computer science II

c m s c 214
spring 2002

Makefiles

Background

First of all, anyone who does not know how to write a makefile must have missed the first discussion section (in the second week of class), because there was a handout on makefiles and tar.

Note: this was written for the g++ compiler. You should make appropriate changes so your code compiles with the cxx compiler.

Separate Compilation

One of the features of C and C++ that's considered a strength is the idea of "separate compilation". Instead of writing all the code in one file, and compiling that one file, C/C++ allows you to write many .cpp files and compile them separately.

With few exceptions, most **.cpp** files have a corresponding **.h** file. A **.cpp** usually consists of:

- the implementations of all methods in a class,
- standalone functions (functions that aren't part of any class),
- and global variables (usually avoided).

The corresponding **.h** file contains

- class declarations,
- function prototypes,
- and extern variables (again, for global variables).

The purpose of the **.h** files is to export "services" to other .cpp files.

For example, suppose you wrote a **Vector** class. You would have a **.h** file which included the class declaration. Suppose you needed a vector in a **MovieTheater** class. Then, you would **#include "Vector.h.**

Compiling a .cpp file

What does the compiler do when it sees **#include .h** file in a **.** file? For example, suppose you are compiling a MovieTheater class, which contains the line **#include "Vector.h"**.

The preprocessor will "insert" the **Vector.h** file into a *copy* of the **MovieTheater.cpp** source code. After the preprocessing phase has completed, the modified copy is sent to the compiler.

What information does the compiler have about **Vector** when compiling **MovieTheater.cpp**? Itn only knows the data members of **Vector.h** and the methods of **Vector.h**. It does NOT know how the methods are implemented, since this information is not stored in the **.h** file.

Fortunately, that's all it needs to know to create an object file. To create an object, you need to know how big it is, and what the data members are. You get that information from **Vector.h**.

The compiler also need to know what are valid methods of **Vector**, what the parameters are and the return types. Again, this information is also in **Vector.h**. This is necessary to make sure that methods called on **Vector** objects are called in a valid manner.

Surprisingly enough, the compiler doesn't need to know how any of **Vector**'s methods are implemented. When compiling **MovieTheatern.cpp**, the compiler never looks at **Vector.cpp**.

However, eventually, if the **MovieTheater** class is to run, there must be a time when the object code (i.e. the **.o** file) for **Vector** is integrated with the object code for **MovieTheater** (and whatever object file contains **main()**). That process occurs later on, during the *linking phase*. We'll talk about linking momentarily.

When you compile **MovieTheater.cpp**, you only create a **MovieTheater.o** file. This is called an *object file*, and is also called a **.o** file. The object file doesn't have enough information to run by itself. It is not an executable.

Typically, when you have many **.cpp** files, you will end up creating a corresponding **.o** file. In g++ and cxx (and most UNIX C++ compilers), this is done with the **-c** option.

```
g++ -Wall -c MovieTheater.cpp
cxx -w0 -std strict_ansi -c MovieTheater.cpp
```

The **-c** option can appear anywhere in the command.

When the compiler sees this option, it assumes you will create a **.o** file from the **.cpp** file. When compiling a **.cpp** file, the compiler does not require a **main()** function to appear in the file, although you can have a **main()**. Also, any implementations of classes that are NOT part of the **.cpp** file, but appear in the **#include** files are not compiled. For example, if **Movie.cpp** includes **Vector.h**, the code for **Vector.cpp** is not compiled when **Movie.cpp** is being compiled.

Instead, the compiler merely checks that the **Vector** methods called correctly. It can check that the calls are correct, even without the implementation. All this information that is needed by the compiler appears in the **.h** files.

Common Misconceptions

There are some common misconceptions.

> **"When you compile a .cpp file using -c option, it compiles all other .cpp files mentioned in the #include .h files"**

Untrue. It only compiles the single **.cpp** file being referred to, and only refers to the **.h** for purposes of creating objects (making sure it's the correct size, etc) and checking that methods are being called correctly.

