SESSION

25

Large Programs

Session Checklist

✔ Separating programs into multiple modules

✔ Using the `#include`-directive

✔ Adding files to a Project

✔ Other preprocessor commands

30 Min. To Go

All of the programs to this point have been small enough to contain in a single .cpp source file. This is fine for the examples contained in a book such as *C++ Weekend Crash Course*, but this would be a severe limitation in real-world application programming. This session examines how to divide a program into parts through the clever use of project and include files.

Why Divide Programs?

The programmer can divide a single program into separate files sometimes known as *modules*. These individual source files are compiled separately and then combined during the build process to generate a single program.

Note

The process of combining separately compiled modules into a single executable is called *linking*.

There are a number of reasons to divide programs into more manageable pieces. First, dividing a program into modules results in a higher level of encapsulation. Classes wall off their internal members in order to provide a certain degree of safety. Programs can wall off functions to do the same thing.

Note

Remember that encapsulation was one of the advantages of object-oriented programming.

Second, it is easier to comprehend and, therefore, easier to write and debug a program that consists of a number of well-thought-out modules than a single source file full of all of the classes and functions that the program uses.

Next comes reuse. I used the reuse argument to help sell object-based programming. It is extremely difficult to keep track of a single class reused among multiple programs when a separate copy of the class is kept in each program. It is much better if a single class module is automatically shared among programs.

Finally, there is the argument of time. It doesn't take a compiler such as Visual C++ or GNU C++ very long to build the examples contained in this book using a high-speed computer like yours. Commercial programs sometimes consist of millions of source lines of code. Rebuilding a program of that size can take more than 24 hours. (Almost as long as it's taking you to get through this book!) A programmer would not tolerate rebuilding a program like that for every single change. However, the majority of the time is spent compiling the source file into object files. The link process is much quicker.

Separating Class Definition from Application Program

This section begins with the EarlyBinding example from Session 22 and separates the definition of the class `Student` from the remainder of the application. To avoid confusion, let's call the result `SeparatedClass`.

Dividing the program

We begin by deciding what the logical divisions of SeparatedClass should be. Clearly the application functions `fn()` and `main()` can be separated from the class definition. These functions are not reusable nor do they have anything to do with the definitions of the class `Student`. Similarly, the `Student` class does not refer to the `fn()` or `main()` functions at all.

I store the application portion of the program in a file called SeparatedClass.cpp. So far the program appears as follows:

```
// SeparatedClass - demonstrate an application separated
//                  from the class definition
#include <stdio.h>
#include <iostream.h>

double fn(Student& fs)
{
    // because calcTuition() is declared virtual this
    // call uses the run-time type of fs to resolve
    // the call
    return fs.calcTuition();
}

int main(int nArgc, char* pszArgs[])
{
    // the following expression calls
    // fn() with a Student object
    Student s;
    cout << “The value of s.calcTuition when\n”
         << “called virtually through fn() is “
         << fn(s)
         << “\n\n”;

    // the following expression calls
    // fn() with a GraduateStudent object
    GraduateStudent gs;
    cout << “The value of gs.calcTuition when\n”
         << “called virtually through fn() is “
         << fn(gs)
         << “\n\n”;
    return 0;
}
```

Unfortunately, this module does not compile successfully because nothing in SeparatedClass.cpp defines the class Student. We could, of course, insert the definition of Student back into SeparatedClass.cpp, but doing so defeats our purpose; it puts us back where we started.

The #include directive

20 Min. To Go

What is needed is some method for including the declaration Student in SeparatedClass.cpp programmatically. The #include directive does exactly that. The #include directive includes the contents of the file named in the source code exactly at the point of the #include directive. This is harder to explain than it is to do in practice.

First, I create the file student.h, which contains the definition of the Student and GraduateStudent classes:

```
// Student - define the properties of a Student
class Student
{
  public:
    virtual double calcTuition()
    {
        return 0;
    }

  protected:
    int nID;
};
class GraduateStudent : public Student
{
  public:
    virtual double calcTuition()
    {
        return 1;
    }

  protected:
    int nGradId;
};
```

Note

The target file of the `#include` directive is known as an *include* file. By convention include files carry the name of the base class they contain with a lower case first letter and the extension .h. You may also see C++ include files with extensions such as .hh, .hpp, and .hxx. Theoretically, the C++ compiler doesn't care.

The new version of the application source file SeparatedClass.cpp appears as follows:

```
// SeparatedClass - demonstrates an application separated
//                  from the class definition
#include <stdio.h>
#include <iostream.h>

#include "student.h"

double fn(Student& fs)
{
    // ...identical to earlier version from here down...
```

The `#include` directive was added.

Tip

The `#include` directive must start in column one. The ". . ." portion must be on the same line as the `#include`.

If you were to physically include the contents of student.h in the file SeparatedClass.cpp, you would end up with exactly the same LateBinding.cpp file that we started with. This is exactly what happens during the build process — C++ adds student.h to SeparatedClass.cpp and compiles the result.

