジ指定の使用 8-125
プロシージャおよびデータ項目のコーディング 4-79
プロシージャ呼出し
静的 4-8
呼び出しスタック項目 8-46
ILE C++およびILE COBOL プロシージャ呼び出し 8-78
プロシージャー・ポインタ呼び出し、説明 8-100
プロセス・コマンド (QCAPCMD)
API 4-44
分散 (DDM)
定義 7-22
バイナリ・ストリーム・ファイルとしてファイルを開く 7-29
文書ライブラリー・サービス
(QDLS) 6-27
ページ不在 4-70
ヘッダ・ファイル
bcd.h 8-62
GENCSR による作成 10-1
lecond.h 8-17
milib.h 8-79
recio.h 6-1
studio.h 6-20
xstd.h 6-20
<bcd.h> 8-139
<decimal.h> 8-127
<errno.h> 8-6
<recio.h> 7-27
<stdio.h> 8-26
<stdio.h> 6-6
変換
コード・ページ、ワイド文字リテラル 9-5
ソース・ファイル CCSID 9-2
ソース・ファイル内のストリーミング・リテラル 9-3
バック 10 進数型者整数型へ 8-129
バック 10 進数型からバック 10 進数型へ 8-127
バック 10 進数型から浮動小数点型へ 8-130
バック 10 進数またはゾーン・データ、自動的 7-22
バック 10 進データ型から 8-127
変換 (続き)
文字列に保管されたゾーン 10 進数データ 7-22
文字列に保管されたバック 10 進数データ 7-22
REXX 変数 4-44
*CLD オブジェクト・タイプから
*LOCALize オブジェクト・タイプに 9-10
変数
値の表示 5-36
値の割り当て 5-38
環境変数 9-15
グローバル 4-57, 4-62
最適化 4-57
式値の判別 5-37
式を代入 5-37
スカラー 5-38
デバッガ中的値の変更 5-38
静的 4-57
正確に表示 5-54
重複名の処理 4-11
デバッガ 5-36
名前の等価化 5-39
評価 5-36, 5-44
有効範囲の定義 5-38
レジスタ、ストレージ、クラスの使用 4-58
I6 進数としての表示 5-40
DDS 変数名 7-4
F11 を使用した表示 5-36
REXX 4-44
volatile 修飾子 4-57
変数、またはコマンド、名前の等価化 5-39
変数定数またはコマンドを指定した名前の等価化 5-39
ポインター
オープン 8-38
ポインタのパスへの影響 4-67
ILE の制約 8-38
オープン・ポインタ以外のポインター 8-39
およびデータ・モデル 8-171
関数
OS リンケージ付き 8-40
関連アクセスを削減 4-70
キャスト 8-42
シャロー・コピーとディープ・コピー 4-70
制約
キャスト 8-42
制約をキャストする 8-42
タイプ 8-37
定義 4-58
デラスベース 8-171
ポインター (続き)
クラスベースに関する変更 8-171
動的キャスト 8-181
入出力、およびキュー NULL フィールド・マップ 7-27
フォーマンスの向上 4-70
比較 4-67
非互換ポインタ・タイプ 8-41
評価 5-36, 5-44
ポインターの呼び出し 8-43
ポインタ変数の宣言 8-39
ラベル 8-38
16 バイト 8-171
8 バイト 8-171
FILE ポインタのキャスト 7-41
IBM i 使用法 8-37
IBM i タイプ 8-37
ILE の制約 8-38, 8-39
ISO C 定義 8-37
_null_map 7-27
ポインターのキャスト、記述 8-42
ポインター変数の宣言 8-39
ポインタをキャストする、記述 8-42
保管ファイル
使用 7-77
入出力に関する考慮事項 7-77
バイナリ・ストリーム関数 7-77
バイナリ・ストリーム・ファイルとしてオープン 7-77
レコード関数 7-78
レコード・ファイルとしてオープン 7-77