Even if you remove the **-c** option, it does not look for all other **.cpp** files. In fact, if you remove the option, it assumes that the **.cpp** file is a stand alone file, with everything defined, and includes a **main()**.

> **"If you write something like:**
>
> ```
> g++ -Wall foo.cpp bar.cpp baz.cpp
> ```
>
> **the compiler will look at "bar.cpp" and "baz.cpp" when compiling "foo.cpp".**

That's not quite true. The compiler compiles each .cpp file separately. If you don't use the **-c** option, then effectively, it creates internal versions of .o files (they aren't made into real files), and then linked.

However, this process is basically the same as if you had compiled each **.cpp** file individually and then linked together at the end.

Now, makefiles

Why all the talk about how .cpp files get compiled in C++? Because of the way C++ compiles files, makefiles can take advantage of the fact that when you have many .cpp files, it's not necessary to recompile all the files when you make changes. You only need to recompile a small subset of the files.

Back in the old days, a makefile was very convenient. Compiling was slow, and therefore, having to avoid recompiling every single file meant saving a lot of time.

Although it's much faster to compile now, it's still not very fast. If you begin to work on projects with hundreds of files, where recompiling the entire code can take many hours, you will still want a makefile to avoid having to recompile everything.

A makefile is based on a very simple concept. A makefile typically consists of many entries. Each entry has:

- a target (usually a file)
- the dependencies (files which the target depends on)
- and commands to run, based on the target and dependencies.

Let's look at a simple example.

```
Movie.o: Movie.cpp Movie.h Vector.h
   g++ -Wall -c Movie.cpp
```

The basic syntax of an entry looks like:

```
<target>: [ <dependency > ]*
   [ <TAB> <command> <endl> ]+
```

A makefile typically consists of many entries. A target appears before the left of a colon. A target is *usually* a file, but not always. The entry tells you how to construct a target, and more importantly, *when* to construct the target.

After the target and the colon, you list out the dependencies. The dependencies are (usually) a list of files which the target file depends on. The dependencies are used by the makefile to detemine *when* when the target needs to be reconstructed (usually by recompiling code). The *make* utility decides when to reconstruct the target based on something very simple: a *timestamp*.

Each file has a timestamp that indicates when the file was last modified. *make* looks at the timestamp of the file, and then the timestamp of the dependencies (which are also files). The idea is this. If the dependencies have changed, then perhaps the target needs to be updated. More precisely, if the dependent files have a more recent timestamp than the target, then the command lines are run. If written correctly, the commands should update the target.

The updates are done recursively. When checking the dependencies, the makefiles checks if any of the dependent files are targets, and if so, it looks at those dependencies, and, based on the timestamp, it may run commands to update the dependent files, and then update the target.

Making .o files

The most common target you will make is a .o file. This is created when you compile a .cpp file using the **-c**.

Here's an example:

```
Movie.o: Movie.cpp Movie.h Vector.h
   g++ -Wall -c Movie.cpp
```

The target is **Movie.o**. The dependencies of a **.o** file are almost always:

- The corresponding .cpp file.
- The corresponding .h file.
- Other .h files

It's a common mistake to assume that other .cpp files are ever dependencies. They shouldn't be. Why?

If you recall, when you compile a **.cpp** file to an object file, the compiler never needs to refer to other .cpp files. That's the benefit of separate compilation. You only need to consider the **.cpp** file and other **.h**. That way, if the **.h** files that a .cpp depend on don't change, but the corresponding .cpp files do, there's no need to recompile the original .cpp file.

Let's illustrate with an example. Suppose **Movie.cpp** depends on **Vector.h**. Perhaps it contains a **Vector** object in the class. Now, if **Vector.cpp** changes but **Vector.h** does not, then there's no need to recompile **Movie.cpp**. **Movie.cpp** doesn't need to see the code for **Vector.cpp**. That's handled later at the linking phase.

