Chapter 11: Modules and Makefiles

11.1 Modular Organization and makefiles.

From the Fanny Farmer cookbook:

Before beginning to mix, be sure that all ingredients and utensils are on hand.

You can't bake a good cake if you don't have enough eggs or if your butter is spoiled. You can't compile or run a program successfully if you need files that are missing or out of date.

11.1.1 History

The C++ standard does not address the issues of code files and header files. These and makefiles developed as part of the Unix system environment because of a clear and evident need to mechanize and automate the production of large-scale applications.

Projects. Both Unix-based and Windows-based systems have supported *projects* as part of an integrated development environment (IDE). The project facilities vary greatly in both power and convenience of use, and can be more difficult for a newcomer to use than the old-fashioned makefile. At this time, the Microsoft project facility is both confusing and intolerant of error. Mac's XCode is flexible and less intolerant of error, but highly complex. Both organize the project using a special project file. The cross-platform Eclipse IDE gives the programmer a choice of producing a traditional makefile or a non-portable project file.

A modern IDE offers much more than just a project facility. Most useful are syntax coloring, global renaming and refactoring, and global search-and-replace. It is clear that, as IDEs improve and our experience with them increases, they will gradually replace the old ways of doing the job: compiling from the command line and hand-built makefiles.

Whether you are using a project or a makefile, one kind of information must be supplied: the full pathname of every source code file and every header file that is needed by the project. Given this information, a project facility and some makefile facilities will generate a list of which modules depend on which other modules. This dependency list is used to determine which modules are unaffected by a change to part of the program. When you ask to build the application, the only modules that will be recompiled are those that have been affected by some editing change since the last build. Thus, if one module of a massive application is changed, we avoid recompiling almost all of the program. Re-linking will happen if anything was recompiled.

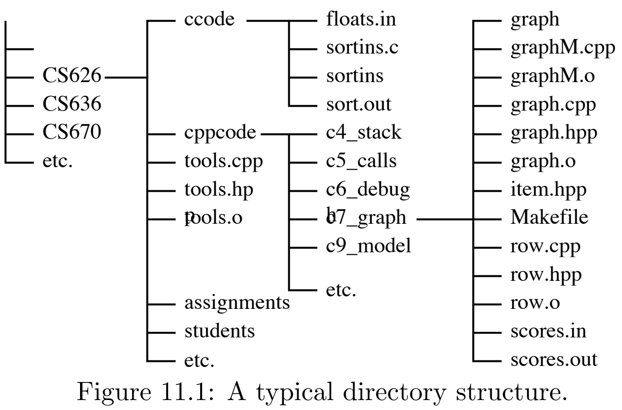

Figure 11.1: A typical directory structure.

Namespaces. Very recently, the C++ standard adopted the concepts of namespaces, and Visual C++ implemented them in conjunction with workspaces. These facilities may or may not replace the traditional makefiles and separate compilation. At the moment the added facility leads more to confusion than to ease of use. The purpose of namespaces is the same as one of the purposees of separate compilation: to allow many modules to define objects and functions freely without concern about conflict with other objects of the same name in other modules.

File and Directory Organization. A multi-module program in C++ consists of pairs of .cpp and .hpp files, one pair for each class (or closely related set of classes), and one .cpp file that contains the main program. (Exceptions: he main program may or may not have a matching .hpp file, and some simple classes are written entirely within the class .hpp file.) Each .cpp file includes just its own .hpp file. Each .hpp file in the project must include the other .hpp files that contain the prototypes it uses.

Experienced programmers define a separate directory for each project. (This saves much pain in the long run.) This directory contains the .cpp, .hpp, and .o files for the project as well as the makefile, the executable file, the input, and the output. Since many projects may use the tools files, they are placed a level or two above the project directories, as indicated by Figure 11.1.