Note

The `#include` directive does not have the same syntax as other C++ commands. This is because it is not a C++ directive at all. A preprocessor makes a first pass over your C++ program before the C++ compiler executes. It is this preprocessor which interprets the `#include` directive.

Dividing application code

The SeparatedClass program successfully divided the class definition from the application code, but suppose that this was not enough — suppose that we wanted to separate the function `fn()` from `main()`. I could, of course, use the same approach of creating an include file `fn.h` to include from the main source file.

The include file solution does not address the problem of creating programs that take forever to build. In addition, this solution introduces all sorts of problems concerning which function can call which function based on the order of inclusion. A better solution is to divide the sources into separate compilation units.

During the compilation phase of the build operation, C++ converts the .cpp source code into the equivalent machine instructions. This machine code information is saved in a file called the object file with either the extension .obj (Visual C++) or .o (GNU C++). In a subsequent phase, known as the link phase, the object file is combined with the C++ standard library code to create the executable .exe program.

Let's use this capability to our own advantage. We can separate the SeparatedClass.cpp file into a SeparatedFn.cpp file and a SeparatedMain.cpp file. We begin by creating the two files.

The SeparatedFn.cpp file appears as follows:

```
// SeparatedFn - demonstrates an application separated
//               into two parts - the fn() part
#include <stdio.h>
#include <iostream.h>

#include "student.h"

double fn(Student& fs)
{
    // because calcTuition() is declared virtual this
    // call uses the run-time type of fs to resolve
    // the call
    return fs.calcTuition();
}
```

The remaining SeparatedMain() program might appear as:

```
// SeparatedMain - demonstrates an application separated
//                 into two parts - the main() part
#include <stdio.h>
#include <iostream.h>

#include "student.h"
```

```
int main(int nArgc, char* pszArgs[])
{
    // the following expression calls
    // fn() with a Student object
    Student s;
    cout << "The value of s.calcTuition when\n"
         << "called virtually through fn() is "
         << fn(s)
         << "\n\n";

    // the following expression calls
    // fn() with a GraduateStudent object
    GraduateStudent gs;
    cout << "The value of gs.calcTuition when\n"
         << "called virtually through fn() is "
         << fn(gs)
         << "\n\n";
    return 0;
}
```

Both source files include the same .h files. This is because both files need access to the definition of the `Student` class as well as to the C++ Standard Library functions.

10 Min. To Go

Project file

Full of expectation, I open the SeparatedMain.cpp file in the compiler and click Build.

Tip

If you're trying this at home, make sure that you have closed the SeparatedClass project file.

The error message "undeclared identifier" appears. C++ does not know what a `fn()` is when compiling SeparatedMain.cpp. That makes sense, because the definition of `fn()` is in a different file.

Clearly I need to add a prototype declaration for `fn()` to the SeparatedMain.cpp source file:

```
double fn(Student& fs);
```

The resulting source file passes the compile step, but generates an error during the link step that it could not find the function `fn(Student)` among the .o object files.

Note

I could (and probably should) add a prototype declaration for `main()` to the SeparatedFn.cpp file; however, it isn't necessary because `fn()` does not call `main()`.

What is needed is a way to tell C++ to bind the two source files into the same program. Such a file is called the *project file*.

Creating a project file under Visual C++

There are several ways to create a project file. The techniques differ between the two compilers. In Visual C++, execute these steps:

1. Make sure that you close any project files created during previous attempts to create the program by clicking Close Workspace in the File menu. (A workspace is Microsoft's name for a collection of project files.)
2. Open the SeparatedMain.cpp source file. Click compile.
3. When Visual C++ asks you if would like to create a Project file, click Yes. You now have a project file containing the single source file SeparatedMain.cpp.
4. If not already open, open the Workspace window (click Workspace under View).
5. Switch to File view. Right click on SeparatedMain files, as shown in Figure 25-1. Select Add Files to Project. From the menu, open the SeparatedFn.cpp source file. Both SeparatedMain.cpp and SeparatedFn.cpp should now appear in the list of functions that make up the project.
6. Click Build to successfully build the program. (The first time that both source files are compiled is when performing a Build All.)

Note

I did not say that this was the most elegant way, just the easiest.

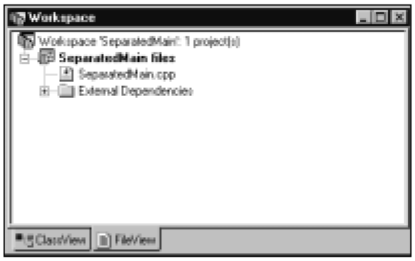


Figure 25-1
Right click the Project in the Workspace Window to add files to the project.