Deciding on dependencies

OK, so it makes sense to have **Movie.o** depend on **Movie.cpp** and **Movie.h**. After all, if either change, then **Movie.o** needs to be recompiled (at least, usually). But what should you put for the other .h dependencies?

That should be easy. Look in the corresponding **.h** and . files and look for any #include files in double quotes (the ones in angle brackets ought to be files like "iostream", etc., which basically never change), and put those down. Many people forget to put these dependencies down, and therefore, they don't recompile the object file when they should.

However, it's a little more complicated than that. Suppose the **Vector.h** depends on other .h files. You ought to add the dependencies there too. Why? Suppose **Vector** has a Foo object as a data member. Now, **Movie** might include a **Vector** object as a data member. Even if **Vector.h** doesn't change, **Foo** might change (let's say a new data member was added). This makes **Foo** bigger (since it has a new data member), which makes **Vector** bigger, which makes **Movie** bigger. Thus, one needs to recompile **Movie** to reflect that change.

It can be tough to track all the .h dependencies, but you can write the dependencies in the makefile.

```
Vector.h: Foo.h
```

In this case, you don't need any targets. It just allows the compiler to determine that a dependency exists (dependencies are transitive, that is, if X is dependent on Y, and Y is dependent on Z, then X is dependent on Z).

Who Determines the Dependencies?

You do. Yes, it would be nice if a program could determine the dependencies for you. After all, doesn't it simply require that you look at the **.h** files and then do this recursively?

As it turns out, the answer is that this can be done automatically (i.e., by a program). That program is called **makedepend**. However, we won't discuss **makedepend** in this tutorial.

Commands

Once *make* determines that there is at least one dependency with a more recent timestamp than the target, it will run the commands that appear just after the *dependency line* (the line containing the target and the target's dependencies). There can be more than one command after the dependency line, but all commands must immediately follow the dependency line, and the first character of each commandn line must be a TAB.

This is, alas, one of the more common errors in making a makefile. The problem, unfortunately, lies not with the user of the makefile, but the person who originally programmed it. That person must have thought that it would be convenient to have each command start with a TAB, because the command would be indented in.

However, the person failed to realize that several spaces look like a TAB, and even a few spaces followed by a TAB, look like a TAB. It's just too hard to see any mistakes, and the person should never have depended on TABs.

Alas, like many UNIX utilities, the original version has managed to hang around forever, despite its flaws. The one advantage of Microsoft programs is that they are always being updated, and that it's hard to find programs that are nearly the same as the original. On the other hand, tools like **make** and **tar** with obvious ways to make it better still exists in pretty much the old format.

As a user, you must be careful with TABs.

Ah, back to business. The commands are *supposed* to generate a new version of the target. For example,

```
Movie.o: Movie.cpp Movie.h Vector.h
   g++ -Wall -c Movie.cpp
```

The g++ command does indeed create a **Movie.o** when done. Keep that in mind. The commands you list should generate the target. You don't want to have a command like:

```
Movie.o: Movie.cpp Movie.h Vector.h  g++ -Wall -c Vector.cpp
```

This command does not produce **Movie.o**. This isn't a good command.

Exectuables as targets

Usually, the purpose of the makefile is to create an executable. That target is often the first target in the makefile. For example,

```
p1 : MovieList.o Movie.o NameList.o Name.o Iterator.o  g++ -Wall MoveList.o Movie.o NameList.o Name.o Iterator.o -o p1
```

If **p1** is the first target in your makefile, then when you type "make", it will run the commands for **p1**. "make" always runs the commands from the *first* target in the makefile. It won't run any of the commands from other targets.

Notice that the command to compile will create an executable called *p1*. It uses the **-o** option to create an executable with a name other than **a.out**.

What to name the makefile

The "make" utility uses the following rules to determine which makefile to run.

1. If you type **make** and there's a file called **makefile**, then it will run the commands from the first target of that file, provided the dependent files are more recent than the target.
2. If you type **make** and there's a file called **Makefile**, but no file called **makefile**, then it will run the commands from the first target of that file, provided the dependent files are more recent than the target.
3. If you type **make -f <filename>** then it will run the commands from the first target of **<filename>**, provided the dependent files are more recent than the target.

