UNIX and C Programming (COMP1000)

Lecture 2: Environments

Updated: 5th August, 2019

Department of Computing
Curtin University

Textbook Reading (Hanly and Koffman)

For more information, see the weekly reading list on Blackboard.

- **Chapter 12: Programming in the Large**
  Unfortunately, some examples in this chapter use structs. We won't discuss structs until lecture 6, so ignore them for now.

Outline

Multi-file C Programs

- Source files should not get too long.
  - Giant .c files are difficult to work with.
- So, most C programs are split into multiple files.
- Separate .c files can be used to group related functions.
  - And they *should* be related!
  - Files containing unrelated functions are also difficult to work with.
- One .c file should contain the `main()` function.

Calling Functions in Different .c Files

- Each .c file is compiled separately.
- But you'll need to make function calls *between files*.

calc.c

```
double square(double n) {
    return n * n;
} /* Not visible from main.c! */
```

main.c

```
int main(void) {
    double result, input = 5;
    result = square(input);
    ...
} /* Compiler doesn't know what "square" means. */
```

- How do we fix this?

External Declarations

- We need a declaration:

calc.c

```
double square(double n) {
    return n * n;
}
```

main.c

```
double square(double n); /* Declaration */

int main(void) {
    double result, input = 5;
    result = square(input);
    ...
} /* Compiler is happy with this. */
```

But then things get messy...

calc.c

```
double square(double n) { return n * n; }
double cube(double n)   { return n * n * n; }
```

main.c

```
double square(double n); /* Declarations */
double cube(double n);
...
result = square(5) + cube(5);
```

aardvark.c

```
double square(double n); /* Repeated from above */
double cube(double n);
...
printf("%lf %lf\n", square(x), cube(y));
```

Don't Repeat Yourself (the DRY principle)

- We could end up repeating the same declarations many times.
  - Say `calc.c` has 5 functions, and 10 other files use those functions.
  - That's 50 declarations.
- This is a very bad idea.
  - Copying and pasting is easy, but. . .
  - *Changing* those declarations is time consuming and prone to mistakes.
- So, we put all the declarations for `calc.c` in a "header file".
- When a file needs to use a function from `calc.c`, we write:

```
#include "calc.h"              /* Note: .h not .c */
```

- This takes the declarations in the header file `calc.h` and puts them *right here*.

Header File — Example

calc.c

```
double square(double n) { return n * n; }
double cube(double n)   { return n * n * n; }
```

calc.h (header file)

```
double square(double n); /* Declarations */
double cube(double n);
```

main.c

```
#include "calc.h" /* Include header file */
...
result = square(5) + cube(5);
```

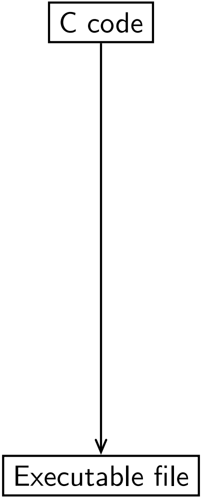
(Put "`#include "calc.h"`" in all files using `square()` or `cube()`.)

Header Files — Summary

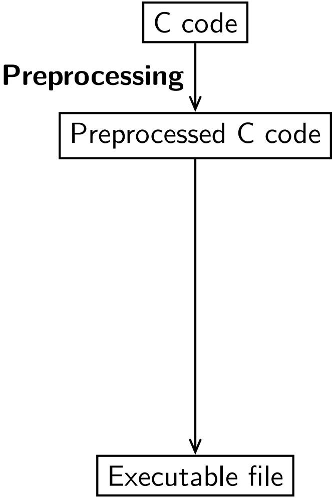
In a multi-file C program:

- Each .c file has a corresponding .h (header) file.
  - (Except perhaps for the .c file containing `main`.)
- A header file declares the functions in its .c file.
- To call those functions from a *different* .c file, that .c file must `#include` the right header file.
- Each .c file also `#include`s its own header file.
  - e.g. `calc.c` would also have "`#include "calc.h"`".
  - Not strictly necessary *for now*.
  - Will become necessary later on, when we declare *types*.