【マ行】
マクロ、拡張 4-61
マッピング
C++ クラスから C 構造体 8-54
IBM i 準拠名への内部 ID 8-47
無条件停止点
除去 5-20
設定 5-20
命名規則
フィールド・タイプ 4-61
ロング・ファイル名 7-7
#pragma mapinc 7-7
#pragma mapinc lname オプション 7-3
#pragma mapinc 名前生成 7-3
メッセージ / マイナーステータスコード 7-42
メッセージ/マイナーステータスコード、検査 8-7
メッセージのモニター (MONMSG) コマンド 4-48
メッセージング・サポート 4-15
文字エスケープ・コード 9-5
文字セット 9-1, 9-9
[ヤ行]
ユーザーリリースシーダー (UEP) 4-7
プログラム・オブジェクトの構成要素
として 4-8
ユーザースーピーグデータ・ストリーム
(UDDS) 7-54
コード・ページ変換での効果 9-5
複数の文字列変換サポート 9-4
文字セット・サポートの効用 9-5
ワイド文字変換サポート 9-4
様式
時刻 9-9
システム応答 9-9
数値 9-9
通貨数値 9-9
日付 9-9
SYSIFCOPT 6-30
呼び出し
プログラム 4-41, 4-44
プログラム、ライブラリ修飾の使用
8-48
プログラム、リンケージ指定の使用
8-80, 8-126
プロシージャ 6-21, 8-99
プロシージャ、リンケージ指定の使用
8-125
メッセージ持ち行列 8-3
呼び出しスタック項目 8-46
CALL コマンド 8-50
C++プログラム 8-95
C++ プロシージャ 8-50
ILE C++プログラム 8-88
OPM プログラム 8-119
呼び出し (CALL) コマンド 5-31, 7-36
使用 4-41
パラメーターの変更 4-43
パラメーターをプログラムに渡す
4-43

[ラ行]
ライブラリ (QSYS.LIB) 6-26, 6-32
ライブラリ名、指定 4-67
リスト 5-13
リスト・ビュー 5-12
リスト・ビューの作成 5-13
ルート・ソース 5-13
ルート・ソース・ビューの作成 5-13
例
サンプル ILE C アプリケーションの
作成 4-75
例外ハンドリング 4-60
C アプリケーションのデバッグ 5-4,
5-10
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外
例外：
ワイルド文字リテラル、コード・ページ変換 9-5
ワイルド文字リテラル、ユニコード・サポート 9-4

数字
10 进数タイプ 5-2
2 バイト文字セット (DBCS) プログラミングの考慮事項
コード化文字セット ID (CCSID) 9-1
2 バイト文字セット (DBCS) 9-1
3 文字表記 9-4

A
ALIAS キーワード 7-4

B
bcd.h ファイル 8-62

C
C 構造体における位置合わせの問題、回避 7-13
C モジュールの作成 (CRTCMOD) コマンド 5-13
C リンケージ、関数に指定 8-83
C ローカル・マイグレーション・テーブル 9-10
CCSID の指定 9-2
characters グラフィック 9-1
コード化文字セット 9-1
種別 9-9
順序付け 9-9
照合 9-9
大小文字変換 9-9
配列の評価 5-36
CHGMOD コマンド 5-56
CL リンケージ 8-82
COBOL リンケージ 8-82
C++ オブジェクト
C プログラムでの使用 8-51, 8-55
C++ クラス 8-160
C++ テンプレートの示的インタフェース 8-165
C++ テンプレートの示的インタフェース 8-166
C++ モジュールの作成 (CRTCPPMOD) コマンド 8-161
C++ 呼び出し規則 8-46