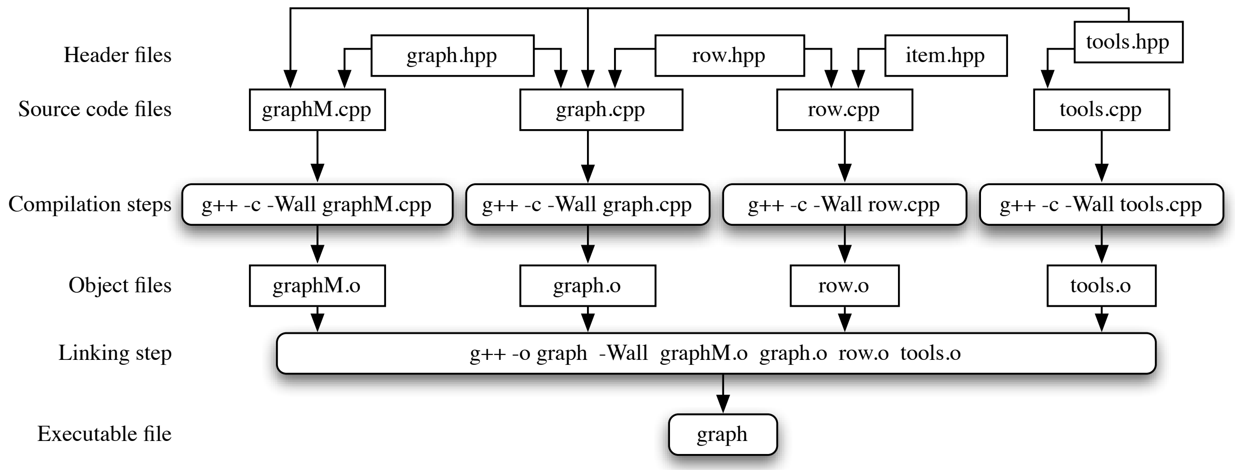

Figure 11.2: The BarGraph application with four code modules.

11.1.2 General Concepts

- A program that is physically located in many files will be compiled as a single module if the main program includes (directly or indirectly) the .cpp files for the other modules. In a multi-module compilation, main() includes the header files of the other modules, *not* their source code.
- When doing a multi-module compile, each module is compiled separately to make a .o file, as shown in Figure 11.2. Then the linker integrates these .o files with the .o files for the libraries to make an executable file.
- A makefile (in Unix) automates the process of separate compilation and linking. The project files in Windows systems accomplish the same goals as a makefile, but a makefile has somewhat more power and flexibility because it can be hand-coded and/or hand-modified. Project files are automatically generated and, therefore, rigid. Makefiles are universally portable in the Unix world.
- One .hpp file may be included by several others, so we must do something to prevent definitions from being included twice in the same module. (This would cause the compiler to fail.) Many systems support this directive at the top of a header file:

  ```
  #pragma once;
  ```

  This is advice to the compiler to include the header only once per module. If your system does not support this, you must surround the entire .hpp file with conditional compilation commands.

```
#ifndef MYCLASS_H
#define MYCLASS_H
... put the class declaration here ...
#endif
```

If an attempt is made to include the same header twice in one module, this ”wrapper” will cause the inclusion to be skipped the second time, avoiding conflicts due to duplicate definitions. The ”wrapper” always follows the pattern shown above, where MYCLASS is traditionally the same as the class name, but written in upper-case (instead of mixed case) letters. Anything that might be the same as a standard library header file name should be avoided.

- Naming. Three names are associated with each module. These could be completely unrelated, but it is better to make them alike. One suggestion is to use the name of the major class within the module, but write that name in three ways:
  - The name of the pair of the .hpp and .cpp files. *Use all lower case* here because it is the most portable from file system to file system.
  - The name of the class within the file. Use *Mixed Case* here because that seems to be becoming customary for class names: List, Cell, BarGraph
  - If your system does not recognize `#pragma once;`, you need to put conditional compilation directives at the top and bottom of each header file. Use *ALL UPPER CASE* letters for the `#define` symbol at the top of the .hpp file, since that is customary for defined symbols. It is also customary to end each one with `_H`. Example: `LIST_H`.
- The linker can fail for three reasons:
  - Some required symbol (usually a function name) is not in any of the modules that are being linked, usually caused by a missing module.
  - Some symbol is defined in two of the modules (double inclusion).
  - Lack of disk space to store the executable file.
- The linker will not succeed if same object is defined *outside the class declarations* in two .hpp files, or in one .hpp file that is included by two others. This causes double inclusion. The `#pragma once;` and `#ifndef` directives cannot help with this problem because they operate at the level of a single module. The linking problem occurs when one method or object is part of the .o file for two modules.

