ALGONQUIN COLLEGE

MODULE 4 : COMPILATION 1 – MAKEFILES AND CPP DIRECTIVES

Professor : Dave Houtman

Office: T323

Office Hrs: Monday 15:30 – 16:00

Wednesday 15:30 – 16:00

Friday 15:30 – 16:00

Email: houtmad@algonquincollege.com

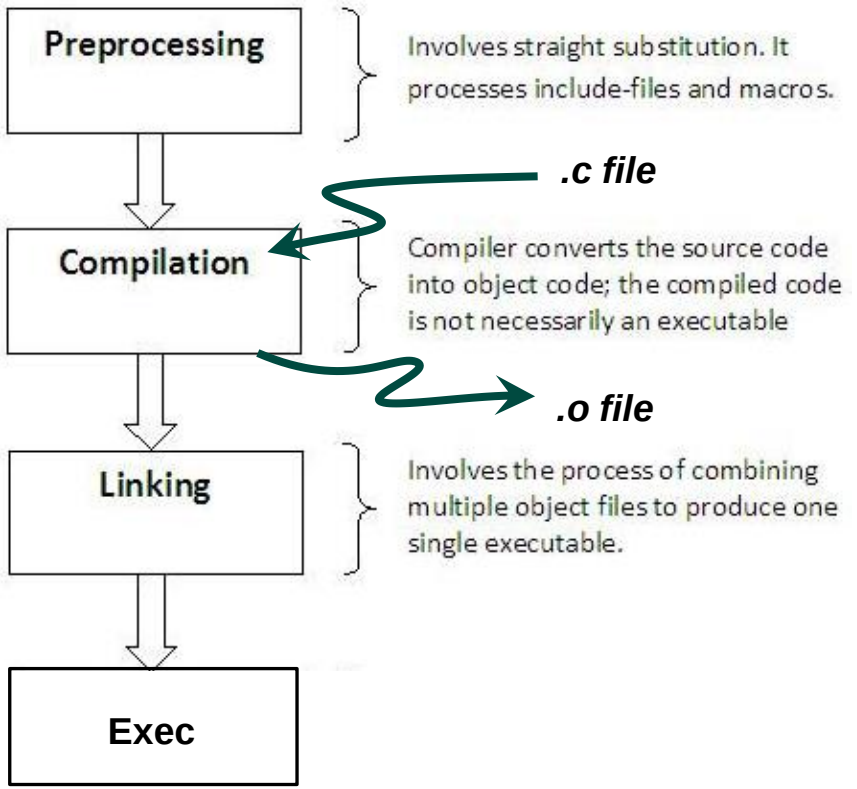
4.1 Controlling C code compilation

- Prior to actual compilation, your C code is affected by two scripting languages that control the compilation and linking processes. While not technically part of the C language itself, each acts as a kind of auxiliary language that helps with the process of creating a new executable file.

    1) Almost every C program contains one or more CPP directives which are embedded in the C code and indicate what parts of the code are to be compiled. Preprocessor directives always begin with '#'; there are twelve such directives in standard C, but (as usual) certain C compilers allow for additional directives
    2) The compiler/linker has its own scripting language, which is stored in a ***makefile***. While it is almost impossible *not* to use at least one preprocessor declaration in your C code, you can avoid using makefiles entirely for most simple projects. However, since industrial-sized projects use makefiles, even though our projects are fairly simple, we still need to know about how to use makefiles.

4.2 Makefiles: an overview

- *Prior to* compilation, the CPP executes *all* preprocessor directives (i.e. everything starting with `#`) and executes these statements before compilation begins.
- During the compilation stage, each `.c` file in the `gcc` command line is converted to an ***object file*** (indicated by .o at the output).
- For code containing multiple object files, linking between these different modules needs to be done before an executable can be created



K&R 187

4.2 Makefiles: an overview

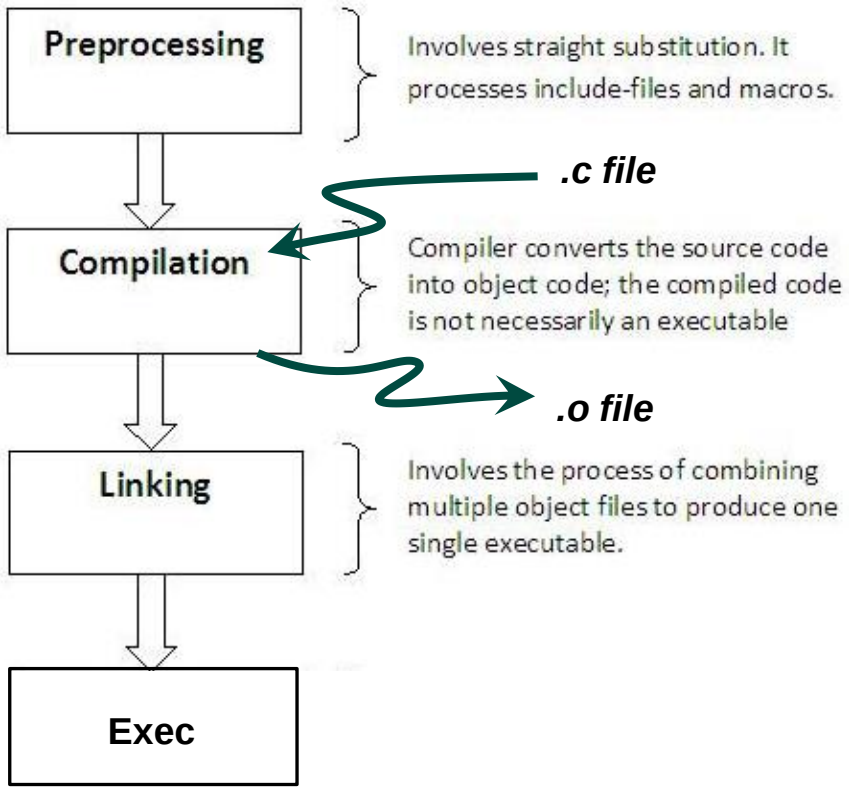
- Large C/C++ projects may contain thousands of functions stored in hundreds of files; it becomes unwieldy to attempt to recompile such a large number of files at the command line. For example, the Win32 API is a library consisting of a ~2000 C functions, referenced via a header file called `windows.h`. This was followed by MFC (Microsoft Foundation Class), a C++-based library which encapsulates the Win32 API's functionality in a series of OOP classes, making the business of programming applications somewhat more manageable—but still leaving the programmer with the problem of compiling hundreds of libraries.
- Not only is it time-consuming for the programmer to deal with all these files, it is time-consuming for the compiler as well. For example, since many of the files will be dependent on .h files like `<stdio.h>` and `"myoutput.h"`, the compiler spends a great deal of time recompiling these files each time it encounters an `#include` directive. Moreover, each time you change part of a program, you have to recompile the entire program all over again.

4.2 Makefiles: an overview

- The `make` utility uses a file named `makefile` to simplify compilation of large projects. *Makefiles* facilitate the construction and maintenance of large C projects, and they speed compilation by recompiling only the part of the program where changes have occurred.
- To see how makefiles work, consider again the compilation process. Files are compiled to object files, which contain a mixture of compiled code and information that allows different code modules to link to one-another. The linker handles the final step of assembling all the pieces into an executable.