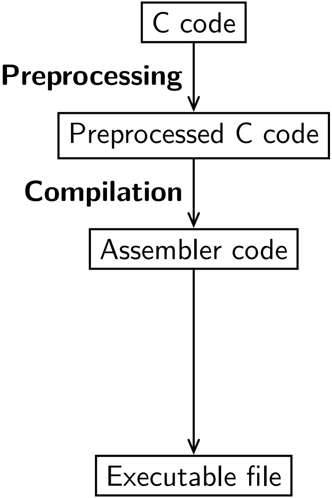
The Compilation Process



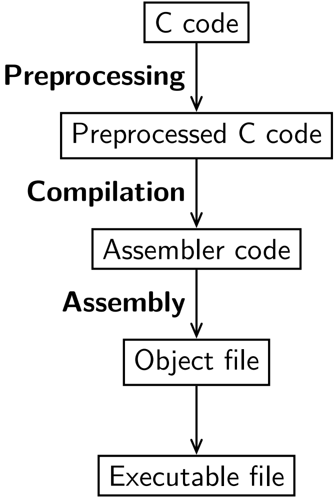
The Compilation Process



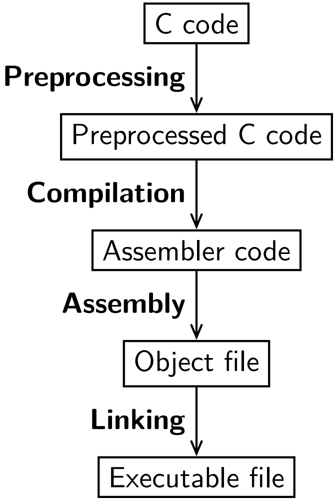
The Compilation Process



The Compilation Process

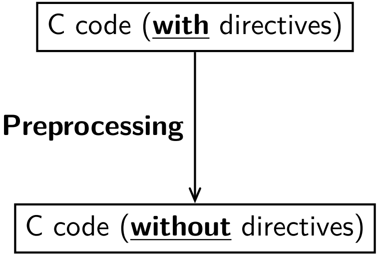


The Compilation Process



The Preprocessor

- A "macro-language", separate from the real C language itself.
- Consists of "directives", each starting with #, embedded in C code.
- (No semicolons required, unlike C itself.)



The #include Directive

- Inserts the contents of a file into the source code.
- In practice, used to access functions defined in other files.
  - (Similar to `import` in Java.)
- Always place these at the top of your code.
- Only `#include` .h (header) files, never .c files.

Examples

```
#include <stdio.h>
```

```
#include "myfunctions.h"
```

- Use `<...>` for standard header files (in pre-defined directories).
- Use `"..."` for your own header files, in the current directory.

The #define Directive

- Assigns a name to a constant value.
- Everywhere that name occurs, the preprocessor replaces it with the given value.
- Used to create constants and “macros”.
- Sometimes useful to place in a header file.

Examples

```
#define PI 3.141592654
```

```
#define OUTPUT_STRING "Hello World!"
```

- There is no equals sign!
- `PI` and `OUTPUT_STRING` are *not* variables!

`#define` — True and False

- C89 has no "boolean" data type (though C99 does).
- Use `int` instead.
- Create constants representing "true" and "false".

Common definition

```
#define FALSE  0
#define TRUE   !FALSE
```

(Why might you want to put this in a header file?)

#define — Macros

- Small snippets of code with parameters (but not functions).
- Works by substitution.

Example (before preprocessing)

```
#define SQUARE(x)  ((x) * (x))

int squareSum(int a, int b) {
    return SQUARE(a + b);
}
```

Result (after preprocessing)

```
int squareSum(int a, int b) {
    return ((a + b) * (a + b));
}
```

#define — Macro Bracketing

- Place brackets around macro parameters.
- Place brackets around the entire macro definition.

Bad example (valid but dangerous)

```
#define SQUARE(x) x * x
```

Good example

```
#define SQUARE(x) ((x) * (x))
```

The #ifdef and #endif directives

A segment of code is only compiled if a given name has been #defined ("conditional compilation").

