Using Abstraction to Extend Functionality ([1], p. 12)

The American Heritage Dictionary defines the term abstraction as “the process of leaving out of consideration one or more properties of a complex object so as to attend to others.” Basically, this means that we need to pick and choose the parts of our objects that are important to us. To abstract our data, we choose to encapsulate those portions of the object that contain certain basic types of functionality (base objects) — that way we can reuse them in other objects that redefine that functionality. Such basic objects are called, not surprisingly, base classes. The extended objects are called inherited classes. Together they form a fundamental principle of C++. Abstraction in C++ is provided through the **pure virtual method**. A **pure virtual method** is a method in a base class that must be implemented in any derived class in order to compile and use that derived class.

Virtual methods are one of the best timesavers available in C++. By allowing people to override just small pieces of your application functionality — without requiring you to rewrite the entire class — you give them the chance to extend your work with just a small amount of effort of their own. The concept of abstraction is satisfied through the virtual method because the base-class programmer can assume that later programmers will change the behavior of the class through a defined interface. Because it’s always better to use less effort, using virtual methods means your code is more likely to be reused — leading to fewer errors, freeing up more time to design and develop quality software.

Creating a Mailing-List Application

This concept is just a little bit abstract (pardon the pun), so here’s a concrete example to show you how abstraction really works: Assume you want to implement a mailing list for your company. This mailing list consists of objects (called mailing-list entries) that represent each of the people you’re trying to reach. Suppose, however, that you have to load the data from one of two sources: from a file containing all the names, or directly from the user’s command line. A look at the overall “flow” of this application reveals that the two sides of this system have a lot in common:

To handle input from a file, we need some place to store the names, addresses, cities, states, and zip codes from a file. To handle input from the command line, we need to be able to load that exact same data from the command line and store it in the same place. Then we need the capability to print those mailing-list items or merge them into another document. After the input is stored in memory, of course, we don’t really care how it got there; we care only how we can access the data in the objects. The two different paths, file-based and command-line-based, share the same basic information; rather than implement the information twice, we can abstract it into a container for the mailing-list data. Here’s how to do that:

1.In the code editor of your choice, create a new file to hold the code for the definition of the class.

In this example, the file is named ch02.cpp , although you can use whatever you choose.

2.Type the code from Listing 2-1 into your file, substituting your own names for the italicized constants, variables, and filenames.

L ISTING 2-1: T HE B ASE M AILING L IST E NTRY C LASS

#include <string>

#include <iostream>

#include <stdio.h>

class BaseMailingListEntry

{

private:

std::string sFirstName;

std::string sLastName;

std::string sAddressLine1;

std::string sAddressLine2;

std::string sCity;

std::string sState;

std::string sZipCode;

public:

BaseMailingListEntry(void)

{

}

BaseMailingListEntry( const BaseMailingListEntry& aCopy )

{

sFirstName = aCopy.sFirstName;

sLastName = aCopy.sLastName;

sAddressLine1 = aCopy.sAddressLine1;

sAddressLine2 = aCopy.sAddressLine2;

sCity = aCopy.sCity;

sState = aCopy.sState;

sZipCode = aCopy.sZipCode;

}

virtual bool First(void) = 0; // A pure virtual function

virtual bool Next(void) = 0; // Another pure virtual function

// Accessors

std::string getFirstName() { return sFirstName; };

std::string getLastName() { return sLastName; };

std::string getAddress1() { return sAddressLine1; };

std::string getAddress2() { return sAddressLine2; };

std::string getCity()

{ return sCity; };

std::string getState()

{ return sState; };

std::string getZipCode()

{ return sZipCode; };

void setFirstName(const char \*strFirstName)

{ sFirstName = strFirstName; };

void setLastName(const char \*strLastName)

{ sLastName = strLastName; };

void setAddress1( const char \*strAddress1)

{ sAddressLine1 = strAddress1; };

void setAddress2( const char \*strAddress2)

{ sAddressLine2 = strAddress2; };

void setCity(const char \*strCity)

{ sCity = strCity; };

void setState(const char \*strState)

{ sState = strState; };

void setZipCode( const char \*strZipCode )

{ sZipCode = strZipCode; };

};

3.Save the file in your source-code editor.

4.Using your favorite code editor, add the code in Listing 2-2.

You may optionally save this code in a separate header file and include that header file in your

main program as well.

L ISTING 2-2: T HE F ILE M AILING L IST E NTRY C LASS

class FileMailingListEntry : public BaseMailingListEntry

{

FILE \*fpIn;

public:

FileMailingListEntry( const char \*strFileName )

{

fpIn = fopen(strFileName, “r”);

}

virtual bool ReadEntry(void)

{

char szBuffer[ 256 ];

fread( szBuffer, sizeof(char),255, fpIn );

if ( feof(fpIn) )

return false;

setFirstName( szBuffer );

fread( szBuffer, sizeof(char),255, fpIn );

setFirstName( szBuffer );

fread( szBuffer, sizeof(char),255, fpIn );

setAddress1( szBuffer );

fread( szBuffer, sizeof(char),255, fpIn );

setAddress2( szBuffer );

fread( szBuffer, sizeof(char),255, fpIn );

setCity( szBuffer );

fread( szBuffer, sizeof(char),255, fpIn );

setState( szBuffer );

fread( szBuffer, sizeof(char),255, fpIn );

setZipCode( szBuffer );

return true;

}

virtual bool First(void)

{

// Move to the beginning of the file, read in the pieces

fseek( fpIn, 0L, SEEK\_SET );

return ReadEntry();

}

virtual bool Next(void)

{

// Just get the next one in the file

return ReadEntry();

}

};

Please note that we do no error-checking in any of this code (that’s to avoid making it any larger). A closer look at this object (before moving on to the last object in the group) shows that this class allocates no storage for the various components of the mailing-list entry — nor will you find any accessor functions to retrieve those components. Yet the class is derived from the base class (which implements all this functionality), so we can utilize the storage defined there. This is a really nice feature; it allows us to encapsulate the data in one place and put the “real” functionality in another. You can also see that we’ve implemented the two required pure virtual functions ( First and Next ) to make the class read the data from a file.