4.2 Makefiles: an overview

- One consequence of this is that, so long as an object file hasn't changed, there's no need to recompile it: that's the shortcut that allows makefiles to speed the recompilation process. But before you can use a makefile, you need to tell the ***make utility*** how to deal with all of the modules of your application.



4.2 Makefiles: an overview

- In the following example, we assume that two files, called `main.c` and `myfile.c`, are to be compiled at the gcc command line using

```
gcc main.c myfile.c -std=c99 -o outfilename
```

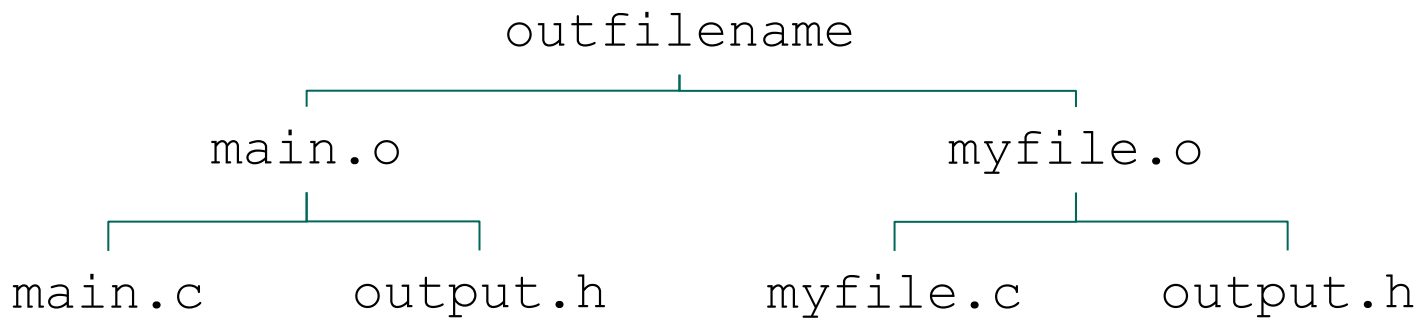
- Furthermore, assume each contains a reference to a header file, `output.h`, which in turn contains a number of `#defines`, `const` variables and `typedefs`. (It could also contain references to `extern`-declared functions, but for now, that's not part of this example.) So `main.c` and `myfile.c` both contain the line

```
#include "output.h"
```

near the start of their code. Since any change to `output.h` can affect both `main.c` and `myfile.c`, the object files created by the compiler are dependent upon the information stored in `output.h`.

4.2 Makefiles: an overview

- We could draw a ***dependency tree*** of the situation, as follows:



This is exactly the set of relationships that we need to script into our makefile, so that the make program 'knows' which branches of the tree will need to be recompiled, and which not, whenever a file in the tree changes

4.3 Using makefiles

- A makefile is a text-based file that consists largely of a series of ***rules***. Each rule has two components. It lists, first

1) The ***dependencies*** between files. In the above example, the executable file `outname` is dependent upon two object files. This is listed as:

```
outfilename: main.o myfile.o
```

This says: `outfilename` (the name of the executable file) depends upon the `main` and `myfile` object files.

- Note the format of the dependency. The dependent file is listed first, followed by a colon, followed by the files it depends on, each separated by spaces.

```
target: dependent_file1.ext dependent_file2.ext
```

4.3 Using makefiles

2) The ***command*** or ***action line*** describes the action(s) that must be taken to fulfill the dependency, i.e what the compiler must do in order to satisfy the dependency. In the above case, the `gcc` compiler must be invoked to output the executable file, `outname`, based upon the two object files listed in the dependency. Hence the action line or command line associated with the above dependency will be:

```
gcc –o outfilename main.o myfile.o
```

This says: `gcc` should create the ouput/executible file `outfilename` using the two object files on which it depends, namely `main.o` and `myfile.o`, as listed in the dependency on the previous slide.

Putting these two pieces together, we have the following rule:

```
outname: main.o myfile.o
	gcc –o outname main.o myfile.o
```

4.3 Using makefiles

- Spacing is important in this language; you can't just throw spaces and tabs in when you please. For example, the dependency line *should* be *flush* against the left margin. The command *must* begin with the tab character (⟷ below). The following example shows three rules:

```
outname: main.o myfile.o                  # dependency line
⟷gcc -o outname main.o myfile.o           # command line
```

- Note that # is used to signal comments in this language—not to be confused with its use in CPP directives.

4.3 Using makefiles

- In the above example, the executable file depended upon object files, hence its dependents will have the .o extension. But where do the .o files come from? .o files depend upon the .c and .h files, therefore these dependencies need to be listed as well. With makefiles, one rule may depend on a rule that *follows* it.

```
main.o: main.c output.h      #main.o depends on 2 files
    gcc -c main.c            #compile main.c to main.o
```

- Similarly, `myfile.o` depends upon `myfile.c` and `output.h`

```
myfile.o: myfile.c output.h  #myfile.o depends on files
    gcc -c myfile.c          #compile myfile.c->myfile.o
```

4.3 Using makefiles

- So our complete makefile describes the relationship between each of the files created along the way, starting with the executable file, and then describing, layer by layer, its dependencies.

```
outname: main.o myfile.o                    # .exe dependency
     gcc -o outname main.o myfile.o         # make .exe from .o

main.o: main.c output.h                     # .o depends on .c,.h
     gcc -c main.c                          # make .o from main.c

myfile.o: myfile.c output.h                 # .o depends on .c,.h
     gcc -c myfile.c                        # make .o from myfile.c
```

4.3 Using makefiles – review of gcc options

- Since we need to control the output from `gcc` exactly, the following options are important for this discussion:

Option	Operations
`-S`	Stop after compilation; do not assemble. The output is in the form of assembler code.
`-c`	Compile or assemble source files to .o, but do not link
`-o`	Write the output to the file specified; assumes compilation is carried out to completeness

- gcc compilation can be altered by literally hundreds of parameters. See

  `http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html`

- The above options are useful for debugging multi-file programs: they allow you to find out whether a bug occurs in the compilation stage, or only in the linking stage.

