C++ Header Files and Makefiles

Forward Declarations

- Recall that if you try to use a function before it is declared, you will get a compiler error
- To get around this we use *forward declarations*
  - That is, we put a copy of the function signature at the top of a program (or before its first use)
  - E.g. `int someFnc( int x, int y );`
- We also need to use forward declarations when using multiple files in our program.
- But having to add a function signature every time can get tedious...
- Instead we can include all necessary declarations in a *header file*

C++ Header Files

- Header files (or `.hpp` files) are libraries of code that can be included in any program
  - Once included, the contents are available as if you had declared them within that very file

E.g.

test.cpp:

```
#include< stdio >
#include< iostream.hpp >

int x;

int main(){
    x = 5;
    cout << x;
    return 0;
}
```

or

test.hpp:

```
int x;
```

&

test.cpp:

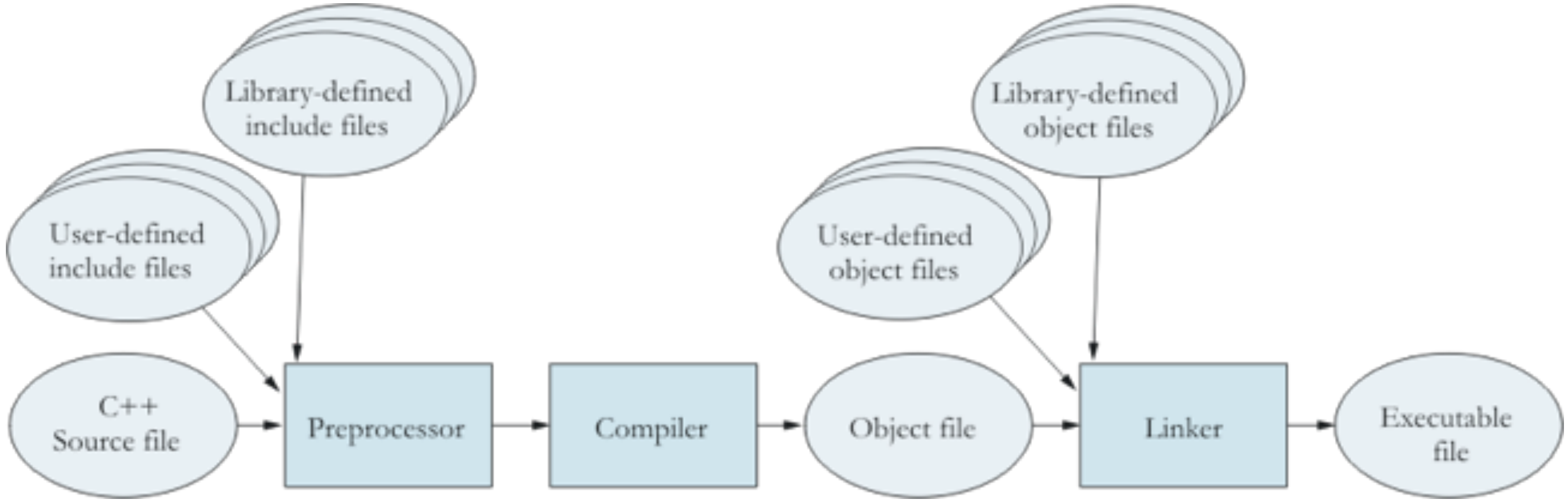
```
#include< stdio >
#include< iostream.hpp >
#include “test.hpp”

int main(){
    x = 5;
    cout << x;
    return 0;
}
```

NOTE: C++ header files are sometimes saved as `.h` files, but this should be reserved for C programs only.

C++ Header Files

- The include lines that put in each program you write include the necessary standard (and standardized) libraries, such as for input and output
  - Including `iostream`, for example, allows you to use `cout`, `cin`, etc
  - The standard, compiler-supplied libraries are always included in angle brackets, without the "`.hpp`"
- Typically header files include only declarations
  - In the case of the `iostream` library, the implementation of features like `cout` is in the runtime support library.

Compiling and Linking (reminder)



Library-defined include files
User-defined include files
C++ Source file
Preprocessor
Compiler
Object file
User-defined object files
Library-defined object files
Linker
Executable file

Makefiles

- How can we manage a larger project with multiple files?
- A makefile!
  - Lists the sources files that make up your current project, and any UNIX commands used to compile the programs
- You can use nearly any text editor to create a makefile
- Once written it is called using “make”
  - This will run any makefile called “Makefile” or “makefile”

Makefiles

- Suppose you have a program called "`prog.cpp`" and a header file called "`prog.hpp`" and you want to produce an *executable* called "`prog`".

Makefile:

```
prog: prog.cpp prog.hpp
	g++ -Wall -g -o prog prog.cpp
```

- Consider the first line:
  - To the left of the colon (`prog`) is the *target*
  - To the right is the *dependency* (`prog.cpp`)
  - That is, the target file depends on `prog.cpp`
  - A target can have any number of dependencies

Makefiles

- When running, make will check the timestamp on any dependent files.
  - If it is changed, it will re-compile. If not, make will tell you that it is up to date.

Makefile:

```
prog: prog.cpp prog.hpp
	g++ -Wall -g -o prog prog.cpp
```

- Consider the second line:
  - This is the UNIX command used to compile
  - This line **must be indented with a tab**
    - (Be careful when cutting & pasting!)
- All of this together is a rule.
  - A makefile can have one or more rules.

Makefiles

Makefile:

```
prog: prog.cpp prog.hpp
     g++ -Wall -g -o prog prog.cpp
```

- The -g enables debugging information
- The -Wall enables all warning messages
- The -o indicates the executable file name
  - Be sure to include this or you could accidentally overwrite your source file!
  - E.g. this is wrong!

```
g++ -Wall -g -o prog.cpp
```

More complicated Makefiles

- In the example we just looked at, it was probably too simple to bother with make.
- Let’s look at a more complicated program (feel free to use this as a template)

Makefile:

```
firstprog: prog.o novowels.o columns.o
	g++ -Wall -g -o prog prog.o novowels.o columns.o
prog.o: prog.cpp prog.hpp
	g++ -Wall -g -c prog.cpp
novowels.o: novowels.cpp
	g++ -Wall -g -c novowels.cpp
columns.o: columns.cpp
	g++ -Wall -g -c columns.cpp
```