University of Illinois at Urbana-Champaign
Department of Computer Science

Tutorial : Makefiles

CS 225 Data Structures and Software Principles

1 Introduction

In this tutorial, you will run through quick a tutorials of the UNIX `make` program, which aids in organizing compilation efforts. Don't be alarmed by the size of this handout – there's a lot of writing, but it's only a detailed descriptions of the `Makefile` we give you and what you have to do. The actual work is minimal.

2 Given code

Before you start this tutorial, you want to copy the tutorial code into your own directory. Do this by first changing to the directory into which you want to copy the tutorial code, and then running the command:

```
cp -R ~cs225/src/tutorial .
```

In case you haven't seen it before, the -R stands for "recursive". It means that the directory "tutorial" and *everything* in it – including the full contents of any subdirectory – will be copied to a new directory called "tutorial" that is a subdirectory of wherever you are now. Don't forget the period at the end of the line!!! The period signifies your "present working directory", i.e. where you are now; putting that period there tells the `cp` command that the copied `tutorial` directory should be placed inside the directory in which you are currently located.

2.1 `make` and the `Makefile`

NOTE: In this course, we will be using Sun's compiler, `CC`, which is obviously going to be a C++ compiler, given that we are using C++ in this course. However, the idea of a `Makefile` works just as well no matter which compiler you choose to use.

2.1.1 The purpose of the `Makefile`

If you have never used a `Makefile` before, then your experience compiling from the command line at most consists of repeatedly typing in lines like the following:

```
CC -c file1.C
CC -c file2.C
    .
    .
    .
```

and assorted other command-line manipulations of individual files. Now, certainly, you could write a script to automatically compile all your files and link them, but then everything would be re-compiled each time you ran the script.

The command `make` is designed to avoid that problem. With `make`, you can easily specify a complex compilation, and simply by typing "`make`" at the command line, have that compilation kick into effect for you. All you need to do is specify some of the compilation details in a file known as a `Makefile`.

There is another benefit, too, however: each time you type "`make`", the system will *only* recompile:

1. The files that you have changed
2. The files that use interfaces that have been re-compiled

For example, suppose you coded a `String` class – with files `string.h` and `string.C` – using an `Array` implementation coded in files `array.h` and `array.C`. Certainly, the `String` files would `#include array.h`, because the `String` files are using the `Array` interface functions.

Now, suppose you had a program that used these files, and you decided after the program was done that you wanted to change some of these files. If you made a change to the `Array` implementation, but not the interface, then only `array.C` would be changed. Now, if you were to re-type `make`, only `array.C` would be recompiled, and then the executable would be linked. Similarly, if all you had changed was `string.C`, then the `make` command would only recompile `string.C` and then re-link. However, if you make a change to `array.h`, then not only would the object file for `Array` have to be recompiled, but also the object file for `String` – since `String` makes use of the `Array` interface, a change in the interface of `Array` potentially affects the compilation of `String` as well, and therefore `String` would need to be recompiled.

However, a file that was unaffected by a change in the `Array` interface would not need to be recompiled, and therefore would not be recompiled (if you set up your `Makefile` correctly).

But, the point is that, once you set up the `Makefile`, all of that is automatically handled whenever you type "`make`".

2.1.2 The `Makefile` explained

Inside the `tutorial` directory you have copied, examine the `Makefile.`. To begin with, you will notice helpful directions throughout, all starting on lines that begin with the pound sign (#). In a `Makefile`, the pound sign is the comment indicator.

Now, look at the first two non-commented lines of the `Makefile`:

```
OBJS = \
        tutfns.o main.o
```

At the start of our `Makefile`, it will be helpful to have a macro that stands for all of the object files needed by the linker. That's what this line is. We have two separate object files, and so we have two different files listed on the line above. In larger programs, where there are many more files, there will be many more object files listed above.

Two things to take note of: First, the slash after the OBJS simply means, "continue on next line". So, the above is really all one line, and if we had many object files and wanted to write them over three or four lines, we would simply put those same slashes at the end of all but the last line, in order to tie all the object files into the same "one-line" macro. Second, the indentation on the second line is NOT eight spaces; it is a tab. *You have to hit the tab key!* If you use spaces, it will be an error and the `Makefile` will not work.

Next, note that we have a few more macros on the next number of lines. We define a name for the executable file, which you can change if you prefer using `mp0` or `jason` or whatever else instead of `a.out`. We also have macros for the compiler and linker and their options. In particular, note that `CC` is the compiler listed, and if we were using `g++` instead, then `g++` would be listed there instead. Also, note the `-g` on the CCOPTS line. The symbol `-g` is a debugging flag, and it will compile the MP with debugging information that can be used by a debugger. However, this adds to the size of the final executable, which is why this is an option rather than always automatically done. If you worked for a company, you would probably want the debugging information to be there when you were developing the code, but you would probably want to compile without it before shipping so that you could ship a program that needed less memory.

Finally, we come to the bottom section, where the dependency specification takes place. Ignore the first two statements for the moment, and move down to the lines that read:

```
tutfns.o : tutfns.h tutfns.C
	$(CC) -c $(CCOPTS) tutfns.C
```

The first line says that the object file `tutfns.o` depends on implementation code from `tutfns.C` and interface code from `tutfns.h`. In general, you will have an object file on that line, followed by a colon, followed by all the files that object file depends on for its own correct compilation.