4.3 Using makefiles – review of gcc options

- The following options, while not central to makefiles, are also important. You've already encountered some of them:

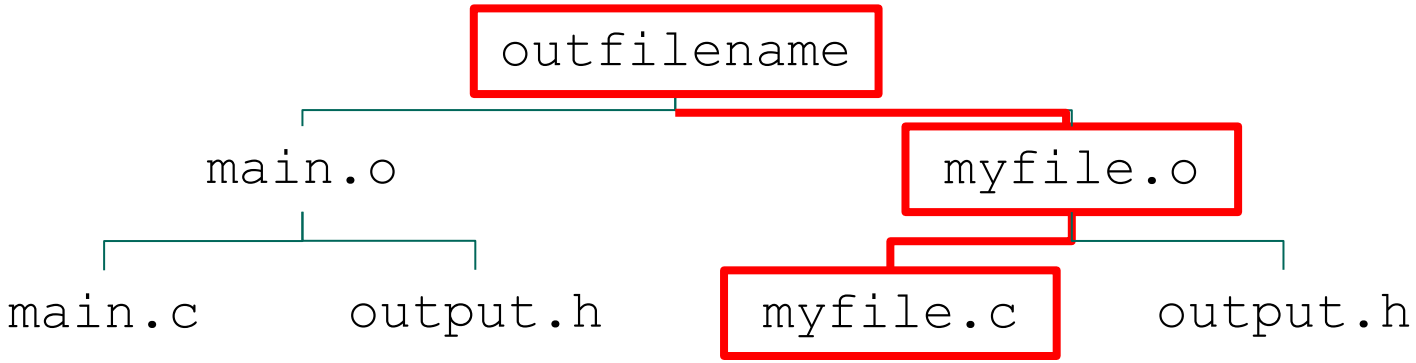
Option	Operations
`-std=c99`	Determines which C and C++ standard to use for compilation; in this case, the c99 standard is to be used.
`-Wall`	-W issues various types of warnings; `-Wall` issues *all* of them
`-ansi`	Supports ISO C89 programs; an early, minimum standard for C
`-pedantic`	Used with `-ansi`; demands strict ISO C; issue all warnings
`-x` *`language`*	Compile for language specified; this may be `c`, `c++`, `objective-c`, `ada`, `fortran`, `assembler`, `java`, …
`-v`	Verbose mode; print all commands run during compilation

- When use the `-v` option, its useful to pipe the output to a file. Try using:

```
gcc myFile.c -o outputFileName -v 2>outfile.txt
```

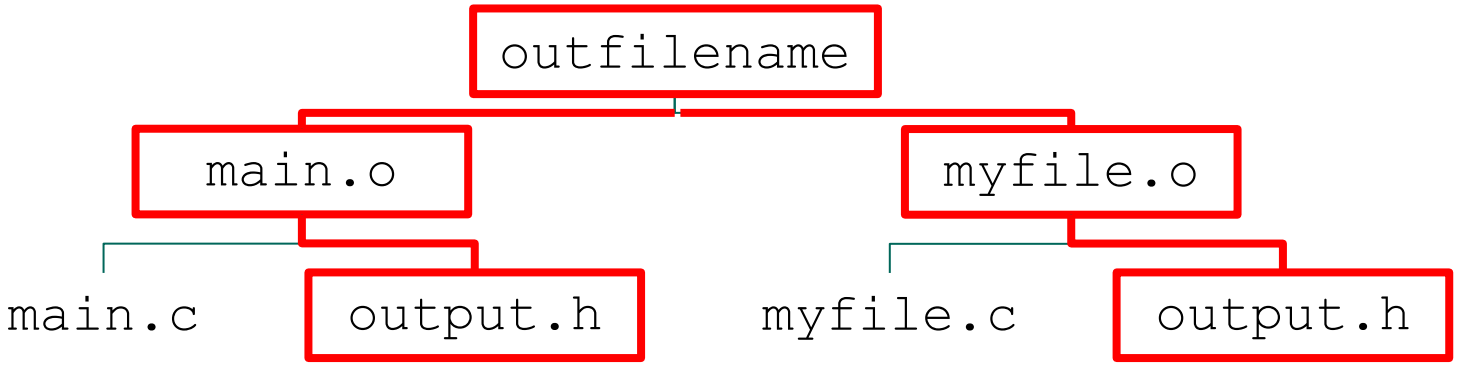
4.3 Using makefiles

- What is the purpose of all this information? As we build the tree, the `make` utility keeps track of which files will need to be changed whenever it, or its dependencies, are updated. For example, if we change `myfile.c`, `myfile.o` will need to change as well. But `main.o` will not; the `make` utility will *not* need to recompile the left branch of the dependency tree.



4.3 Using makefiles

- But if we change `output.h`, both `main.o` and `myfile.o` will need to change.



- The `make` utility will need to recompile both `main.c` and `myfile.c` since both their object files are dependent upon `output.h` according to the lines:

```
main.o: main.c output.h
myfile.o: myfile.c output.h
outname: main.o myfile.o
```

4.3 Using makefiles

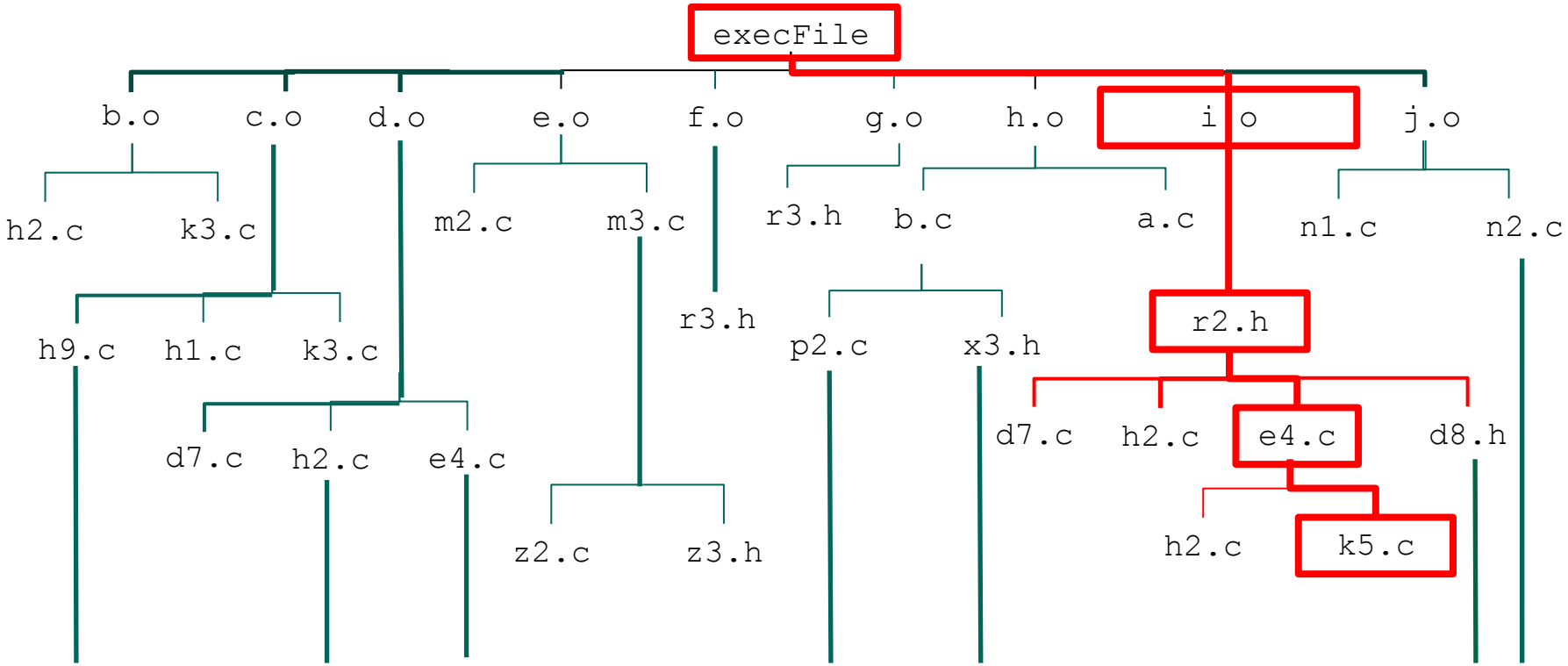
- By telling `make` how these files are connected, we tell it how to keep track of which branches of the tree will need to be recompiled when changes are made (using the dependency line), and how to do this (using the action line).
- Since we tell `make` about user-defined files like `output.h`, why not tell it about system libraries like `stdio.h`? Why not use:

```
main.o: main.c stdio.h output.h
```

Why not? Because these never change, and only files that change will affect compilation.