5. Save the source file in your source-code re-editor.

6. Using the code editor, add the code in Listing 2-3 to your source-code file. You may optionally save this code in a separate header file and include that header file in your main program as well.

7.Save the source file in your source-code editor.

L ISTING 2-3: T HE C OMMAND L INE M AILING L IST E NTRY C LASS

class CommandLineMailingListEntry : public BaseMailingListEntry

{

private:

bool GetALine( const char \*prompt, char \*szBuffer )

{

puts(prompt);

gets(szBuffer);

// Remove trailing carriage return

szBuffer[strlen(szBuffer)-1] = 0;

if ( strlen(szBuffer) )

return true;

return false;

}

bool GetAnEntry()

{

char szBuffer[ 80 ];

if ( GetALine( “Enter the last name of the person: “,

szBuffer ) != true )

return false;

setLastName( szBuffer );

GetALine(“Enter the first name of the person: “,

szBuffer );

setFirstName( szBuffer );

GetALine(“Enter the first address line: “, szBuffer );

setAddress1(szBuffer);

GetALine(“Enter the second address line: “, szBuffer );

setAddress2(szBuffer);

GetALine(“Enter the city: “, szBuffer );

setCity(szBuffer);

GetALine(“Enter the state: “, szBuffer);

setState(szBuffer);

GetALine(“Enter the zip code: “, szBuffer );

setZipCode( szBuffer);

return true;

}

public:

CommandLineMailingListEntry() {

}

virtual bool First(void)

{

printf(“Enter the first name for the mailing list:\n”);

return GetAnEntry();

}

virtual bool Next(void)

{

printf(“Enter the next name for the mailing list:\n”);

return GetAnEntry();

}

};

Testing the Mailing-List Application

After you create a class, it is important to create a test driver that not only ensures that your code is correct, but also shows people how to use your code. The following steps show you how:

1.In the code editor of your choice, reopen the source file to hold the code for your test program. In this example, I named the test program ch02.cpp .

2.Type the code from Listing 2-4 into your file, substituting your own names for the italicized constants, variables, and filenames.

L ISTING 2-4: T HE M AILING -L IST T EST P ROGRAM

void ProcessEntries( BaseMailingListEntry \*pEntry )

{

bool not\_done = pEntry->First();

while ( not\_done )

{

// Do something with the entry here.

// Get the next one

not\_done = pEntry->Next();

}

}

int main(int argc, char \*\*argv)

{

int choice = 0;

printf(“Enter 1 to use a file-based mailing list\n”);

printf(“Enter 2 to enter data from the command line\n”);

scanf(“%d”, &choice );

if ( choice == 1 )

{

char szBuffer[ 256 ];

printf(“Enter the file name: “);

gets(szBuffer);

FileMailingListEntry fmle(szBuffer);

ProcessEntries( &fmle );

}

else

if ( choice == 2 )

{

CommandLineMailingListEntry cmle;

ProcessEntries( &cmle );

}

else

printf(“Invalid option\n”);

return 0;

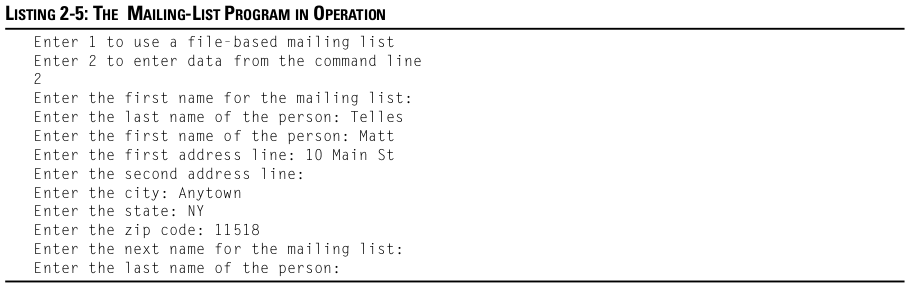
}

The main function for the driver really isn’t very busy — all it’s doing is creating whichever type of object you want to use. The ProcessEntries function is the fascinating one because it is a function that is working on a class type that doesn’t do anything — it has no idea which type of mailing-list entry object it is processing. Rather, it works from a pointer to the base class. If you run this program, you will find that it works as advertised, as you can see in Listing 2-5.

You could likewise create a file containing all entries that we just typed into the various fields above to

enter those fields into the system. You can do all of this without changing a single line of the ProcessEntries function! This is the power of pure virtual functions, and thus the power of abstraction.

L ISTING 2-5: T HE M AILING -L IST P ROGRAM IN O PERATION

When you create a set of classes that are all doing the same general thing, look for the common elements of the class and abstract them into a common base class. Then you can build on that common base in the future, more easily creating new versions of the classes as the need arises.