D
DBCS 文字 9-5
DBGVIEW バラメーター
オプション 5-1
デバッグのためのプログラムの準備 5-12
CRTBNDC コマンド 5-13
CRTBNDCP コマンド 5-13
CRTCMOD コマンド 5-13
CRTCPPMOD コマンド 5-13
ILE オプション 5-18
*ALL オプション 5-13
*EXPPAC オプション 5-13
*LIST オプション 5-13
*NONE オプション 5-15
*SHOWINC オプション 5-13
*SHOWSKP オプション 5-13
*SHOWSYS オプション 5-13
*SHOWUSR オプション 5-13
*SOURCE オプション 5-13
DDM ファイル
作成 7-23
入出力に関する考慮事項 7-29
バイナリ・ストリーム関数 7-30
バイナリ・ファイルとしてオープン 7-29
有効なキーワード・バラメーター 7-28
レコード関数 7-28
レコード・ファイルとしてオープン 7-27
DDM ファイル作成 (CRTDDMF) コマンド 7-29

E
errno 錆の検査 8-6
初期化および再初期化 8-6
ヘッダー・ファイル 8-6
EVAL デバッグ・コマンド
サンプル・ソース 5-48

F
close() 関数 7-57
fopen() 関数 6-2, 6-5, 6-6, 6-8, 6-11, 6-12, 6-17, 6-18, 6-32, 6-33, 7-57, 7-66
fread() 関数 6-19, 7-57
freopen() 関数 6-4, 6-8, 6-12, 6-17
fopen() 関数 7-57
fwrite() 関数 7-57, 7-66

G
GENCSRC ユーテリティー 7-2
GENCSRC ユーテリティー、説明 10-1
GENCSRC ユーテリティー、レベル検査 7-9

H
HLL 固有ハンドラー
使用 8-24

I
IBM i 用 2 進行 10 進数クラス・ライブラリー 8-62
ICF ファイル
作成 7-60
使用 7-59
デフォルトのプログラム変更 7-61
入出力に関する考慮事項 7-59
バイナリ・ストリーム関数 7-60
バイナリ・ストリーム・ファイルとしてオープン 7-60
標識 7-41
分離標識域 7-41
メジャーミニアムナビコード 7-42
レコード関数 7-61
レコード単位の処理 7-60
レコード・ファイルとしてオープン 7-60
ICF ファイル作成 (CRTICFF) コマンド 7-60
IFS インタフェースの 64 ビット・バージョン、使用可能化 6-45
IFS の 64 ビット・バージョン 6-30
ILE C 関数、説明 8-99
ILE C 標準 4-63
ILE 条件ハンドラー 8-18
ILE 条件ハンドラー (CEEHDLR) バインド可能 API 8-17
ILE 条件ハンドラー (CEEHDLU) バインド可能 API 8-17
ILE ソース・デバッグ 5-4, 5-10
開始 5-13
コマンド 5-2
制御
関数呼び出し 5-2
タイプ・キャスト 5-2
10 進数タイプ 5-2
OPM プログラムの受け入れを可能にする 5-18
ILE リンケージ 8-82
INDARA キーワード 7-42

X-10 IBM i: ILE C/C++ プログラマーの手引き
Integrated Language Environment (ILE)
活動化 4-40
グループの活動化 4-14
条件ハンドラー 8-17
ISO C 標準 4-63

L
LAN Server/400 (QLANSrv) 6-28
LC_ALL ロケール変数 9-14
LC_COLLATE ロケール変数 9-14
LC_CTYPE ロケール変数 9-14
LC_MONETARY ロケール変数 9-14
LC_NUMERIC ロケール変数 9-14
LC_TIME ロケール変数 9-14
LC_TOD ロケール変数 9-14
LOCALETYPE オプション
CRTBINDC コマンド 9-13
CRTCMOD コマンド 9-13

M
modules
サイズに対するデバッグ・データの影響 5-1
デバッグ・オプションの設定 5-19
ビューの変更 5-19
プログラム識別情報、変更 5-54
別のビュー 5-19