4.3 Using makefiles

- To see the real value of makefiles, imagine a dependency tree consisting of several hundred branches. Rather than recompile the entire structure each time a single change was made, makefiles allow you to recompile just the affected branches:



4.3 Using makefiles

- Makefiles also allow for the use of variables, which act as placeholders. When variables are used, the `$` symbol is used to signal this fact to the `make` utility.

- For example, in the following makefile, we could change the compiler used by altering the first line:

```
CC = gcc                # variable definition; use gcc

outname: main.o myfile.o
    $(CC) -o outname main.o myfile.o

main.o: main.c output.h
    $(CC) -c main.c

myfile.o: myfile.c output.h
    $(CC) -c myfile.c
```

4.3 Using makefiles

- This allows variable names to be assigned to certain commonly-used commands/tokens used during compilation/linking. `$(xx)` is used to indicate that the variable named *xx* should have its actual value/name inserted into the dependency line or command/action line. For example:

```
CC = gcc                              # define gcc as CC
CFLAGS = -ansi -pedantic -Wall        # set compiler flags
OBJS = main.o myfile.o                # set obj files to OBJS
PROG = outname                        # set outname to PROG

$(PROG): $(OBJS)
        $(CC) $(OBJS) -o $(PROG)
main.o: main.c output.h
        $(CC) $(CFLAGS) -c main.c
myfile.o: myfile.c output.h
        $(CC) $(CFLAGS) -c myfile.c
```

4.3 Using makefiles

- Notice that this allows us to :
  - Change the compiler. Perhaps we wish to use the Visual C++ compiler on our code, rather than gcc. Simply change the CC variable to the new compiler.
  - Change the compilation options to something more relaxed. We could just use `CFLAG=`, with no assignments, and remove the requirement to compile using the more rigorous `-ansi -pedantic -Wall`
  - Change the output name of the file. Simply change the `PROG` variable at the top of the makefile.

4.3 Using makefiles

- Additionally, we can override the filenames in a consistent way that allows us to substitute .o files for .c files., using the ":" directive:

```
PROG = myfile.c
SOURCE = $(PROG:.c=.o)
```

The above line says: change all occurrences of .c in **PROG** to .o and set the result equal to **SOURCE**. In this limited case, **SOURCE = myfile.o**

4.3 Using makefiles

And then the rules in the main body of the makefile go from

```
myfile.o: myfile.c output.h
	$(CC) $(CFLAGS) -c myfile.c
```

to

```
$(SOURCE): $(PROG) output.h
	$(CC) $(CFLAGS) -c $(PROG)
```

4.3 Using makefiles

- We can tokenize the entire makefile in such a way that, in order to recompile a large program containing several files, we need only to change the words assigned to the makefile variables.

- All large software vendors use some form of makefile to build their projects. For very large projects such as OSs, makefiles are essential.

- Makefiles can also be used to keep track of version control. For example, say 'Build 999' works with one set of files, but 'breaks' when programmers makes changes to their code, so that 'Build 1000' fails. Having backups of these files and the makefile that put the program together allows for faster project recovery, and makes detecting the bad file(s) much easier.

4.3 Using makefiles

- Too compile and link a program using makefile, type

```
make
```

at the command line. The default file is called `makefile`. If you want to override the default file name (which you should, for clarity), you must specify the name of the file name explicitly:

```
make Lab5v201
```

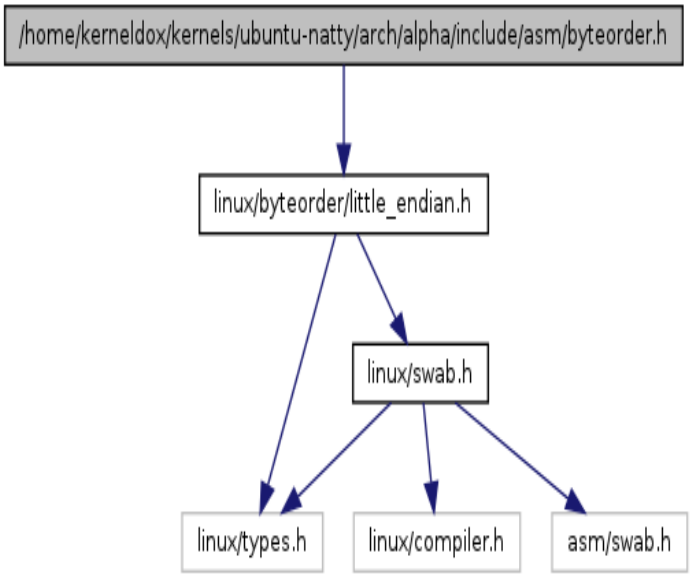
4.3 Using makefiles

- Note that dependency trees are used in UNIX/LINUX documentation to describe the relationship between files:

byteorder.h File Reference

```
#include <linux/byteorder/little_endian.h>
```

Include dependency graph for byteorder.h:



Go to the source code of this file.

http://kerneldox.com/kdox-ubuntu-natty/dd/d48/arch_2alpha_2include_2asm_2byteorder_8h.html

4.3 Using makefiles

- A modern IDE consists of several tools tied seamlessly together (which, historically, were separate programs) including an editor, compiler, linker, debugger, and project manager—programs which are still dealt with separately (and non-graphically) in the C world. Makefiles are there in the IDE, controlling the actual compilation, but they're largely hidden from view, working automatically (until there's a problem, and you need to change the default behaviour of the compilation/linking process.) Chances are, whenever you click on a checkbox to select an IDE option related to the behaviour of your executable code, you are throwing a switch—somewhere—in a makefile.

