Writing Make Files

1. **Example program:**

   Here, we will use a program that deals with some geometric shapes to illustrate writing a make file to compile a program.

   The program is made up of 5 source files:

   - `main.cpp`, a main program.
   - `Point.h`, header file for the Point class.
   - `Point.cpp`, implementation file for the Point class.
   - `Rectangle.h`, header file for the Rectangle class.
   - `Rectangle.cpp`, implementation file for the Rectangle class.

   These files are available to download.

2. **Compiling source code *separately*:**

   Let's review what we need to do to compile this program *separately* (i.e., to intermediate *object files*) and then link it into an executable named `main`.

   To just compile source code, use the `-c` flag with the compiler...

   ```
   % g++ -c main.cpp
   % g++ -c Point.cpp
   % g++ -c Rectangle.cpp
   ```

   This will generate the object files:

   - `main.o` (for `main.cpp`),
   - `Point.o` (for `Point.cpp`), and
   - `Rectangle.o` (for `Rectangle.cpp`)

   Then, to link the object files (`.o`) into an executable, we use the compiler again (although this time it will just pass the `.o` files on to the *linking* stage):

   ```
   % g++ -o main main.o Point.o Rectangle.o
   ```

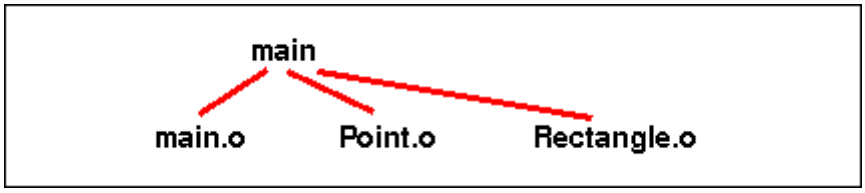
3. **Dependency chart:**

   We would like to know what files need to be regenerated when we change certain parts of our program. We can determine this by creating a *dependency chart*.

   Let's start from our goal (i.e., to have an executable named `main`)...

   Linking Dependencies

   The executable **main** is generated from 3 object files, **main.o**, **Point.o** and **Rectangle.o**. Thus, **main** *depends* on those 3 files.

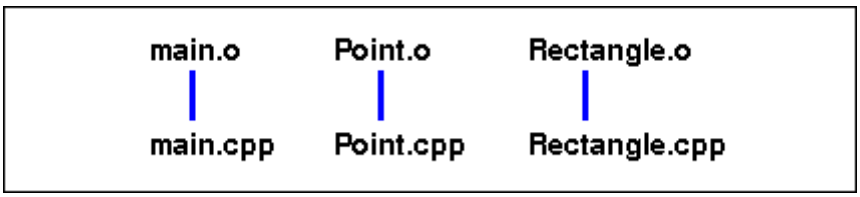


To generate **main** from those files, we *link* them together.

Compiling Dependencies

Next, the object (`.o`) files depend on the `.cpp` files. Namely:

- **main.o** depends on **main.cpp**,
- **Point.o** depends on **Point.cpp**, and
- **Rectangle.o** depends on **Rectangle.cpp**.

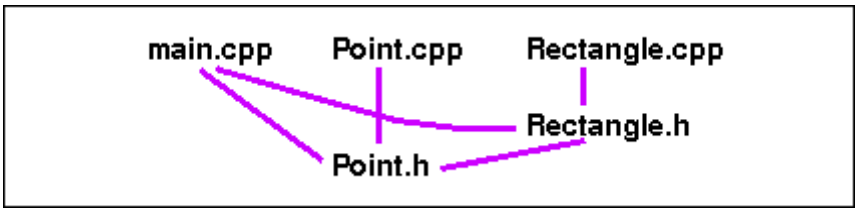


You generate these object files by *compiling* the corresponding `.cpp` files.

Include Dependencies

Finally, source code files (`.cpp` and `.h`) depend on header files (`.h`) that they include:

- 3 source code files include **Point.h** and
- 2 files include **Rectangle.h**.

