Developing applications using the Eclipse C/C++ Development Toolkit

Creating a C++ application using the Standard Template Library and the CDT

Michael Galpin

July 10, 2007

When most people think "Eclipse," they think "Java™ IDE." Eclipse is much more. With the proper plug-ins, one can use Eclipse to work with PHP, Ruby, and Groovy, C, and C++. In this article, you get started with the the Eclipse C/C++ Development Toolkit (CDT). We build a simple application in C++ using the CDT and the C++ Standard Template Library (STL) to streamline the process and take advantage of some of the powerful features of C++.

This article, which is a follow-up to "C/C++ development with the Eclipse Platform," is intended for C++ developers who want to learn C++ development using the Eclipse CDT. A simple C++ application is developed in the article. The application makes use of the C++ STL. Readers should be familiar with the STL, as well as with basic object-oriented programming principles such as inheritance and polymorphism. A familiarity with Eclipse will be helpful, but is not required.

Before we start

You need to install the following:

- **Eclipse**
  We're using the CDT, which is a plug-in to Eclipse, so of course you need Eclipse. The article uses Eclipse V3.2.

- **Java Runtime Environment**
  We're building a C++ application, but we're using Eclipse. Eclipse is a Java application itself, so it needs a Java Runtime Environment (JRE). The article uses Eclipse V3.2, which requires a JRE of V1.4 or higher. If you want to also use Eclipse for Java development, you'll need a Java Development Kit (JDK).

- **Eclipse C/C++ Development Toolkit (CDT)**
  This article is about the CDT, so you'll need it, of course. For instructions on installing the CDT on early versions of Eclipse, read a "C/C++ Development with the Eclipse Platform" (developerWorks 2003).
• **Cygwin**  
If you're using Microsoft Windows®, you will find Cygwin — which provides a Linux®-like environment on Windows — helpful.

• **GNU C/C++ Development Tools**  
The CDT uses the standard GNU C/C++ tools for compiling your code, building your project, and debugging the applications. These tools are GNU Compiler Collection (GCC) for C++ (g++), make, and the GNU Project Debugger (GDB). If you're a programmer using Linux or Mac OS X, there's a pretty good chance these tools are installed on your machine. The article contains instructions for setting up these tools for Windows.

**The Eclipse CDT**

The Eclipse CDT is an Eclipse plug-in that transforms Eclipse into a powerful C/C++ IDE. It was designed to bring many of the great features Eclipse enjoyed by Java developers to C/C++ developers, such as project management, integrated debugging, class wizards, automated builds, syntax coloring, and code completion. When Eclipse is used as a Java IDE, it leverages and integrates with the JDK. Similarly, the CDT leverages and integrates with standard C/C++ tools, such as g++, make, and GDB. This has lead to it becoming very popular on Linux, where those tools are readily available and used for most C++ development. The CDT can be set up on Windows to use the same tools. There is also an ongoing effort to get the CDT to work with Microsoft's C++ tools to make it even more attractive to Windows C++ developers.

**Installing the CDT**

We start by assuming you installed Eclipse and can run it. If not, consult Eclipse's Web site for getting up and running. Let's install the CDT. The CDT is an Eclipse plug-in, so it uses Eclipse's Software Updates feature. Select **Help > Software Updates > Find and Install**.

**Figure 1. Eclipse Software Updates**

Next, you'll want to choose **Search** for new features to install.
If you're using a newer version of Eclipse, the Callisto or Europa discovery sites should be included. (Editor’s note: Since this was written in April 2007, the Europa release was still in the planning stages. However, installing Europa is expected to be similar to Callisto.) Simply select it and click Finish.
Figure 3. Callisto Discovery Site

Eclipse might ask you to choose from a list of mirror sites for the Callisto Discovery Site. Pick whatever one seems closest to you. You should see a list of plug-ins from the Callisto Discovery Site. You'll want to select C and C++ Development and click **Next**.
Figure 4. Available Callisto plug-ins

You'll be asked to accept the license for the CDT. Once you've done that, you can click **Next**. You'll see a summary of what's going to be downloaded and installed. Simply click **Finish**.
Eclipse’s Update Manager will then download the CDT plug-in from the mirror site you selected earlier. The CDT is about 11 MB total, so this could take a few minutes, depending on your Internet connection speed. Once everything is downloaded, you’ll be asked to confirm that you want to install the new features. Click **Install All**.
After you finish installing CDT, you'll be asked to restart Eclipse. Go ahead and do that. Once Eclipse restarts, the CDT will be ready to go.