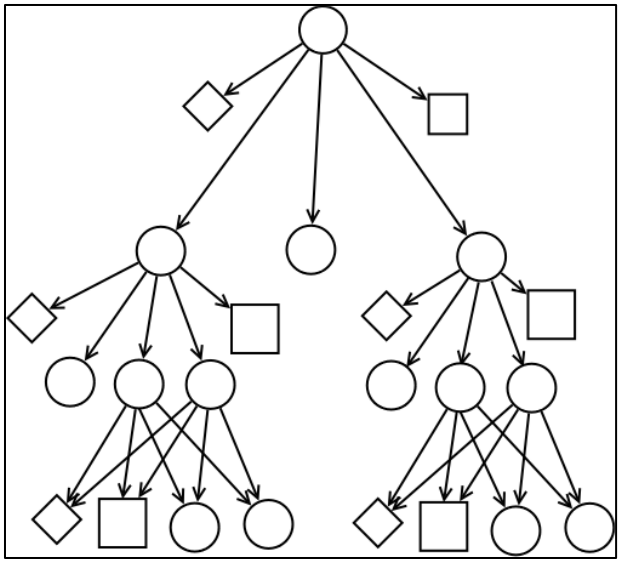
- Microsoft has its own versions of `gcc` and `make`. The compiler is called `CL.EXE` (compiler and linker) and `make` is known (universally, or at least throughout the MS universe) as `NMAKE.EXE`. For details, the documentation at http://msdn.microsoft.com/en-us/library/9s7c9wdw.aspx forms a starting point.

4.3 Using makefiles

- Makefiles form a rather large topic in their own right. For additional information, these web pages form a good starting point:

http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
http://courses.cms.caltech.edu/cs11/material/c/mike/misc/make.html
http://www.cs.bgu.ac.il/~sadetsky/openu/myMakeTutorial.txt
http://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/makefile.html
http://www.cprogramming.com/tutorial/makefiles.html
http://www.delorie.com/djgpp/doc/ug/larger/makefiles.html
http://web.stanford.edu/class/cs107/guide_make.html
http://www.jfranken.de/homepages/johannes/vortraege/make_inhalt.en.html

Example 13 : Using a makefile



Problem Description:

Using `Example 11`, which tested the `SimpleStats` library, create a makefile that automates the process of compilation and linking.

Example 13: Using a makefile

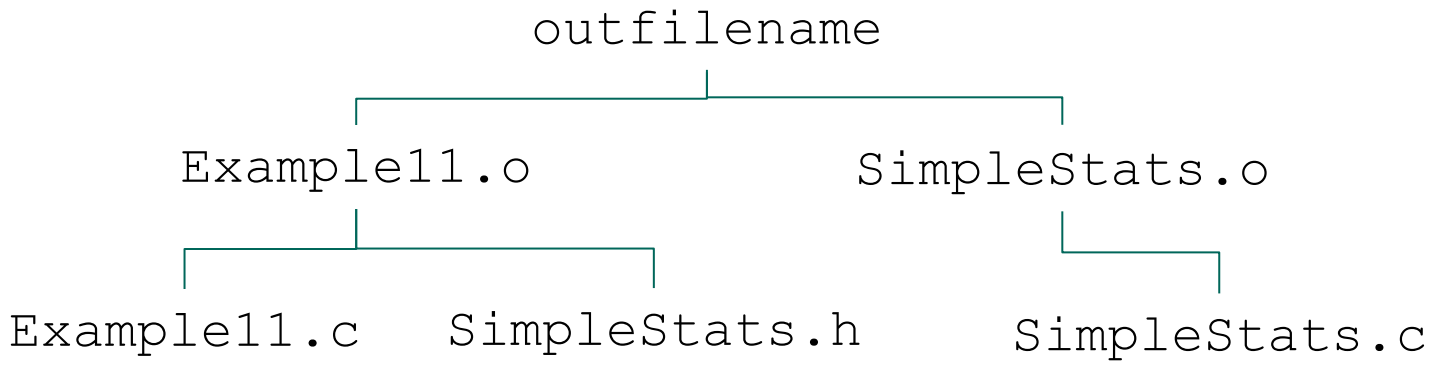
```
houtmad@ubuntu:~/examples$ make
gcc                          -c Example11.c
gcc                          -c SimpleStats.c
gcc                         Example11.o SimpleStats.o -o myoutfilename
houtmad@ubuntu:~/examples$
```

Following a change to Example11.c, the new output becomes

```
houtmad@ubuntu:~/examples$ make
gcc                          -c Example11.c
gcc                         Example11.o SimpleStats.o -o myoutfilename
houtmad@ubuntu:~/examples$
```

Example 13: Using a makefile

The dependency tree for this project was:



Example 13: Using a makefile

The makefile corresponding to this dependency tree would therefore be:

```
CC = gcc                          # define gcc as CC
PROG = myoutfilename              # set outname to PROG

$(PROG): Example11.o SimpleStats.o
      $(CC) Example11.o SimpleStats.o -o $(PROG)

Example11.o: Example11.c SimpleStats.h
      $(CC)  -c Example11.c

SimpleStats.o: SimpleStats.c
      $(CC)  -c SimpleStats.c
```

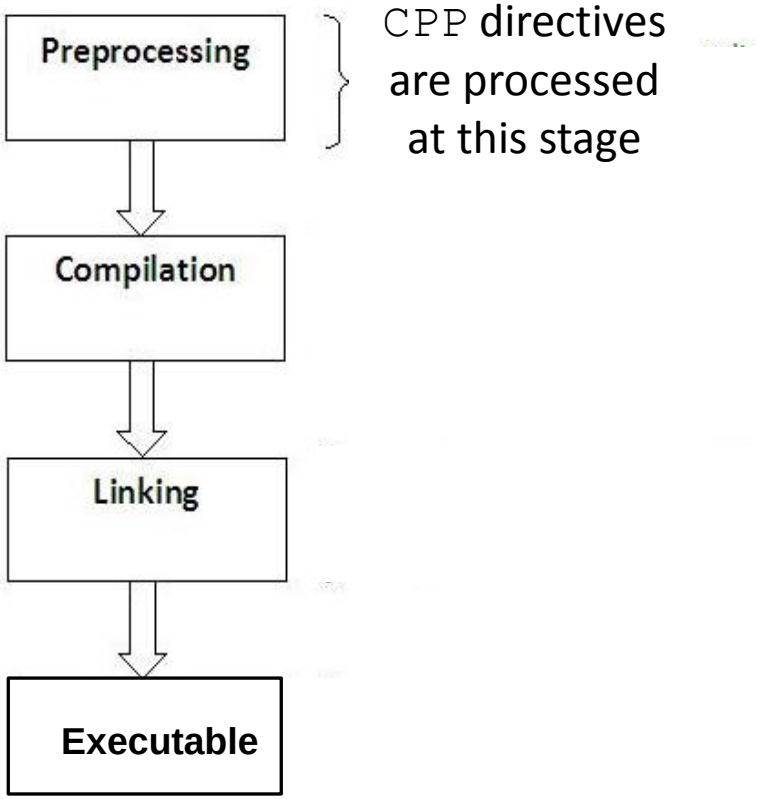
4.3 Using makefiles

A few simple words of advice about makefiles are in order:

- While the default name of the file used by the `make` utility is `makefile`, you should give each of your makefiles a special name; don't rely on the default name exclusively. This ensures you've always got 'backup' makefiles to fall back on
- Large makefiles may require documentation as extensive as that required by the C programs they support. Therefore, you should make liberal use of `#comments`.
- A dependency tree clearly indicates the associations between files. It forms an excellent way to compose your ideas about the construction of the makefile. There should be one rule for each branching of the dependency tree (even when the branch only includes a single item).