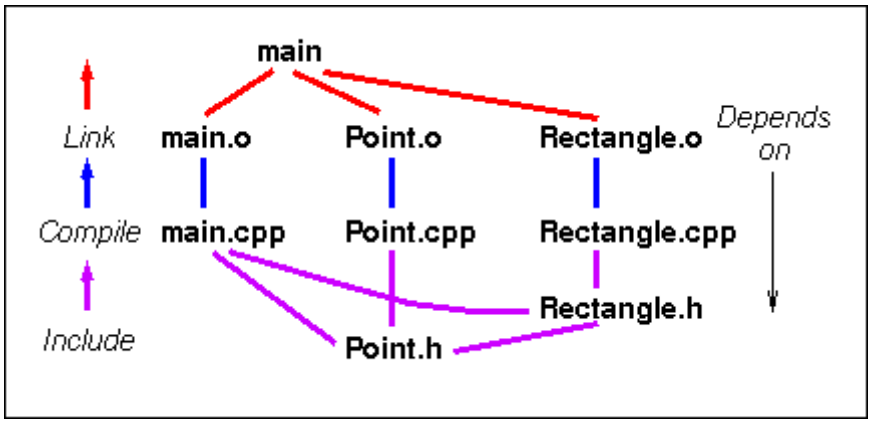


Notice that there may be additional *indirect* include dependencies (e.g., `Rectangle.cpp` depends on `Point.h` since `Rectangle.cpp` includes `Rectangle.h`, which includes `Point.h`).

Note: Include dependencies are a little different since the `.h` files aren't used to *generate* other files, unlike `.cpp` and `.o` files.

Here is the complete *dependency chart*...

Dependency Chart for `main` Executable



Dependencies go downward.
How to "get" or generate files goes upward.

Note: We've just listed all the dependencies for the executable *main*. The executable depends on some things *directly* (like the `.o` files) and some things *indirectly* (i.e., you need the `.cpp` files to generate the `.o` files to generate the executable).

---

4. **Using dependency chart:**

   With such a dependency chart, we can answer questions about what needs to be regenerated if certain files change.

   For example, suppose we change the file `main.cpp`...*What has to be regenerated?*

   Answer: First, recompile `main.cpp` to get `main.o`. Then, relink all the object files together again to get the executable `main`.

   Now, suppose we changed `Point.h`?

   Note: When either a `.cpp` file changes or a header file included (directly or indirectly) by a `.cpp` file changes, we have to regenerate the corresponding `.o` file.

   ---

5. **Make utility and make files:**

   Since it is tedious to recompile pieces of a program when something changes, people often use the *make* utility instead.

   Make needs a *make file* that encodes both the dependencies between files and the commands needed to generate files.

   When you run the *make* utility, it examines the *modification times* of files and determines what needs to be regenerated. Files that are *older* than the files they depend on must be regenerated. Regenerating one file may cause others to become *old*, so that several files end up being regenerated.

   Aside: Note the difference between the *make utility* (a program you run) and a *make file* (that tells the *make utility* how to compile/link a specific program).

   A make file should be named either **Makefile** (first letter uppercase) or **makefile** (all lowercase). The make file should be in the same directory as the source code files.

   Aside: Make files can have other file names, but then you'll have to tell the *make* utility what the name of the make file is (using a commandline option).

   Finally, it is easiest if the make file is in the same directory as the source code files.

6. **Writing a make file:**

   You can create a make file in any text editor. For this example, we will examine the parts of this Makefile, which generates the executable `main`. *Go ahead a download that make file.*

   Simple make files have at least 2 parts:

   - A set of *variables*, which specify things like the C++ compiler/linker to use, flags for the compiler, etc.
   - A set of *targets*, i.e., files that have to be generated.

   Make files may have comments that (hopefully) add to readability. Any line starting with a pound sign (#) is a comments. Note that our make file has some comments.

Note: Keep in mind that make files use their own *language*, so don't expect things in them to look like programming languages you know.

Variables

The first variable listed in our make file is for the compiler to use...

```
CXX = g++
```

In general, the form for setting a variable is:

```
VARNAME = value
```

(The space around the = is optional for our version of *make*, but adds to readability).

The second variable says what flags to pass to the C++ compiler...

```
CXXFLAGS = -Wall -g
```

Here, the variable includes the flag for *all warnings* and the one that adds *debugging information* (so that the executable can later be run under a debugger).

The 2 variables, `CXX` and `CXXFLAGS`, are the ones that our version of *make* expects to be set for the C++ compiler and flags to the C++ compiler, respectively.

Aside: You can add your own variables, but *make* has a set of standard variables, like these, which it uses for the compiler, flags, etc.

Targets

With the needed variables, we can deal with the targets, which are files that must be *generated*.

For each target, there are typically 1 or 2 lines in a make file. Those lines specify:

1. its dependencies (easy to determine from a dependency chart)
2. and possibly a command to generate the target (easy to determine from knowledge of separate compilation).

By default, the first target in the make file is the one that gets generated when you just say "`make`" at the commandline, so it should be the name of the executable.

Target: Executable

Let's examine the first target in our make file (i.e., for `main`)...

For a target's dependency line, the target file name should be listed, followed by a colon (`:`) and the files it depends on:

```
main: main.o Point.o Rectangle.o
```

For a target that is an executable, we just list the object files it depends on.