  However, if a function is defined entirely *inside* a class declaration (in an .hpp file), or defined as an inline function *after* the class (in an .hpp file), the .hpp file may be included by several other modules without causing a conflict.

  If a data object (such as a constant array of strings) must be directly visible to two modules, you must define it in one module and use `extern` declarations to link it to the other modules.

11.1.3 Enumerations and External Objects

The problem. An enum declaration creates a type, and types are usually defined globally. Several parts of a program typically need to use the enumerated type, so it cannot be conveniently put inside a single class. The best place is a header file, enums.hpp, that contains the enum declarations for the whole application.

Further, to use enum types effectively, you need a civilized way to input an enum value and to output it. Numeric codes are not civilized. Input can be easily handled by reading a menu selection and processing it with a switch. Output is a harder problem because we must go from the enum constant to an English word that the user will understand. We can do this by creating an array of output strings, parallel to your enum list, and using the enum value to subscript the array when we want output.

However, this array is an object, so it cannot go into the .hpp file with the corresponding enum declaration. If that .hpp file were included in more than one compile module, the array object would be compiled more than once and be part of two .o files. The linker would then be unable to link the application because it would see two objects with the same name.

The Solution The solution to this linking problem is to use the `extern` storage class. Identical extern declarations are included by several files, but this does not cause a linking problem because an extern declaration *with no initializer* does not create an object. (It instructs the linker to look for and link to an object that was created elsewhere.) An extern declaration *with an initializer* creates the object that all the other modules will link to. Therefore, we write one copy of an extern declaration *with an initializer* in some .cpp file. For this purpose, you might use the file that contains main(), a .cpp file that contains the application's primary class, or a separate file called enums.cpp.

Details of the Solution

1. Create a file named enums.hpp.
   (a) Into it, put all the enum declarations needed by your application. Suppose you need an enumeration of colors: red, white, blue. Sometimes you need an additional code for errors:
   ```
   enum Color { ERROR, RED, WHITE, BLUE };
   ```
   (b) For each enum declaration, you need an array of strings for making civilized output. Below each enum declaration, put a declaration similar to this just :
   ```
   extern const char* colorNames[4];
   ```
   (c) `#include enums.hpp` in the .hpp file for every class that needs to use the colors or the column status. For output, simply use an enum constant to subscript this array of words.
   ```
   Color itemColor = RED;
   cout << color ordered << " \t" << colorNames[itemColor];
   ```
2. In enums.cpp or some other .cpp file, put a matching extern declaration for
   ```
   extern const char* colorNames[4] = {"Error","Red","White","Blue" };
   ```

11.1.4 Rules

It should be clear by now that the physical division of source code into .hpp files and .cpp files is not easily compatible with the logical division of code into classes. Thus, the traditional compilation and linking methods that worked well for C are not a good fit for C++. These rules need to be understood when trying to debug the global structure of a project file or makefile:

1. Every function, variable, or class must be declared before it can be referenced.
2. If two definitions need to reference each other, the cycle must be broken by declaring one, declaring and defining the second in terms of it, then giving the full definition of the first one.
3. When given in different places, a definition must exactly match its declaration.
4. Every function (and external variable) that is used must be defined exactly once in the completed, linked, project.

11.2 A Makefile defines the project.

In a Unix environment, a makefile automates and speeds up the process of compiling and linking the application and makes it much easier to keep everything consistent.

- It allows you to list the object modules and libraries that comprise the application and say how to produce and use each one.
- For each module, it allows you to specify the code and header files on which it depends.
- When you give the command to make or build the project, the facility generates the executable program. Initially, this means it must compile every module and link them together.
- When you say make, control goes immediately back to you if nothing has changed since the last linking.
- When you change a source-code module, the corresponding object module becomes obsolete. The next time you make the application, it will be recompiled.

- When you change a header file, the object files for all modules that depend on it become obsolete. The next time you make the application, they will be recompiled.
- When you recompile a module, the corresponding executable program becomes obsolete, so the application will be relinked.
- It allows you to easily specify a set of flags that control how the compiler will work. Some useful flags are described later in this handout.
- It lets you automate the process of deleting files that are no longer needed.

Makefile syntax.