Example

```
#define DEBUG 1
...
int i, sum = 0;
for(i = 0; i < 100; i++) {
    sum += i;
    #ifdef DEBUG
    printf("%d ", sum);
    #endif
}
```

Without the first line, the preprocessor edits out the printf().

More Conditional Compilation

- Preprocessor names can be defined on the compiler command-line as well:

```
[user@pc]$ gcc main.c -o program -D DEBUG=1
```

- `#ifndef` checks that a given name *has not* been defined.
- `#else` is available for convenience.
- `#if` and `#elif` are slightly more flexible versions.
- These all work like an `if-else` statement, but at *compile time* (not run time).

Avoiding Multiple Inclusion

- With the `#include` directive, some files may be included multiple times.
- This may cause problems in some situations.
- Can be avoided using conditional compilation.

Example (where "…" is the normal header file contents)

```
#ifndef FILENAME_H
#define FILENAME_H
...
#endif
```

Only the first inclusion will count, because `FILENAME_H` will be defined from then on.

Assembly Language

- “Compiling” translates source code into “assembly language”.
- This is a simple language, but extremely verbose and barely human readable.
- Different brands of CPU often require different assembly languages.
- An “assembler” (e.g. the one inside gcc) translates assembly code into machine code.

```
addNumbers:
    pushl %ebp
    movl  %esp, %ebp
    movl  12(%ebp), %eax
    movl  8(%ebp), %edx
    leal  (%edx,%eax), %eax
    popl  %ebp
    ret
```

Machine Code

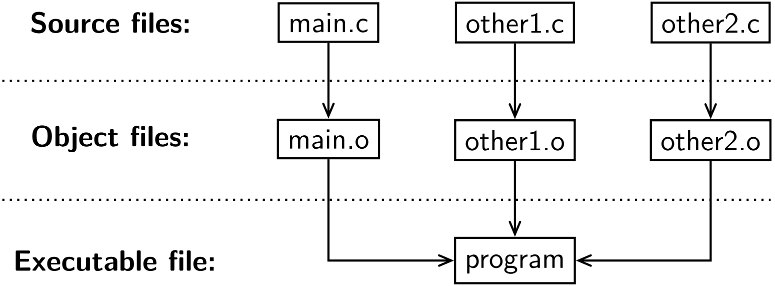
- The machine-code form of a program is the code that *actually runs*.
- Machine code is a compacted form of assembly language.
- Each instruction takes only a few bytes (many only 1 byte).
- There are no names, spacing or syntactical constructs.
- None of it makes sense when simply printed on the screen.
- Much of it cannot be printed at all, because it doesn't match any printable characters.
- That is, it is *not* human-readable!

Object Files

- Preprocessing, compiling and assembly are just sequential steps:
  - One input and one output each.
  - Together, all three are often called "compiling".
- The last output is an object (.o) file for each .c file.
  - (This has *nothing* to do with object orientation!)
- .o files contain compiled code, but they are not executable.
- They contain functions and function calls that have not yet been "linked" to each other.

Linking

- Takes one *or multiple* .o files and produces a single executable.
- Determines how to physically arrange functions in memory.
- Connects function calls to the actual functions, by translating their names to their memory locations.
- This makes function calls possible (especially *between* files)!



Linking with gcc

- To compile a single .c file into an object (.o) file *without* linking:

```
[user@pc]$ gcc -c filename.c
```

- To link multiple .o files into an executable file:

```
[user@pc]$ gcc obj1.o obj2.o ... -o executable
```

- For example:

```
[user@pc]$ gcc -c main.c -Wall -pedantic -ansi
```

```
[user@pc]$ gcc -c other1.c -Wall -pedantic -ansi
```

```
[user@pc]$ gcc -c other2.c -Wall -pedantic -ansi
```

```
[user@pc]$ gcc main.o other1.o other2.o -o prog
```

Libraries

- Often you'll need to link to an external library.
- Sometimes you'll need more than just the `#include` directive.

Example

Under Linux, if you use the math library, you need to:

- place "`#include <math.h>`" at the top of your code, **and**
- when linking, use the "`-lm`" switch (lowercase L, not 1).

```
[user@pc]$ gcc -lm main.o calc.o -o mathyprogram
```

Linker switches

The "`-lm`" switch is specific to the math library. For other libraries, see the manpage.