On the next line, there should be a command that generates the target:

```
<Tab>$(CXX) $(CXXFLAGS) -o main main.o Point.o Rectangle.o
```

Note: This line **MUST** start with a `<Tab>` (i.e., hit the Tab key)...it cannot start with a regular space!

Note that the variables `CXX` and `CXXFLAGS` are used, which means that the command is really:

```
g++ -Wall -g -o main main.o Point.o Rectangle.o
```

Note: Using a variable in a make file involves using the dollar sign (`$`) and then the variable name in parentheses.

Targets: Object Files

Our next target is the object file, `main.o`.

For a target that is an object file, we list all files it depends on. This means its corresponding `.cpp` file, plus any header files its `.cpp` file includes (i.e., include directly or indirectly, but not system header files like `iostream` that aren't going to change).

So, the dependencies line looks like:

```
main.o: main.cpp Point.h Rectangle.h
```

The generation command for this target is:

```
<Tab>$(CXX) $(CXXFLAGS) -c main.cpp
```

Make is kinda smart

We could have written the `main.o` target more simply. *Make* already knows how to generate certain kinds of files. Let's take advantage of *make*'s smarts for the next target...

The next object file, `Point.o` depends on `Point.cpp`, however, *make* already knows that `.o` files can be generated from their corresponding `.cpp` files, so all we need is:

```
Point.o: Point.h
```

Note: Although some *make* utilities can automatically determine what `.h` files a file depends on, some cannot, so you should list the *include* dependencies.

We don't need any *generation* line since we set up the variables `CXX` and `CXXFLAGS`. *Make* will use them to figure out a compilation command.

Finally, there is only one target left, `Rectangle.o`. Taking advantage of *make*'s smarts, it only requires:

```
Rectangle.o: Rectangle.h Point.h
```

Note that `Rectangle.o` depends on `Point.h` indirectly (i.e., recall that `Rectangle.cpp` includes `Rectangle.h`, which includes `Point.h`).

That's all for the make file!!! It just encodes the dependencies and the generation commands for 1 executable and its 3 object files.

7. **Using a make file:**

With a make file for our executable, it is easy to compile that executable.

Try using the make file we've given you. First, remove the old executable and object files (if there are any):

```
% rm main *.o
```

Run *make* with the command:

```
% make
```

Remember that this will cause *make* to generate the *first target* in the make file, which is `main`.

Since nothing has been compiled/linked, it will do:

```
g++ -Wall -g -c main.cpp
g++ -Wall -g   -c -o Point.o Point.cpp
g++ -Wall -g   -c -o Rectangle.o Rectangle.cpp
g++ -Wall -g -o main main.o Point.o Rectangle.o
```

I.e., compile the 3 `.cpp` files into object files, and then link the object files into an executable named `main`.

(If you have trouble, be sure that the make file is named **Makefile** and is in the same directory as the source code files.)

Now, type the *make* command again (let's explicitly tell it what target to generate this time) and you get:

```
% make main
make: `main' is up to date.
```

Here, *make* tells you nothing has changed, so it has no work to do.

Next, suppose we change the program by adding some code to the end of `main()` (in `main.cpp`) to move the rectangle and then print out its new top right point:

```
r1.move(2, 3);
cout << "\nRectangle r1 moved by 2, 3--top right now at "
     << r1.get_top_right().get_x() << ", "
     << r1.get_top_right().get_y() << endl;
```

Edit `main.cpp`, adding those lines, and then save to disk.

After the change, when we use *make*:

```
% make
g++ -Wall -g -c main.cpp
g++ -Wall -g -o main main.o Point.o Rectangle.o
```

it doesn't bother regenerating `Point.o` or `Rectangle.o`, since the files those depend on have not changed.

Finally, try changing `Point.h` and then *re-make*...

```
% make
g++ -Wall -g -c main.cpp
g++ -Wall -g   -c -o Point.o Point.cpp
g++ -Wall -g   -c -o Rectangle.o Rectangle.cpp
g++ -Wall -g -o main main.o Point.o Rectangle.o
```

Since all the `.cpp` files depend on that header (either directly or indirectly), they all have to be recompiled and linked together.

8. **Additional notes on *make*:**

Other versions of *make* may use different variables for C++ instead of `CXX` and `CXXFLAGS`.

In a make file, if you need to continue a line, you cannot just continue it on the next line. You must end a line with a `\` (backslash, not the forward slash) to tell *make* that the line continues. Only break lines

where space would normally go.

So, the first target could have been written:

```
main: main.o \
      Point.o \
      Rectangle.o
<Tab>$(CXX) $(CXXFLAGS) -o main \
          main.o Point.o Rectangle.o
```

We have only described the basics of *make* here. Most versions of *make* have additional capabilities.