O
open() 関数 6-18
open() メンバー関数 6-5, 6-9, 6-13
OS リンケージ関数 (C++) 8-82

P
perror() 関数 8-6

Q
QINLINE 6-3
QTEMP ライブラリー 6-3
QXX 関数 8-73

R
recio.h ファイル 6-1
RIOFB_T 4-64
RPG リンケージ 8-82
RTBND. パフォーマンスの最適化に使用 (C++) 8-174

S
setbuf() 関数 6-2
setvbuf() 関数 6-2
signal
SIGABRT 8-25
SIGFPE 8-25
SIGILL 8-25
SIGINT 8-25
SIGIO 8-25
SIGOTHER 8-25
SIGSEGV 8-25
SIGTERM 8-25
SIGUSR1 8-25
SIGUSR2 8-25
SIG_DFL 8-25
SIG_IGN 8-25
SLTFLD キーワード 7-9
SQL ILE C オブジェクト作成 (CRTSQLID) コマンド 4-6
stderr 6-3
stdio.h ファイル 6-20
STEP デバッグ・コマンド
イントゥ, 別へ 5-31
オーバー 5-30
中へ 5-31
 strerror() 関数 8-6

T
TEMPINC 8-167
TEXT キーワード 7-4
try catch throw (C++), 使用 8-152
try-catch-throw (C++), 使用 8-32
TYPEDEFFX キーワード 7-9

V
va_arg マクロ 8-133
VREF リンケージ、指定 8-82

X
xxfblk.h ファイル 6-20

【特殊文字】
#include ディレクティブ 7-2, 8-168
#pragma argument ディレクティブ 8-60, 8-76, 8-82
#pragma cancel_handler ディレクティブ 8-32
#pragma convert ディレクティブ 9-1, 9-3, 9-5, 9-8
#pragma descriptor ディレクティブ 8-101
#pragma disable_handler ディレクティブ 8-9
#pragma enum 8-71
#pragma exception_handler ディレクティブ 4-59, 8-8, 8-9, 8-12
#pragma implementation ディレクティブ 8-167, 8-169
#pragma linkage (PGMNAME, OS) ディレクティブ 8-79
#pragma linkage ディレクティブ 8-60, 8-62
#pragma map ディレクティブ 8-47, 8-60, 8-79, 8-83
#pragma mapinc ディレクティブ 7-5, 7-27
データベース・ファイル 7-17
複数のレコード要素 7-18
ヘッダー記述 7-2
レコード要素 7-3
ロング・レコード・フィールド名 7-7
C 構造体タイプ定義の作成 7-1
#pragma nosigtrunc ディレクティブ 8-138
#pragma pack ディレクティブ 8-69
#pragma undeclared ディレクティブ 8-171
#pragma HANDLE 制御処置 8-11
#pragma HANDLE_NO_MSG 制御処置 8-11
#pragma IGNORE 制御処置 8-11
#pragma IGNORE_NO_MSG 制御処置 8-11
#pragma INVOKE 制御処置 8-11
#pragma TS_calloc() 8-172
#pragma TS_free() 8-172
#pragma TS_malloc() 8-172
#pragma TS_realloc() 8-172
#pragma EXIT_MSGID グローバル変数 8-8
Racquire() 関数 7-46, 7-58, 7-59, 7-61
Rclose() 関数 7-59
Rdevatr 7-41
Rdevattr() 関数 7-59
Rfeed() 関数 7-59, 7-70
Rfread() 関数 7-70
Rformat() 関数 7-58, 7-59
Rindara() 関数 7-59
Riosbk() 関数 6-20, 7-51, 7-59
Ropen() 関数 6-1, 6-5, 6-6, 7-57, 7-58
Rreadd() 関数 7-59
Rpgmdev() 関数 7-49, 7-58, 7-59, 7-61
Rreadn() 関数 7-58, 7-59, 7-61
Rreadd() 関数 7-58, 7-59
Rupfb() 関数 7-59
Rwrited() 関数 7-59
Rwritedt() 関数 7-59
Rwrite() 関数 7-59
Rwrread() 関数 7-59
__ptr128 8-171
__ptr64 8-171