Storage class specifiers

- These keywords can form part of a declaration: `auto`, `register`, `typedef`, `extern` and `static`. e.g.

```
static double x = 3.0;
```

- They are *not* part of the datatype, but indicate *where* a variable is stored.
  - `auto` is the default for "local" variables (and is almost never explicitly written).
  - `register` tries to store variables in CPU registers.
  - `typedef` causes the declaration to create a new datatype, rather than a variable. (This is discussed in lectures 3 and 6).
  - `static` and `extern` are discussed on the next few slides.

Local variables

- A “local” variable is the kind you already know.
- Created inside functions.
- Only accessible from within that same function.
- Can also be restricted to one pair of braces in a function.

Example

```
void theFunction(void) {
    int x = 0;     /* Only exists in theFunction. */
    ...
    if(...) {
        int y = 5; /* Only exists in the 'if'.    */
        ....
    }
}
```

Global variables (evil!)

- Created *outside* functions — stand-alone.
- Accessible from *all* functions.
- Possibly accessible across different source files.
- Creates a mess! (High coupling.)
- **Avoid** global variables. Instead, use parameters to pass data between functions! (This discussion is merely FYI.)

Example

```c
#include <stdio.h>

int globalVar = 42;      /* Never do this! */

void printGV(void) {
    printf("%d", globalVar); /* prints 42 */
}
```

The extern storage class

- `extern` can be used on both function and variable *declarations*.
- It means that the *definition* occurs somewhere else.
- For functions, this is true anyway, so `extern` is redundant.
- Consider these equivalent declarations:

```
float theFunction(int x, int y);
```

```
extern float theFunction(int x, int y);
```

  The `extern` form often appears in header files, but only as a reminder.
- For variables, `extern` allows you to access global variables across files. This is both complicated and dangerous, so we won't waste our time on it!

The `static` storage class

- `static` can also be used on function and variable declarations.
- Two distinct meanings, depending on where it occurs:

1. `static` makes a function (or global variable) *inaccessible* from outside this file.

```
static void privateFunc(int x, int y) { ... }
```

   - Good practice for functions that don't need to be accessed elsewhere.
   - Vaguely similar to the "private" keyword in OO languages (like C++ or Java), but don't confuse .c files with classes!

2. `static` makes a local variable *persistant* throughout the program's runtime.

Static local variables

- Ordinary local variables disappear when a function ends.
- `static` local variables *don't* disappear.
- They keep their values between function calls.
- Initialised only once, when the function is first called.

Example

```
void count(void) {
    static int counter = 0;
    counter++;
    printf("%d\n", counter);
}
```

`counter` increases each time `count()` is called, and never resets until the program ends.

The Terminal vs. the Shell

The Terminal

- The window where input and output occurs.
- Takes a program's output and displays it.
- Gives a program its input from the keyboard.

The Shell

- A program that runs *other* programs within the terminal.
- Interprets and executes your commands.
- Temporarily gives control of the terminal to another program, when you run it.

Shells

- Many different shells are available:
  - `sh` (Bourne shell) and `bash` (Bourne Again shell);
  - `csh` (C shell) and `tcsh`;
  - `ksh` (Korn shell);
  - `zsh` (Z shell);
  - `ash` (Almquist shell);
  - `cmd.exe` (the Windows command prompt);
  - And others.
- These all do essentially the same thing, in different ways.
- `bash` is the most popular on Linux and OS X.

UNIX/Shell Commands (1)

- We've talked a lot about how to run gcc, but let's step back.
- `gcc` is one command among many.
- You should be familar with a few other commands as well:
  - `ls` — list files;
  - `cd` — change directory;
  - `pwd` — show the present working directory;
  - `cat` — concatenate and output files (or just one file);
  - `cp` — copy files;
  - `mv` — move files;
  - `rm` — remove (delete) files;
  - `mkdir` — make directory;
  - `rmdir` — remove directory;
  - `echo` — print a message;
  - Etc.