You will be told in class or in a webpage what you should name your file.

Running make on the command line

There are several ways to run **make**.

1. Type **make** and hit return. It will look for a makefile in the current directory and run the commands of the first target, assuming that it has to based on the dependencies.
2. Type **make -f <filename>** and hit return. It will look for a makefile with the name <filename> in the current directory and run the commands of the first target, assuming that it has to based on the dependencies.
3. Type **make <target>**. It will again look for a file called **makefile** (and if that's not there, it looks for **Makefile**) and locate the target. This does not have to be the first target. It will run the commands provided the dependencies are more recent than the target.

Macro definitions for flexibility

You should use macros. You SHOULD use macros! YOU SHOULD USE MACROS!!!

Use MACROS!!!! (We mean it).

Macros allow you to define "variables" which are substituted in. Here's an example of how to use them:

```
OBJS = MovieList.o Movie.o NameList.o Name.o Iterator.o CC = g++ DEBUG = -g CFLAGS = -Wall -c $(DEBUG) LFLAGS = -Wall $(DEBUG) p1 : $(OB
```

Macros are usually put at the beginning of a makefile.

A macro has the following syntax

```
<macro_name> = <macro_string>
```

On the left hand side of the equal sign is the *macro name*. Basically, you should follow the convention of only using upper case letters and underscores.

On the right hand side is some string, which will be substituted. A macro is very similar to a **#define** in C/C++. It allows for substitution. It looks throughout the file for anything starting with a dollar sign and surrounded by either parentheses or braces.

In particular, it looks for things taht look like $(CC) or ${CC}--- both versions are acceptable, although I will only use the one with parentheses.

If the macro string is too long for one line, you can put a backslash, followed immediately by a return (newline). "make" will assume that the next line is a continuation of the line that has the backslash followed by the newline.

In fact, you can always use the backslash followed by a newline to indicate a continuing line. It doesn't just appear in macros. It can also apply to dependencies, commands, etc.

Why use macos?

You want to use macros to make it easy to make changes. For example, if you use macros, it's easy to change the compiler and compiler options between g++ and cxx. It's easy to turn on and of debug options. Without macros, you would use a lot of search and replace.

You use macros in makefiles for the same reason you define constants in programs. It's easier to update the files and make it more flexible.

Common macros for C++ programming

The example in the previous section shows common examples of macros.

- **CC** The name of the compiler
- **DEBUG** The debugging flag. This is **-g** in both g++ and cxx. The purpose of the flag is to include debugging information into the executable, so that you can use utilities such as **gdb** to debug the code.
- **LFLAGS** The flags used in linking. As it turns out, you don't need any special flags for linking. The option listed is "-Wall" which tells the compiler to print all warnings. But that's fine. We can use that.
- **CFLAGS** The flags used in compiling and creating object files. This includes both "-Wall" and "-c". The "-c" option is needed to create object files, i.e. **.o** files.

You will notice that once a macro is defined, it can be used to define subsequent macros.

Dummy targets

There are times when you don't want to create a target. Instead, you want to run a few commands. You can do so with dummy targets. A dummy target is a target which is not a file. However, the **make** utility doesn't know it's not a file, and will run commands as if it was a file that had to be created.

There are three popular dummy targets used in makefiles.

make clean

Occasionally, you will want to remove a directory of all .o files and executables (and possibly emacs backup files). The reason? If you have written your makefile incorrectly, it's possible that there is a .o file that is not being updated when it should.

By removing all such files, you force the makefile to recompile all .o files, thus guaranteeing the most recent rebuild. Of course, this is not something that should be necessary if you've created a makefile correctly. It defeats the purpose of creating a makefile in the first place!