Tip

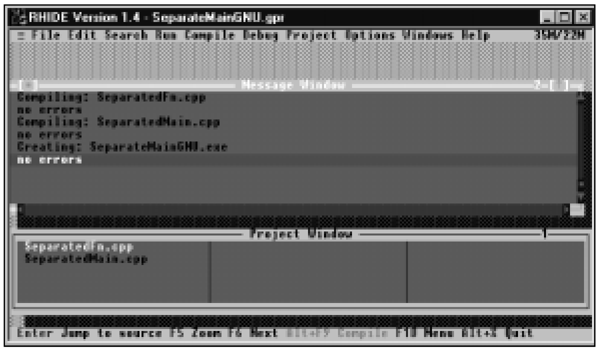
The SeparatedMain project file on the accompanying CD-ROM contains both source files already.

Creating a project file under GNU C++

Use these steps to create a project file under rhide, the GNU C++ environment.

1. Without any files open, click Open Project in the Project menu.
2. Type in the name SeparatedMainGNU.gpr (the name isn't actually important — you can choose any name you want). A project window with the single entry, <empty>, opens along the bottom of the display.
3. Click Add Item under Project. Open the file SeparatedMain.cpp.
4. Repeat for SeparatedFn.cpp.
5. Select Make in the Compile menu to successfully create the program SeparatedMainGNU.exe. (Make rebuilds only those files that have changed; Build All rebuilds all source files whether changed or not.)

Figure 25-2 shows the contents of the Project Window alongside the Message Window displayed during the make process.


RHIDE Version 1.4 - SeparateMainGNU.gpr
≡ File Edit Search Run Compile Debug Project Options Windows Help
Message Window
Project Window
Enter Jump to source F5 Zoom F6 Next Alt+F9 Compile F10 Menu Alt+X Quit

```
Compiling: SeparatedFn.cpp
no errors
Compiling: SeparatedMain.cpp
no errors
Creating: SeparateMainGNU.exe
no errors
```

```
SeparatedFn.cpp
SeparatedMain.cpp
```

Figure 25-2
The rhide environment displays the files compiled and the program linked during the project build process.

Reexamining the standard program template

Now you can see why we have been including the directives `#include <stdio.h>` and `#include <iostream.h>` in our programs. These include files contain the definitions for the functions and classes that we have been using, such as `strcat()` and `cin>`.

The standard C++-defined .h files are included using the <> brackets, whereas locally defined .h files are defined using the quote commands. The only difference between the two is that C++ looks for files contained in quotes starting with the *current* directory (the directory containing the project file), whereas C++ begins the search for bracketed files in the C++ include file directories. Either way, the programmer controls the directories searched via project file settings.

In fact, it is the very concept of separate compilation that makes the include file critical. Both SeparatedFn and SeparatedMain knew of `Student` because student.h was included. We could have typed in this definition in both source files, but this would have been very dangerous. The same definition in two different places enhances the possibility that the two could get out of synch — one could get changed without the other.

Including the definition of `Student` in a single student.h file and including that file in the two modules makes it impossible for the definitions to differ.

Handling outline member functions

The example `Student` and `GraduateStudent` classes defined their functions within the class; however, the member functions should have been declared outside of the class (only the `Student` class is shown — the `GraduateStudent` class is identical).

```
// Student - define the properties of a Student
class Student
{
  public:
    // declare the member function
    virtual double calcTuition();

  protected:
    int nID;
};

// define the code separate from the class
 double Student::calcTuition();
 {
     return 0;
 }
```

A problem arises if the programmer tries to include both the class and the member functions in the same .h file. The function `Student::calcTuition()` becomes a part of both SeparatedMain.o and SeparatedFn.o. When these two files are linked, the C++ linker complains that `calcTuition()` is defined twice.

Note

When the member function is defined within the class, C++ takes special pains to avoid defining the function twice. C++ can't avoid the problem when the member function is defined outside of the class.

External member functions must be defined in their own .cpp source file as in the following Student.cpp:

```
#include "student.h"
// define the code separate from the class
 double Student::calcTuition();
 {
     return 0;
 }
```

Done!

Review

This session demonstrated how the programmer can divide programs into multiple source files. Smaller source files save build time because the programmer need only compile those source modules that have actually changed.

- Separately compiled modules increase the encapsulation of packages of similar functions. As you have already seen, separate, encapsulated packages are easier to write and debug. The standard C++ library is one such encapsulated package.
- The build process actually consists of two phases. During the first, the compile phase, the C++ source statements are converted into a machine readable, but incomplete object files. During the final, link phase, these object files are combined into a single executable.
- Declarations, including class declarations, must be compiled along with each C++ source file that uses the function or class declared. The easiest way to accomplish this is to place related declarations in a single .h file, which is then included in source .cpp files using the `#include` directive.
- The project file lists the modules that make up a single program. The project file also contains certain program-specific settings which affect the way the C++ environment builds the program.

Quiz Yourself

1. What is the act of converting a C++ source file into a machine-readable object file called? (See “Why Divide Programs?”)
2. What is the act of combining these object files into a single executable called? (See “Why Divide Programs?”)
3. What is the project file used for? (See “Project File.”)
4. What is the primary purpose of the `#include` directive? (See “The #include Directive.”)