4.4 The Standard CPP directives

- There are a dozen standard C preprocessor (CPP) directives in C; you've seen two of them: `#include` and `#define`.
  - The # symbol indicates a CPP directive
  - CPP directives are deleted from your code prior to actual compilation.
  - CPP directives cannot exceed a single line in length; however, the '\' character can be used to continue a long line on the next line.



4.4 The Standard CPP directives

Directive	Purpose
`#include`	Used to include .h and .c files
`#define`	Used to define single-line macros and constants
`#undef`	Undefines a symbolic constant previously defined with `#define`
`#ifdef`	Determines flow of compilation based on *string literal definition*
`#ifndef`	"If not defined", the opposite of `#ifdef`
`#if`	Determines flow of compilation based on a *Boolean result*
`#elif`	Works like `else if` in C; conditional on non-zero Boolean result
`#else`	Works like `else` in C; execution occurs if `#if` or `#ifdef` is false
`#endif`	Ends a conditional block that begins with `#if`, `#ifdef`, or `#ifndef`
`#error`	Generates error message output during precompilation
`#line`	Used to redefine line numbers in the output; used for debugging
`#pragma`	Used to provide compiler-dependent information

4.4 The Standard CPP directives – Checking for defined values

- Once a symbolic constant is defined, it can be used to signal that a particular action should be performed. While you are used to see `#define` used like this

```
#define CM_PER_INCH 2.54
```

it can also be used like this:

```
#define SETFLAG0
```

which simply creates the token SETFLAG0, without assigning a value.

4.4 The Standard CPP directives – Checking for defined values

- We can then test to see if `SETFLAG0` exists or not using either

```
#ifdef SETFLAG0
```

or

```
#ifndef SETFLAG0
```

where `#ifndef` means "is not defined"—the opposite of `#ifdef`

4.4 The Standard CPP directives – Checking for defined values

- Once a symbolic constant is defined, it is normally defined for the lifetime of the program. This is not always desirable, since we may wish to redefine a constant for another purpose. `#undef` allows the programmer to *undefine* a symbolic constant, allowing it to be reset at some point during preprocessing.

K&R 189

For example, assume we have downloaded and included a library in our C program called `AMSMathConstants.h` that contains the value of PI listed to a very high number of significant figures, much larger than those given in `math.h`:

```
#define PI 3.14159265358979323846264338327950288419716939937\
51058209749445923078164062862089986280348253421\
17067982148086513282306647093844609550582231 7
```

4.4 The Standard CPP directives – #undef

But now assume we wish to include a file of Unicode characters whose value is defined in a file called `"unicode.h"`. Assume that the symbolic value `PI` is defined in this file as well, but this time it is set to its Unicode value:

```
#define PI L"\u03C0";
```

Clearly, `PI` cannot have two different values. There will be a compilation error if this situation is allow to persist. The solution is to conditionally 'undefine' one of the `PI`s at some point during the compilation so that a conflict does not occur.

4.4 The Standard CPP directives – #undef

For example:

```
#include AMSMathConstants.h
   // Mathematical value of PI is defined…
   // Do something mathematical with it
   …
#undef PI
   // mathematical PI is now undefined
   …
#include "unicode.h"
   // Unicode PI is now defined…
   // Use it to print out the symbol π
   …
#undef PI
   // Now no PI is defined
```

4.4 The Standard CPP directives – Conditional Compilation

- The standard CPP directives also include `#if`, `#elif`, and `#endif` For example, consider code compiled for three two hardware platforms, which vary according to the size of RAM on board. If `RAM` is either not defined or was undefined, an appropriate error is generated by the CPP. If the value of `RAM` is defined correctly, compilation will include one of the two libraries available, *but only one*. CPP directives thus have the ability to affect *what* code gets compiled, and what does not, *directly affecting the size of the executable*.

```
#define RAM 128
#if (RAM==32)
   #include SmallMemoryModel.h
#elif (RAM==64)
   #include MediumMemoryModel.h
#else
   #error "Incorrect RAM size set"
#endif
#ifndef RAM
   #error "RAM size not defined"
#endif
```

4.4 The Standard CPP directives – Conditional Compilation

- In this example, `#ifdef` is used to determine which type of sorting algorithm to use in the code depending of whether or not the CPP value `SMALL_N` is defined or not. (Note that `SMALL_N` does not need to be set to anything; it only needs to exist).

K&R 191

```
#define SMALL_N
   ...

#ifdef SMALL_N
    sorted = mergeSort(a[]);
#else
    sorted = qSort(a[]);
#endif
```

4.4 The Standard CPP directives – Conditional Compilation

- Certain CPP directives are defined automatically in different OSs. The following can be used to determine whether code is being compiled for the Windows operating system or for Unix:

```
// _WIN32 is defined for Windows, but not UNIX.
// So it can be used to determine which header
// to include for compilation

#ifdef __unix__
   #include <unistd.h>
#elif defined(_WIN32)
   #include <windows.h>
#else
   #error "unknown operation system"
#endif
```

Examples from : http://stackoverflow.com/questions/1352784/preprocessor-examples-in-c-language and http://en.wikipedia.org/wiki/C_preprocessor

4.4 The Standard CPP directives – Conditional Compilation

- ANSI C (but not traditional C) contains a preprocessor operator, `defined()`, which returns a boolean value based on whether it's argument has been defined or not. So given

```
#define SOMETHINGEXISTS
```

The statement

```
#ifdef SOMETHINGEXISTS
```

works exactly the same as

```
#if defined(SOMETHINGEXISTS)
```

You may see either of these used in code. Functionally, they are identical.

4.4 The Standard CPP directives – Conditional Compilation

- Other examples of where preprocessor directives could be used to affect compilation include:
  - bigEndian v. littleEndian issues, e.g. Intel CPUs v. Motorola CPUs
  - Compilations having output in different languages for different markets, e.g. for the Canadian market, we need French and English versions for government products
  - 2 byte v. 4 byte integers
  - Version control e.g. we may have 'TODO's in code for the next version. For the production version of the code, these TODOs should be suppressed. This can be done by wrapping the TODOs in CPP conditional ifs.

4.4 The Standard CPP directives – Conditional Compilation

- gcc allows predefined symbolic constants to be passed to the CPP. This uses the `-D` option at the command line, so

```
gcc -Dname ...
```

tells gcc to treat *name* as a predefined value.

If you use

```
gcc -DNDEBUG ...
```

then the CPP treats `NDEBUG` as a predefined value. Why is this important? Because `assert()` checks to see if `NDEBUG` is declared or not. If `NDEBUG` *is* declared, *`asserts are turned off`*. Therefore, to compile your code so that the end-user does not see those awkward assertions in your code, set the `-DNDEBUG` option when using gcc to produce your commercial code.

4.4 The Standard CPP directives – `#line`, `#error`, and `#pragma`

- Three other miscellaneous CPP directives are available for use: `#error`, `#line`, and `#pragma`.

K&R 192

`#error` is used to signal an error condition; it is a standard format output error message. Could you output a `printf()` at this point instead. Sure, but that assumes that your code gets compiled first; this message is triggered during the precompilation stage, and so is more valuable for catching errors that would otherwise be impossible to see otherwise.