Nevertheless, if you find that the code isn't working the way it should, and you suspect it's due to a buggy makefile, you can run **make clean.** This is how it usually looks:

```
clean:  \rm *.o *~ p1
```

The backslash prevents "rm" from complaining if you remove the files (especially, if it's been aliased with the **-i**). Normally, you remove all .o files, all files ending in **~** (which are emacs backup files), and the name of the executable (or executables, as the case may be). In the example above, the executable is called p1.

Notice that "clean" does not cause any files to be created. In particular, **clean** is not created. There are also no dependencies. Basically, make notices that there is no such file as **clean** and begins to run the commands in the command lines to create the file (which won't be created).

The effect is that running **make clean** causes the commands to always run. That's fine, because it's used to remove files, we don't want. Usually, you run **make** after running **make clean** to rebuild the executable.

make tar

When you submit files, you will often have to tar many files together. It's easy to forget to tar all files, and it's easy to make mistakes on the tar syntax. By adding a dummy target that creates a tar file, you can make a tarfile without errors (presuming you "debug" it long before you need to submit).

Here's an example:

```
tar:  tar cfv p1.tar Movie.h Movie.cpp Name.h Name.cpp NameList.h \  NameList.cpp Iterator.cpp Iterator.h
```

Again, there are no dependencies. However, this time a file does get created (**p1.tar**). You could make **p1.tar** the target if you want, but this way, it always recreates **p1.tar** since **tar** should not be a file that exists in your directory.

make all

Finally, there are a few occasions when you may need to create more than one executable or do more than one task. You can do this with an all target. For example, suppose you want to create three executables: p1, p2, and p3.

You can write

```
all: p1 p2 p3 p1: Foo.o main1.o  g++ -Wall Foo.o main1.o -o p1 p2: Bar.o main2.o  g++ -Wall Bar.o main2.o -o p2 p3: Baz.o main3.o  g++ -\
```

Notice that the **all** target has no command lines. However, it lists the dependencies as p1, p2, and p3. It attempts to build the most up-to-date versions of those files (if they are up-to-date, it will let you know). Having an all target is convenient for making a bunch of executables, without having to type **make** on each target separately.

Common errors in makefiles

Perhaps the most common error in writing a makefile is **failing to put a TAB at the beginning of commands.** The second most common error is to put a TAB at the beginning of blank lines. The first error causes the commands not to run. The second causes the "make" utility to complain that there is a "blank" command.

Unfortuantely, it's hard to see TABs. You need to go into the editor, and go to the beginning of command lines, and move the cursor forward. If it jumps several spaces, you know there is a TAB at the beginning. If it doesn't, you know there isn't.

If you have a command that continues on the next line with a backslash (followed immediately by a newline), then the continuation line does not need to start with a TAB (it makes sense, after all, it is treated as a continuation, not as a new command).

Another common error is not hitting return just after the backslash, should you choose to use it.

Of course, another error is not getting the dependencies correct, but that's not exactly a makefile error, per se.

Putting it all together

Here's a sample makefile for creating an executable called **p1.**

```
OBJS = MovieList.o Movie.o NameList.o Name.o Iterator.o
CC = g++
DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)

p1 : $(OBJS)
    $(CC) $(LFLAGS) $(OBJS) -o p1

MovieList.o : MovieList.h MovieList.cpp Movie.h NameList.h Name.h Iterator.h
    $(CC) $(CFLAGS) MovieList.cpp

Movie.o : Movie.h Movie.cpp NameList.h Name.h
    $(CC) $(CFLAGS) Movie.cpp
```

```
NameList.o : NameList.h NameList.cpp Name.h
    $(CC) $(CFLAGS) NameList.cpp

Name.o : Name.h Name.cpp
    $(CC) $(CFLAGS) Name.cpp

Iterator.o : Iterator.h Iterator.cpp MovieList.h
    $(CC) $(CFLAGS) Iterator.cpp

clean:
    \rm *.o *~ p1

tar:
    tar cfv p1.tar Movie.h Movie.cpp Name.h Name.cpp NameList.h \
            NameList.cpp  Iterator.cpp Iterator.h
```