**Windows configuration**

If you're running Eclipse on Linux or Mac OS X, you're ready to start using the CDT to develop a C++ application. If you're on Windows, there might be a few more steps. As mentioned, CDT relies on the standard GNU C++ development tools: g++, make, and GDB. These are usually included on Linux or Mac OS X. They're usually not included with Windows. But don't worry. These tools can be easily installed on Windows. Perhaps the easiest way is to install Cygwin. Cygwin provides Linux-like environment on Windows (see Related topics). When installing Cygwin, you'll be asked to pick the packages you want to install. Make sure to go into the development section and select gcc: g++, make, and GDB. This will cause their prerequisites to be installed, too.

Once you're done installing Cygwin, you'll need to add g++, make, and GDB to your path. The easiest way to do this is to add Cygwin's bin directory to your path, since that's where g++, make, and GDB can be found. Once that's done, restart Eclipse.

**Playing the lottery**

At this point, we should be ready to start developing our application with CDT. Let's pause to figure out what we want to develop. The sample application is a simple command-line program for generating lottery numbers. Many states have lotteries, and the rules vary quite a bit. We'll allow the user to pick which state lottery he wants to generate numbers for. This will provide us a good way to use C++'s support for polymorphic behavior.
Creating the project

Eclipse uses the concepts of perspectives to allow for various plug-ins to customize their commands and views. Eclipse starts off by default in the Java perspective. CDT includes its own perspective, so we'll want to switch to that. To do that, select **Window > Open Perspective > Other**. You should see a list of perspectives available to you. Select the C/C++ perspective and click **OK**.

**Figure 7. Select C/C++ perspective**

Eclipse should now look something like Figure 8.
Eclipse organizes your code into projects, so we'll want to create a new project. Select **File > New > Managed Make C++ Project**.

You might have noticed there were several different options for the project. We wanted a C++ project. We selected a "Managed Make," since that will allow Eclipse to create the make file for us. You could select a "Standard Make" flavor and write your own make file. We should now be in the New Project wizard, where we'll name our project **Lottery** and click **Finish**.

This will create an empty project, which you should see in the C/C++ Projects window. Right-click on the project and select **New > Source Folder**. This will bring up the "New Source Folder" wizard, where we’ll name our folder **src** and click **Finish**.
Basic lottery

We're ready to start creating some code. We'll start by creating the executable of our application. Right-click on the source folder we just created and selected New > Source File, as shown in Figure 10.

Let's create an empty main method for now. This is just a placeholder; we'll add more to this after we've created the rest of our project.

```c
int main()
{
    return 0;
}
```

Save your project, and Eclipse will make it for you automatically. You should see some output in the console indicating that it compiled successfully.

We're ready to create our first class. Right-click on the source folder we just created and select New > Class.

Figure 10. New class

This should bring up the New Class wizard. We'll give our class a namespace lotto, and we'll call our class Lottery.
Eclipse will now create stubs for your class. CDT does a lot of nice things for you. It generates the appropriate compiler directives in the header file. It encourages best practices by generating separate interface (Lottery.h) and implementation (Lottery.cpp) files. It encourages another best practice by making your class' destructor virtual. We can enter the source code for these classes as seen in Listings 1 and 2.

**Listing 1. Lottery.h**

```cpp
#ifndef LOTTERY_H_
#define LOTTERY_H_
#include <string>
#include <sstream>
#include <cstdlib>
#include <time.h>
using namespace std;
namespace lotto
{
    class Lottery
    {
        protected:
            int ticketSize, maxNum;
            int* generateNumbers();
        public:
```
Listing 2 shows the implementation file for the \texttt{Lottery} class.

\textbf{Listing 2. Lottery.cpp}

\begin{verbatim}
#include "Lottery.h"

namespace lotto {

Lottery::Lottery(int size, int num) {
    this->ticketSize = size;
    this->maxNum = num;
}

Lottery::~Lottery() {
}

int* Lottery::generateNumbers() {
    int* allNums = new int[this->maxNum +1];
    for (int i=0; i <= this->maxNum; i++)
        allNums[i] = 0;
    int* nums = new int[this->ticketSize];
    for (int i=0; i < this->ticketSize; i++)
        if (allNums[val])
            i--;
        else
            { 
             nums[i] = val;
             allNums[val] =1; 
            }
    return nums;
}

string Lottery::printTicket()
{
    ostringstream str;
    int* nums = this->generateNumbers();
    for (int i=0;i< this->ticketSize; i++)
        { 
            str << *nums << ' '; 
            nums++; 
        }
    return str.str();
}

}
\end{verbatim}
What's this code doing? Well, our Lottery class has two attributes. The ticketSize attribute is the number of numbers on the lottery ticket. The maxNum is the maximum number on the ticket. Later, we'll use the Florida state lottery as an example. There, you pick six numbers from 1 to 53, so ticketSize would be 6 and maxNum would be 53.