`#line` is used to insert a line number into your code, which may or may not correspond to the actual line number of the code. This is useful for labelling each section of code starting with a unique line number; this when '`error on line XXXX`' occurs, you know exactly where to start looking.

4.4 The Standard CPP directives – `#line`, `#error`, ⌘ and `#pragma`

- The **`#pragma`** directive is compiler-specific. Compiler vendors may use it for their own specific purposes. It is"often used to allow suppression of specific error messages, manage heap and stack debugging and so on" (http://en.wikipedia.org/wiki/C_preprocessor)

- gcc has a few specific pragma directives of its own. Each of these begins with the word `gcc`. For example:

```
#pragma gcc poison wordlist
```

`poison` causes the compiler to flag an error whenever a word in *`wordlist`* is present in the code. For example, if you decided `scanf()` is too dangerous to use and wanted to ensure it was not used in your code:

```
#pragma gcc poison scanf
```

See: http://gcc.gnu.org/onlinedocs/cpp/Pragmas.html for gcc's pragmas

4.5 Predefined types and macros

- More generally, CPP directives can be used to solve a problem commonly encountered in the world of mixed 32-bit and 64-bit architectures. As explain earlier, the data type returned by the `sizeof()` operator may be either `%u` or `%lu` depending on your architecture. So you could use the following to solve this problem:

```
#ifdef 64BITS
   printf("%lu", sizeof(int));
#else
   printf("%u", sizeof(int));
```

4.5 Predefined types and macros

- In actual fact, C solves this problem for you, having 'predefined' a solution.

`size_t` is a data type `typedef`ed to hold an `unsigned int` large enough to store any address used by your computer. This is the natural data type returned by `sizeof()` for your computer—in fact, for any computer on which `sizeof()` is used; it is universally transportable across platforms. The format specifier for `size_t` is **`%zu`**.

– think of `%zu` as being optionally `%u` or `%lu` depending on your bus architecture. So when using `sizeof()`, in order to ensure your code is transportable, you would write something like:

```
size_t sizeOfArray = sizeof(someArray);
printf("Size of the variable:%zu\n", sizeOfArray);
```

This solves the problem of having to write either `%u` or `%lu`.

4.5 Predefined types and macros

- In addition to specially defined types, ANSI C contains five predefined macro values that are defined automatically when you start your program. They cannot be overridden or removed. Thus, the following words should be treated along with the C keywords as 'variable names to be avoided'

Predefined Macro	'Type'	Value
__DATE__	string	Contains the current date
__FILE__	string	Contains the file name
__LINE__	integer	The current line number in a program
__STDC__	integer	Nonzero if the implementation follows ANSI standard C, 0 otherwise
__TIME__	string	The current time

Note that predefined macros are both preceded and followed by two underscore characters (__). As previously stated, as a general rule, avoid using __ in your variables.

K&R 193

4.5 Predefined types and macros

- The `line` directive allows you to insert line numbers into your output. This is often used in conjunction with the `__LINE__` macros and is mainly used in debugging

```
#include <stdio.h>

int main(void) {
   func_1();
   func_2();
}

#line 100
func_1() { // so this is line 101
   printf("func_1-the current line number is %d\n",__LINE__);
}

#line 200
func_2() { // so this is line 201
   printf("func_2-the current line number is %d\n",__LINE__);
}
```

Examples from: http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Flc.htm

4.5 Predefined types and macros

- The output from this program is:

```
func_1-the current line number is 102
func_2-the current line number is 202
```

- This could be used in a more sophisticated fashion to detect potential runtime errors due to alternate compilations. For example,

```
#define DENOMINATOR 32
…
function calcRatio(int numerator){
   #line 0
   #ifndef DENOMINATOR
     puts("possible divide by 0 error in line "
     #__LINE__ " of calcRatio in file " __FILE__);
   #else…
       return (numerator/DENOMINATOR);
}
```

4.6 Miscellaneous

- Note that, it is possible to see the result of all CPP directives just prior to the start of actual compilation using `cpp` at the command line in place of `gcc`:

```
cpp hello.c
```

will dump out the contents of `hello.c` to your screen with all of its CPP directives 'unraveled'. Since this is a rather messy business—even `stdio.h` expands to a few hundred lines of code—a better solution is to pipe the results directly to a file:

```
cpp hello.c >outfile.txt
```

- All of C's preprocessor directives are available in C++ as well. Additionally, C++ adds two other directives: `#import` and `#using`

4.7 C 'functions' revisited

- Many of the functions you use in C are not really functions in the traditional sense at all. In fact, we might describe C's functions as falling into four separate categories:

1) Operators that look and act like functions
2) `#define` macros that look like functions
3) True C functions
4) System calls that use the Linux OS, but which are available to C

4.7 C 'functions' revisited—operators

1. Operators that look and act like functions

- This includes, most famously, the `sizeof()` operator, which despite appearances is just an operator, no different than, say `++`, `*=`, etc. Note also, `sizeof()` is a reserved word—and therefore not a true function. However, `sizeof()` is often described as "the only built-in function in the C language"

- Note that the `defined()` operator, described earlier, is also a 'built-in function' this time of the CPP, not of C language itself.

4.7 C 'functions' revisited—macros

2. `#define` macros that look like functions

- Many of the common functions that you use in C are standard macros defined in systems libraries; they look like standalone functions, but in fact are built up of simpler units using CPP directives. `assert()` is one such 'function'. The actual code for `assert()` might look like this:

```
#if defined(NDEBUG)
   #define assert(ignore) ((void) 0)       /* i.e. ignore it */
#else
   #define assert(expr)                    \
      #if (!expr)) {                       \
         printf("\n%s%s\n%s%s\n%s%d\n",    \
           "Assertion failed: ", #expr,    \
           "in file ", __FILE__            \
           "at line ", __LINE__            \
        abort(); }                         \
#endif
```

(from: A Book on C, Pohl and Kelly, pg. 388-389)

4.7 C 'functions' revisited—macros

2. `#define` macros that look like functions

- Other useful macros are found in the '`get`' and '`put`' families of macros. These replace `printf()` and `scanf()` in most day-to-day operations involving string output:

Macro Name	Operation
`getchar(void)`	Reads in a single character from the keyboard
`putchar(int c)`	Outputs the character corresponding to c
`gets(str)`	Reads in an array of chars `str` from the keyboard
`puts(str)`	Outputs the array of chars str to the console

K&R 205

4.7 C 'functions' revisited—true C functions

3. True C functions

- This includes all 'low-level' functions like `printf()` and `scanf()`. However, it can be surprisingly difficult to distinguish 'true' functions from macro functions that look and act similar and are often more powerful than their low-level counterparts.

  Why does this matter? When errors occur in functions, it is important to understand whether the function is a macro—in which case its code probably exists in a .h file and can be more easily resolved—or whether the problem lies in a true low-level function in a .c library, making the cause of the error more difficult to resolve.