UNIX/Shell Commands (2)

- The shell provides you with a “prompt”:

```
[user@pc]$
```

  As you know, this is where you enter commands. (The prompt itself also contains useful information, if you look closely.)

- Use `man` (“manual”) to get help on a given command:

```
[user@pc]$ man cat
```

  (Note: `man` can also be used with standard C functions.)

- Alternatively, most commands let you do this:

```
[user@pc]$ cat --help
```

Executing Commands

When you type in a command:

1. The shell performs various "expansions" and "substitutions":
   - The symbols `*`, `?`, `~` and `[...]` are "expanded".
   - Variables are replaced with their values (discussed shortly).
   - Many other things...
2. Your input is broken up into words, separated by whitespace. Usually:
   - The first word is the command name.
   - The other words are the parameters.

(Note: this is only a simplistic overview.)

Command Arguments

- Command "arguments" (or "parameters") are strings of text, supplied to a command or program. (A kind of user input.)
  - The user is free to enter any number of arguments.
  - Lecture 4 will discuss how to use them in C programs.
- Arguments starting with a dash ("-") are called "switches", "options" or "flags".
- Switches alter the behaviour of a command.
- Some switches take their own argument (e.g. "-o" for gcc).

$$\texttt{[user@pc]\$}\ \overbrace{\texttt{gcc}}^{\text{command name}}\quad \underbrace{\texttt{-Wall}}_{\text{switch}}\quad \overbrace{\texttt{prog.c}}^{\text{ordinary argument}}\quad \underbrace{\texttt{-o}\quad \texttt{prog}}_{\text{switch with argument}}$$

The Command Name

The command name itself might be:

- An "builtin" command — a feature of the shell itself:

```
[user@pc]$ source file.sh
```

- A filename containing a "/" — an executable file to run:

```
[user@pc]$ ./program apple banana caroot
```

- An executable file in the "search path":

```
[user@pc]$ ls -l Desktop
```

  - The shell searches a list of directories for the file "`ls`".
  - The directories to search are specified by a variable called `PATH` (typically containing `/bin` and `/usr/bin`).

- Standard UNIX commands may be *either* builtins *or* files in the search path.

Asynchronous Commands

- To run a command "in the background", end it with "`&`"
- Particularly useful for commands/programs that open GUI windows:

```
[user@pc]$ gedit somefile.txt &
```

  Here, you can still use the shell while `gedit` is running.
- Backgrounded programs can still output to the terminal, even while you're typing a command
- This can lead to confusion (just be aware of it!)

Shell Variables

- In the shell, all variables are strings.
- No declarations needed.
- They are created and assigned like this:

```
varname="Some text"
```

  - No spaces except inside quotes! Really.
- They are accessed using a $ sign:

```
echo The string is $varname
```

  - The shell will replace "`$varname`" with "`Some text`".
  - `echo` will then display "The string is Some text".
- Use `{...}` around the variable name if necessary:

```
echo ${varname}y stuff
```

This will print out "Some texty stuff".

Environment Variables

- Every program has an "environment", consisting of "environment variables" containing system settings.
- These are inherited when the program starts.
- They can be accessed just like normal shell variables:

```
echo $ENVVAR
```

  - Where `ENVVAR` is just an example, not a real variable.
  - The uppercase is conventional for environment variables.
- To modify them in `bash`, use the `export` builtin command:

```
ENVVAR="new value"
export ENVVAR
```

OR equivalently:

```
export ENVVAR="new value"
```

Common Environment Variables

- `PATH` — a list of directories to search for commands.
- `CLASSPATH` — a list of directories to search for Java classes.
- `LD_LIBRARY_PATH` — a list of directories to search for native shared libraries.
- `USER` — your username.
- `HOSTNAME` — the name of the computer.
- `HOME` — your home directory.
- `PWD` — the current directory.
- `SHELL` — the current shell (e.g. `/bin/bash`).
- `TERM` — the type of terminal being used.
- Many more (often application-specific).

(Note: `CLASSPATH` is Java-specific, and doesn't really have anything to do with UNIX.)