The generateNumbers method generates an array of numbers corresponding to the numbers on a lottery ticket. It uses the STL function `rand()` to generate numbers randomly. The allNums array is used to keep track of what numbers have been generated so far, so we can make sure we don't get a duplicate number on our ticket. Finally, the printTicket() creates a string representation of our ticket.

When you save the files, Eclipse builds your project automatically. Again, if you save the project, it should be compiled and you should see compilation messages in the console, as shown in Listing 3.

**Listing 3. Compiler output in console**

```bash
**** Build of configuration Debug for project Lottery ****
make -k all
Building file: ../src/Main.cpp
Invoking: GCC C++ Compiler
g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"src/Main.d" -MT"src/Main.d"
-o"src/Main.o" "../src/Main.cpp"
Finished building: ../src/Main.cpp

Building target: Lottery
Invoking: MacOS X C++ Linker
g++ -o "Lottery" ./src/Lottery.o ./src/Main.o
Finished building target: Lottery

Build complete for project Lottery
```

**MegaLottery class**

You might have noticed that we made the printTicket() method virtual when it was declared in the header file. That will allow us to subclass Lottery and override this method. We wanted to do that because some states have a lottery with a "mega" number. This is a separately drawn number that any ticket must match in addition to the other numbers drawn. Let's create a MegaLottery class for these states that will subclass Lottery.

Once again, right-click on our source folder and select New > Class, as we did earlier. This time in the New Class wizard, we'll declare our new class in the same namespace, but call it MegaLottery.
To subclass `Lottery`, select the **Add** button next to the Base Classes section. This will bring up the Choose Base Class dialog. You can start typing the name of the class, and Eclipse will narrow the list of base class candidates quickly. You'll want to select `Lottery` and click **OK**.
Figure 13. Choose base classes

We can enter the code for `MegaLottery`, as shown in Listings 4 and 5.

Listing 4. MegaLottery.h

```cpp
#ifndef MEGALOTTERY_H_
#define MEGALOTTERY_H_

#include "Lottery.h"

using std::string;

namespace lotto
{
    class MegaLottery : public lotto::Lottery
    {
```
protected:
    int maxMegaNum;
public:
    MegaLottery(int,int,int);
    virtual ~MegaLottery();
    string printTicket();
};
}
#endif /*MEGALOTTERY_H_*/

Listing 5 shows the implementation file for the MegaLottery class.

Listing 5. MegaLottery.cpp

#include "MegaLottery.h"

using namespace std;

namespace lotto
{
    MegaLottery::MegaLottery(int ticketSize, int maxNum, int maxMegaNum) :
        Lottery(ticketSize,maxNum)
    {
        this->maxMegaNum = maxMegaNum;
    }

    MegaLottery::~MegaLottery()
    {
    }

    string MegaLottery::printTicket()
    {
        ostringstream tick;
        tick << Lottery::printTicket() << " Mega: ";
        int mega = 1 + rand() % this->maxMegaNum;
        tick << mega;
        return tick.str();
    }
}

The main difference between Lottery and MegaLottery is that MegaLottery has an extra attribute maxMegaNum. This is the max value that the mega number can take. It overrides the printTicket() method. It uses the base class to generate the first part of the ticket, then it generates the mega number and appends it to the string representation of the ticket.

We just need a way to create the various lotteries. We'll use a class Factory Pattern to do this. We'll do this by adding a LotteryFactory class. We want all Lotteries to come from the same factory, so we'll make LotteryFactory a singleton. The code for it is in Listings 6 and 7.

Listing 6. ifndef LOTTERYFACTORY_H_

#define LOTTERYFACTORY_H_

#include "Lottery.h"

namespace lotto
{

class LotteryFactory
{
public:
    enum State { California, Florida };  
    static LotteryFactory* getInstance();
    Lottery* getLottery(State);
private:
    LotteryFactory();
};

Listing 7 shows the implementation file for the LotteryFactory class.