- Any line that starts with a `#` is a comment. (Lines 1, 6, 9, 18, 22.)
- To define a macro, write a name followed by an = sign and the definition. We define symbols this way on lines 7 and 10. To use a macro, enclose its name in `$(...)`. We use the defined symbols in lines 12, 13. 16, and 20.
- Object and executable files are built from source and header files and, therefore, depend on them. To define a dependency, list the name of a target file followed by a colon and the list of source files that are used to create it. Lines 12, 19, 23, 24, and 25 define dependencies. If any file on the right changes, the file on the left will be regenerated the next time "make" is executed.
- Any line that is indented MUST start with a single tab character; it won't work if the tab is missing or if it is replaced by spaces.

11.2.1 Making the Bargraph Application

The makefile.

```
1   # Rule for building a .o file from a .cpp source file
2   .SUFFIXES: .cpp
3   .cpp.o:
4       g++ -c $(CXXFLAGS) $<
5
6   # Compile with debug option and all warnings on.
7   CXXFLAGS = -g -Wall -I../..
8
9   # Object modules comprising this application
10  OBJ =   graphM.o graph.o row.o ../../tools.o
11
12  graph: $(OBJ)
13      g++  $(CXXFLAGS) -o graph $(OBJ)
14
15  clean:
16      rm -f $(OBJ) graph
17
18  # Use tools source file from grandparent directory --------
19  ../../tools.o:   ../../tools.cpp ../../tools.hpp
20      g++ -c $(CXXFLAGS) ../../tools.cpp -o ../../tools.o
21
22  # Dependencies
23  graph:      graphM.cpp graph.hpp row.hpp item.hpp ../../tools.hpp
24  graph.o:    graph.cpp graph.hpp row.hpp
25  row.o:      row.cpp row.hpp item.hpp
```

- Please note that the line numbers are part of this explanation, they do *not* belong in the makefile.
- Lines 3 and 4 are a template, or general rule, for creating a compilation command. Line 3 says that this rule tells how to convert a .cpp file to a .o file. The command is on line 4. It refers to two macro symbols, `CXXFLAGS` (defined on line 7) and `$<` which stands for the name of a .cpp file that will be supplied later. This template will be used automatically generate a compilation command every time a .o file is needed and there is no specific rule for creating it.

- Line 7 defines the symbol CXXFLAGS, a listof the options, or flags, to be used by the C++ compiler and linker. I sometimes use these flags:
  - `-Wall` to turn on all warnings and `-Wno-char-subscripts` to turn off the single warning about using characters as subscripts.
  - Since I keep my tools one or two levels higher in my directory tree, I use the `-I..` or `-I../..` flag to define the search path for files included from this other directory.
  - If you want to use the on-line debugger, you must also give a `-g` flag.
- Line 10 defines the symbol OBJ to be the list of object files that compose this application. We define the list once, and use it three times, in lines 12, 13, and 16. The defined symbol makes it easy to write three commands that work with the same list of object files and makes it easy to modify that list when a module is added or removed.
- This makefile defines two commands (lines 12. . . 16) that can be used from the command line:

`make graph`	Lines 12 and 13	Compile and link the graph application.
`make`		This is the same as make graph.
`make clean`	Lines 15 and 16	Clean out any existing .o files and, finally, delete the executable file.

- Line 13 is the linking command that will be used.
- Line 16 is a command that will delete all the generated files. We use it to clean up a directory after finishing a project or to force a fresh compilation of all modules.
- Lines 19 and 20 define the dependency and the compile command for the tools file. This rule is given separately because it is different from the general rule on line 4.
- Line 20 is the command that will be used to compile the tools file in the grandparent directory. If the .o file already exists, this step will be skipped. After the first project is compiled once, tools.o will be in the grandparent directory where the linker will find it for other projects.
- If the tools were only one directory level up, instead of two, the following lines would be changed:

```
7.  CXXFLAGS = -g -Wall -I..
10. OBJ =   graphM.o graph.o row.o ../tools.o
18. # Use tools source file from parent directory ----------
19. ../tools.o:   ../tools.cpp ../tools.hpp
20.     g++ -c $(CXXFLAGS) ../tools.cpp -o ../tools.o
23. graph:      graphM.cpp graph.hpp row.hpp item.hpp ../tools.hpp
```

- This file should be named `Makefile` or `makefile` and should be in the same directory as the majority of the source code files.