PATH-like Variables

- PATH, CLASSPATH and LD_LIBRARY_PATH all contain a *list* of directories.
- Directories are separated by ":".
- PATH *might* be defined like this:

```
export PATH=/bin:/usr/bin:/opt/bin
```

- We can *append* or *prepend* directories like this:

```
export PATH=${PATH}:/usr/local/bin
```

```
export PATH=/usr/local/bin:${PATH}
```

Here, PATH is set to a new string, which contains the old string.

Pathname Expansion

- A word containing `*`, `?` or `[...]` will undergo "pathname expansion".
- The shell will treat the word as a *pattern*, where:
  - `?` stands for any single character.
  - `*` stands for any sequence of characters (including zero).
  - `[...]` stands for a single character from the given set. For example, `[abcm-z]` stands for "a", "b", "c" or any character from "m" to "z".
  - `[^...]` stands for a single character *not* from the given set.
- The shell replaces the pattern with a list of matching files.
- However, files starting with "." are ignored (unless the pattern itself starts with ".")
- If no files match, the pattern is left unchanged.

Tilde ("~") Expansion

- The symbol ~ is replaced with your home directory.
- For example:

```
ls ~/Desktop
```

  - ~ becomes `/home/username`.
  - The actual command line is "`ls /home/username/Desktop`".
- You can also get to *other people's* home directories.
- Put their username after the ~:

```
ls ~/joe
```

  - This lists the files in `joe`'s home directory, if `joe` exists.
  - You will rarely have permission to do this.

Expansion — Examples

Pattern	Expands to...
`*`	all files (not starting with ".") in the current directory.
`?`	all one-letter files.
`*.c`	all files ending in ".c".
`abc*.o`	all files starting with "abc" and ending with ".o".
`[A-Z]*`	all files starting with an uppercase letter.
`.*`	all files starting with "." (not matched by any of the above patterns).
`.[^.]*`	all files starting with one "." only.
`*/*`	all files in subdirectories of the current directory.
`~/def/*`	all files in `def`, in your home directory.

Quoting (1)

- Backslashes and quotes prevent certain shell actions.
- This is useful when:
  - We want a normal "∗", "&", etc. character.
  - We want to have whitespace *inside* a word.
- Any single character preceeded by a backslash is taken literally

```
echo Some special characters: \&, \*, \\, \"
```

- Everything in single quotes ('. . . ') is taken literally:

```
echo '${PATH} ==' ${PATH}
```

- Double quotes (". . . ") allow variable substitutions and backslashes only:

```
gedit "strange *\"file${var}.txt"
```

If `var` contains "X", `gedit` will open "strange␣*"fileX.txt".

Quoting (2)

- Quotes are often used when assigning variables:

```
varname="Some Text Containing Whitespace"
```

(Without the quotes, the whitespace would make this illegal.)

- Quotes are also used when dealing with strange filenames.

Aliases

- Aliases allow you to create shortcuts for commands, or to redefine commands.
- Compared to shell variables:
  - Aliases also have a name, replaced by a textual value.
  - Aliases only apply to the first word in a command, and there are no $ signs.
- You define an alias like this:

```
alias name='command param1 param2 ...'
```

- Aliases are commonly used to specify default parameters:

```
alias rm='rm -i'
```

  - After this, typing "`rm`" will actually invoke "`rm -i`".
  - `-i` causes `rm` to ask you before deleting a file!

Settings Files

- `bash` reads several files containing lists of shell commands.
- When you log in, bash reads:
  1. `/etc/profile` — a system-wide configuration file.
  2. `~/.profile` (or `~/.bash_profile`, or `~/.bash_login`) — a user-specific configuration file.

  These set up environment variables — `PATH`, `CLASSPATH`, etc.
- When you open a new terminal (*after* you log in):
  - `bash` reads `~/.bashrc` only.
  - This is where you can put alias definitions! (Or any other shell-specific settings.)
- When you log *out*:
  - `bash` reads `~/.bash_logout`.

Make

- Compiling large programs can take a long time.
- When a small change is made, you don't need to recompile the whole program.
- Recompile an object file when:
  - The original .c file changes.
  - Any of the included .h files change.
- The "make" tool helps automate this, and generally makes compiling easier.