Listing 7. LotteryFactory.cpp
#include "LotteryFactory.h"
#include "MegaLottery.h"

namespace lotto
{
    LotteryFactory::LotteryFactory()
    {
        srand(time(NULL));
    }

    Lottery* LotteryFactory::getLottery(State s)
    {
        if (s == LotteryFactory::California)
        {
            Lottery* lotto = new MegaLottery(5,47,27);
            return lotto;
        }
        else
        {
            Lottery* lotto = new Lottery(6,53);
            return lotto;
        }
    }

    LotteryFactory* LotteryFactory::getInstance()
    {
        static LotteryFactory instance;
        return &instance;
    }
}

The LotteryFactory has an enum of the different types of lotteries. We've only put in Florida and California in the example, but it shouldn't be hard to add as many as you want. The LotteryFactory's constructor seeds the rand() function used by our lottery classes. We just need to implement our executable's main method.

Listing 8. Main.cpp
#include <iostream>
#include "LotteryFactory.h"
using namespace lotto;
int main()
{
    LotteryFactory* factory = LotteryFactory::getInstance();
    cout << "What lottery do you want to play?" << endl;
    cout << "(1) California, (2) Florida" << endl;
    int cmd;
    scanf("%d", &cmd);
    Lottery* lotto = 0;
    switch (cmd)
    {
    case 1 :
        lotto = factory->getLottery(LotteryFactory::California);
        break;
    case 2 :
        lotto = factory->getLottery(LotteryFactory::Florida);
        break;
    default :
        cout << "Sorry didn't understand that" << endl;
    }
    cout << "Ticket: " << lotto->printTicket() << endl;
    delete lotto;
    return 0;
}

Running the program

We're ready to run our program. Select Run > Run.

Figure 14. Choose base classes

Select C/C++ Local Application and click the New button.
Figure 15. New C/C++ run profile

This will bring up the Create run configuration interface for the Lottery project. You'll need to select its executable by clicking the Search Project button.

Figure 16. Search project for executable
You can select the binary that Eclipse created for you and click **OK**.

**Figure 17. Search project for executable**

Just click **Run**, and the program should run in your console. The code below shows some sample output.

```
What lottery do you want to play?
(1) California, (2) Florida
2
Ticket: 12 28 24 15 35 22
```

### Debugging the program

Our program should run fine, but let's take a look at debugging the application. First, create a breakpoint in our code. Pick a line and right-click next to it and select **Toggle Breakpoint**.
Figure 18. Create breakpoint

We need to create a debug configuration, much like we created a run configuration. Select Run > Debug.

Figure 19. Create debug configuration

This should bring up the Debug configuration. This is based on the Run configuration, and you shouldn't need to change anything. Just click Debug.
Once the debugger starts, it will prompt you to switch to the Debugger perspective. Do so. Notice that in the configuration we set things to break automatically at the startup of our main method. Thus, the debugger should break immediately and you should see a screen something like Figure 21.
Summary

We've built and debugged our lottery application. You can easily add more lottery schemes to it. Some of these could involve additional subclasses. CDT makes it easier than ever to create these classes and class hierarchies, and to run and debug the application to test it.
Related topics

- **Integrate an external code checker into Eclipse CDT** (Alex Ruiz, developerWorks): Learn how to execute C/C++ code analysis tools with Codan in Eclipse.
- Get an overview of the CDT in "C/C++ development with the Eclipse Platform."
- Dig deep into the CDT's architecture in the five-part series titled "Building a CDT-based editor."
- As someone interested in C/C++ development, you might want to check out a trial of IBM's **XL C/C++ compiler for Linux** or **AIX®**.
- Windows developers can learn about migrating to the CDT in "Migrate Visual Studio C and C++ projects to Eclipse CDT."
- Windows developers can also check out the **CDT-MSVC project**, a project for incorporating Microsoft's compiler and debugger with CDT.
- Learn about **MinGW**, the GNU C/C++ tools for Windows included with Cygwin.
- Download **Cygwin** a Linux-like environment for Windows. It consists of two parts: A DLL that acts as a Linux API emulation layer providing substantial Linux API functionality and a collection of tools that provide a Linux look and feel.
- The **Eclipse C/C++ Development Toolkit (CDT)** download information contains the latest information about the available versions of CDT.
- Check out the "Recommended Eclipse reading list."
- For an introduction to the Eclipse platform, see "Getting started with the Eclipse Platform."

© Copyright IBM Corporation 2007
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)