The second line specifies how to build this object file. Here, to build `tutfns.o` we simply compile `tutfns.C`. However, notice that by using `$()` we are invoking one of our previously defined macros. So, the second line is read by the system as:

```
	CC -c -g tutfns.C
```

Finally, notice that, again, that second line *must be tabbed*, and that your `Makefile` will not work if you use spaces instead.

Look now at the next two lines, the lines for the `main.o` executable. We see that the dependency line for `main.o` has two files: the `main.C` file and also `tutfns.h`. If you look through `main.C`, you will notice that `main.C` makes use of functions declared in `tutfns.h` and defined in `tutfns.C`. So, if `tutfns.h` was changed, that would mean that - potentially - the code in `main.C` no longer works (for example, what if we eliminated the functions in the `tutfns.h` file entirely?) So, since `main.C` uses the interface, or the set of function headers, that `tutfns.h` provides, if the `tutfns.h` file changed, then that means the collection of function headers being offered to `main.C` has changed, and so `main.o` should be recompiled as well, just so that we can be sure it will still work. Hence, `tutfns.h` must appear on the dependency line for `main.o`. (You could leave it out, but you could then end up with compilation and linker errors if you did change `tutfns.h`.)

When you first type "`make`", the system will attempt to create an executable by running the following lines:

```
$(EXENAME):  $(OBJS)
	$(LINK) $(LINKOPTS) $(OBJS)
```

which, given the previously-defined macros, expands to:

```
a.out: tutfns.o main.o
	CC tutfns.o main.o
```

Since our macro definition for `LINKOPTS` was empty, the `$(LINKOPTS)` is replaced by nothing at all.

In order to do the linking step, it is necessary for the object files to be generated, and thus they are, using the lines we discussed above and leaving alone any currently-existing object file that does not need to be re-compiled. Then, once the object files are generated, the compiler links them all together and creates the executable.

Finally, note the lines:

```
clean:
        -rm *.o $(EXENAME)
        -rm -rf SunWS_cache
```

Here, we are specifying a particular command for `make`. When we typed `make`, we could just as well have typed `make a.out`. We don't need to, because in the absence of any extra command line name, the system assumes we mean the executable. However, we could also have compiled individual object files by typing lines such as `make tutfns.o`. Just like we can do that, we can run the lines listed after "`clean`" above by typing `make clean`.

These lines remove *all* object files, the executable file, and the compiler cache, `SunWS_cache`. What you are left with is just your source code – as if you had never compiled in the first place. This is helpful whenever you want to do a fresh compile, from scratch. Why would you want to do this? Well, there are a few reasons. First, errors in the `Makefile` can result in problems, such as when dependency files are changed but the `Makefile` isn't coded to worry about those changes. In that case, once you fix the `Makefile` you generally want to recompile from scratch, and so you first run `make clean` to clean away the old machine code. Second, sometimes there are problems with compilation of some code that makes it difficult to handle a change to that code correctly without a complete recompile. (You will run into this as we deal with templates.) So, if you are ever getting a *really* bizarre linker error or set of linker errors, and you don't know what is causing them, sometimes what you should try is simply running `make clean` and then recompiling from scratch. By allowing the compiler to re-read the code instead of trying to keep up with dependency issues, you often get around the particular problem things are stuck on.

This is not to say that most of your errors will be like this; quite to the contrary, most of your errors will be of your own doing. But, `make clean` is there for the times when you need it, or at least when you think you *might* need it.

That, then, is an explanation of the `Makefile`. Throughout the semester, you will see different `Makefile` files, but the ideas are always the same as those described above. You can get even *more* complicated, though. If you want to learn about `make` in depth, we recommend you look for a comprehensive reference on the matter (O'Reilly publishes a small book on `make`, for example).

2.1.3 Using `make`

Now, while in your `tutorial` directory, simply type "`make`" and hit return. You will see compilation begin, and then stop, once an error is detected. The error is that `main.C` is using a variable called `Debugrray` which is not defined. So, open `main.C` using your choice of text editor, and fix line 4 so that that instead of reading

```
   /* 4  */   AssignMore(Debugrray, 4);
```

it instead reads

```
   /* 4  */   AssignMore(DebugArray, 4);
```

That is, add the capital 'A' to correct the spelling mistake. Now, rerun make, and this time the program should compile correctly. Note that when you rerun make, the object file tutfns.o is not created again. The compiler needed to create main.o again, because main.C had been changed. But none of the files that tutfns.o depends on were changed, so the existing version of tutfns.o had to be accurate and there was no need to re-compile the tutfns.C file to produce a newer version of tutfns.o.

List the files in your directory, and you should see among them tutfns.o, main.o, and a.out. Now, run the make clean command we discussed earlier above, and then list the files in your directory again. All three of those object files should now be gone, leaving you with only the four files you originally copied – tutfns.h, tutfns.C, main.C, and the Makefile.

That's it! You have now completed the Makefile tutorial. You will be using make to compile all your MPs this semester.