Makefiles

- A “makefile” tells “Make” *what* to create, *when*, and *how*.
- Contains a series of “rules”.
- Each rule consists of:

  Target – the file to create/update.
  Prerequisites – the file(s) needed to make it.
  Recipe – the command(s) to make it, indented with a single TAB character (not with spaces).

```
targetfile : prerequisite1 prerequisite2 ...
        command1
        command2
        ...
```

- A makefile has several rules, one after another.
- Makefiles can also have comments (documentation) starting with #.

What Make Does

- To run make:

```
[user@pc]$ make
```

- Make looks for a file called “Makefile” (exactly), and reads it.
- Make looks at the first rule, by default.
  - This is usually fine, but you *can* specify a different target:

    ```
    [user@pc]$ make anothertarget
    ```

- Make asks these questions:
  - Does the target file exist?
  - Is the target newer than all its prerequisites?
- If so, the target is “up-to-date”, and nothing happens.
- If not, make will run the command(s) to (re)create it.
  - Make just *assumes* the commands will create the target file.
  - *You* must provide the correct commands.

What About the Other Rules?

- A prerequisite in one rule might be a target in another (with its own prerequisites).

```
endfile : middlefile1 middlefile2 ...
          commandA

middlefile1 : startfile1 startfile2
          commandB
...
```

- If so, make runs the other rule as well.
  - Does the secondary target ("`middlefile1`" above) exist?
  - Is the secondary targer newer than its own prerequisites?
- Typically, most makefile rules are connected in this way.
  - Make will create `middlefile1` and `endfile`, in that order.
  - Once created, make will update them as needed.

Makefiles and C

- Make is very powerful, but we'll focus on a limited subset of it.
- To create a makefile for a C application, we'll have:
  - One rule (the first rule) to create the executable.
  - One rule to create *each* object file.
- The executable's prerequisites are the object (.o) files.
- Each .o file's prerequisites are:
  - A single .c file with the same name;
  - Any .h files #included by the .c file;
  - Any other .h files #included by those .h files, and so on.
- Why are all the .h files included in the prerequisites?
  - If .h files change, we would like Make to recompile the .c file.
  - The contents of the .h files (particularly macros and constants) will affect the compiled result.
- No rules to create .c or .h files.
  - These files are created by the programmer!

Makefile Example

- Say our C application has these files:

  `main.c` – contains the `main` function.
  `aardvark.c` – contains aardvark-related functions.
  `aardvark.h` – contains aardvark-related declarations.
  `narwal.c` – contains narwal-related functions.
  `narwal.h` – contains narwal-related declarations.

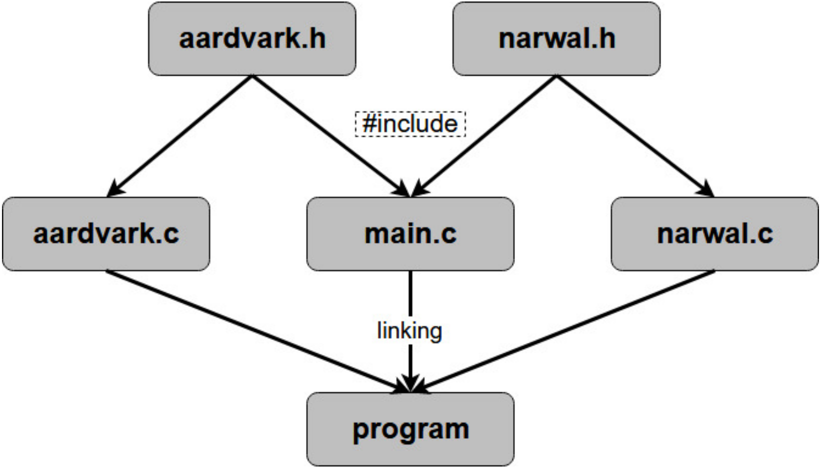
- `main.c` and `aardvark.c` both contain this line:

```
#include "aardvark.h"
```

- `main.c` and `narwal.c` both contain this line:

```
#include "narwal.h"
```

Makefile Diagram



Makefile Example – Rule for the Executable

- First, we write a rule for making the executable:

```
program : main.o aardvark.o narwal.o
        gcc main.o aardvark.o narwal.o -o program
```

- We make "`program`", given `main.o`, `aardvark.o` and `narwal.o`.
  - This is the linking step.
  - At first, these .o files don't exist, but we'll create them too in other makefile rules.
- Make will run the gcc command if:
  - `program` does not exist, or
  - `program` is older than any of the .o files.