4.7 C 'functions' revisited—system calls

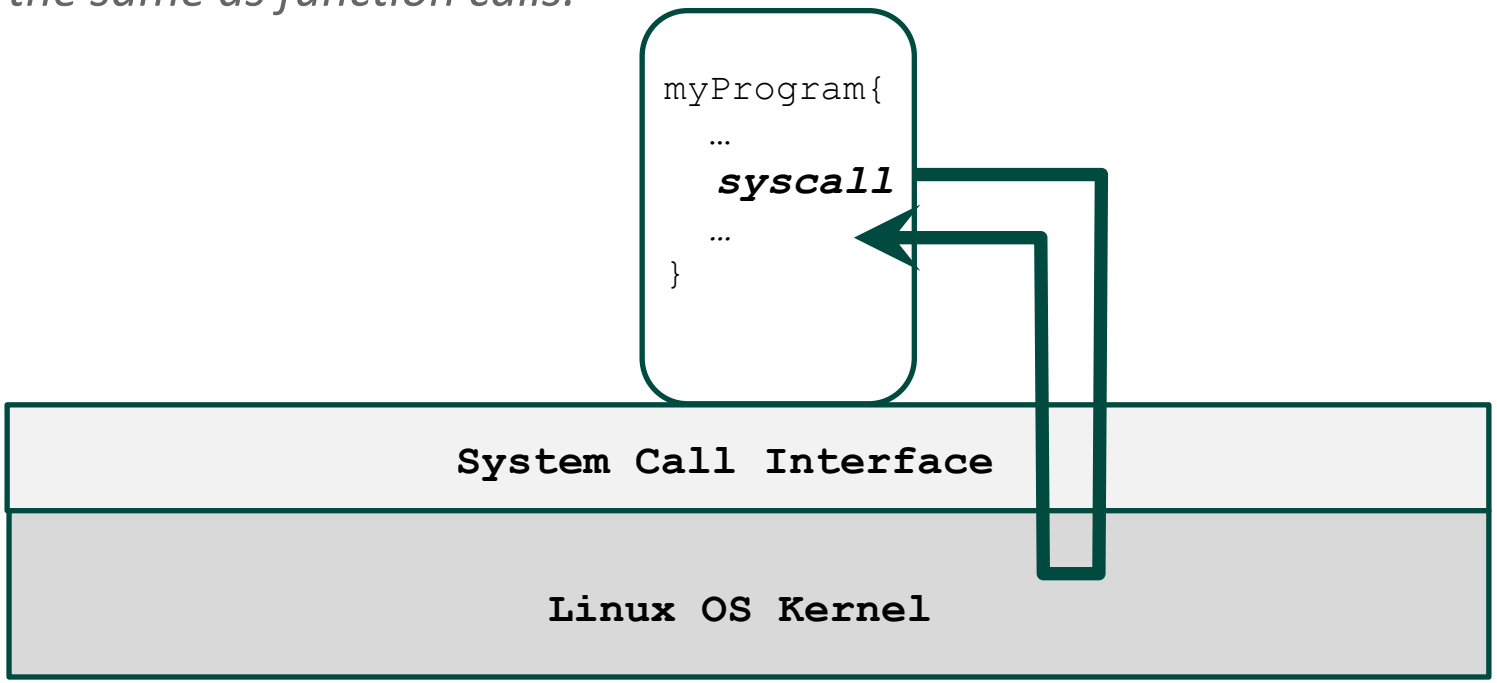
4. System calls that use the Linux OS directly

- Because of the connection between C and UNIX/LINUX, it is possible to call a small set of Linux functions directly from within a C program. You've already encountered some of these OS functions in examples, such as `system()`, `pause()`, and `sleep()`.
- In principal, rather than existing in libraries available to the C compiler, ***system calls*** are executed *directly* by the Linux kernel.
- In actual practice (i.e. this is compiler dependent) system calls are wrapped inside a C library function and require a `'.h'` header file to execute.

4.7 C 'functions' revisited—system calls

4. System calls that use the Linux OS directly

- System calls work by making (again, in principle) a direct call to the Linux kernel. Following execution inside the kernel, control returns to the calling process. *To the programmer, system calls look and act exactly the same as function calls.*



4.7 C 'functions' revisited—system calls

4. System calls that use the Linux OS directly

- Because they use the operating system so directly, system calls are extremely powerful and allow us to do, in C, things which would be impossible in other languages.

- Amongst the most powerful of these is the call to `fork()`, which we shall use later with regard to network programming. `fork()` allows the user to split off separate execution streams. Each program is given its given its own `pid`, or process ID, and continues executing in is own time-slice, which is allocated by the operating system. Thus `fork()` allows for multiple threads of a program to be executed in isolation, each with its own stack space and execution thread.

4.8 POSIX

- POSIX stands for "Portable Operating System Interface", an open standard for operating systems.
- It is an API that defines a standard set of functions, constants, and interfaces that allow applications to be ported between different Unix-based OS's. (It was a subsystem of Windows OS's up until Windows 2000, but not afterward).
- Since the Mac OS is UNIX-based, Mac OSX is POSIX compliant
- Other fully- or mostly-compliant POSIX OS's include
  - *LynxOS*
  - *Solaris*
  - *QNX*
  - *Linux*
  - *VxWorks*
  - *Cygwin*
  - *OpenBSD*
  - *HP-UX*

  …in other words, most of the major UNIX releases
- It is an IEEE, ISO and ANSI standard

4.8 POSIX

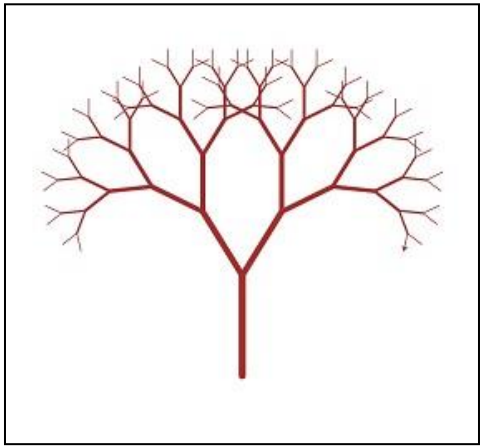
- Because of the relationship between UNIX and C, the POSIX library (below) is largely consistent with the standard C library.

assert.h	*fenv.h*	*pthread.h*	***stdio.h***	*utime.h*
complex.h	***float.h***	*setjmp.h*	***stdlib.h***	***wchar.h***
ctype.h	*inttypes.h*	*signal.h*	***string.h***	*wctype.h*
dirent.h	*iso64.h*	*stdarg.h*	*sys/stat.h*	
dlfcn.h	***limits.h***	*stdbool.h*	*tgmath.h*	
errno.h	***locale.h***	*stddef.h*	***time.h***	
fcntl.h	***math.h***	*stdint.h*	***unistd.h***	

Libraries used in examples or labs in this course are indicated in boldface type.

Programming Challenges

Create a makefile for Example 12b



Problem Description

Create a makefile for Example12b, which relied on both the `SimpleStats.c` library and the `DistributionDisplay.c` library. Furthermore, if you have access to `ConsoleData.h` and `ConsoleData.c` libraries, use them as well (modify your code so the display will never run off the end of the console and wrap around the screen). If you have access to the Unicode character set, you might want to try outputting something other than '*' to the histogram.