Makefile Example – Rules for the Object Files (1)

- Next, we write a rule for each .o file.
  1. Create `main.o`, based on `main.c` and both .h files.
  2. Create `aardvark.o`, based on `aardvark.c`/`.h`.
  3. Create `narwal.o`, based on `narwal.c`/`.h`.
- For example:

```
main.o : main.c aardvark.h narwal.h
        gcc -c main.c -Wall -ansi -pedantic
```

- This is the compiling step ("-c" for compile only; don't link).
- Make will run this command if:
  - `main.o` does not exist, or
  - `main.o` is older than `main.c` or either .h file.
- Notice that the .h files are *not* part of the command.
  - gcc knows about the .h files, because the .c file tells it.

Makefile Example – Rules for the Object Files (2)

- The final two rules, for completeness.

```
aardvark.o : aardvark.c aardvark.h
        gcc -c aardvark.c -Wall -ansi -pedantic
```

```
narwal.o : narwal.c narwal.h
        gcc -c narwal.c -Wall -ansi -pedantic
```

- `aardvark.o` *does not* depend on `narwal.h` (and vice versa).
  - They only #include one .h file each.

Makefile Example – Put Together

- A makefile is a *single* file, so let's see it altogether:

```
program : main.o aardvark.o narwal.o
        gcc main.o aardvark.o narwal.o -o program

main.o : main.c aardvark.h narwal.h
        gcc -c main.c -Wall -ansi -pedantic

aardvark.o : aardvark.c aardvark.h
        gcc -c aardvark.c -Wall -ansi -pedantic

narwal.o : narwal.c narwal.h
        gcc -c narwal.c -Wall -ansi -pedantic
```

- Typing "`make`" will run each of these commands, *if needed*.
- However, we're not quite finished yet!

Cleaning Up

- Makefile rules can do anything, not just compile or link.
- Traditionally there's also a "clean" rule to remove all generated files (object and executable files).

Common definition

```
clean:
	rm -f program main.o aardvark.o narwal.o
```

- To execute this rule, run:

```
[user@pc]$ make clean
```

- This is a bit of a hack. It works because the file "`clean`" is never actually created (hopefully).

Make Variables

- Makefiles have their own variables (a bit like shell variables).
- Why?
  - To avoid repetition.
  - To allow easy configuration changes.

Example

```
CFLAGS = -Wall -pedantic -ansi
...
main.o : main.c aardvark.h narwal.h
        gcc -c main.c $(CFLAGS)
...
```

- We'd do the same for the other two .o rules.
- So, we can modify them in one place.

Traditional Make Variables

- You should use at least the following variables (for the purposes of this unit):
  - `CFLAGS`: Flags for the C compiler, used for each .o file rule.
  - `OBJ`: A list of .o files, used in the executable rule and the clean rule.
- Others to consider:
  - `EXEC`: The executable filename, used in the executable and clean rules.
  - `CC`: The C compiler command. Mostly this is just gcc, but in principle, on other platforms, it can change.

A Better Example Makefile

```
CC = gcc
CFLAGS = -Wall -pedantic -ansi
OBJ = main.o aardvark.o narwal.o
EXEC = program

$(EXEC) : $(OBJ)
        $(CC) $(OBJ) -o $(EXEC)

main.o : main.c aardvark.h narwal.h
        $(CC) -c main.c $(CFLAGS)

... # Similar rules for aardvark.o and narwal.o.

clean:
        rm -f $(EXEC) $(OBJ)
```

Coming Up

- Next week's lecture will introduce you to pointers!
- Make sure you do the tutorial exercises — you will use header files and Makefiles throughout the rest of the unit.