SYS2/CCCP: Program Flow, Pre-processing and Compilation

For Java and Python Programmers

Dr David Griffin

Program Flow

These slides are going to be delivered quite quickly

Most of this is stuff you will have seen in Java

Just pay attention to any differences which are highlighted

Comments

```
// this is a single line comment


/*

This is a block comment

If you're using a really old C compiler, it might not support single line comments and you
have to use block comments on a single line

*/


/* like this */


/** this is a Java documentation comment. In C it's just a regular comment and doesn't
have a special meaning */
```

If-Else

```
if (x == 4) {
  printf("x is 4");
} else if (x == 5) {
  printf("x is 5");
}
else {
  printf("x is not 4 or 5");
}
```

- Works like Java!
- Condition can be any variable or comparison
- If it's a variable, the check is "if variable is 0, it's False, otherwise it's True"
- No special keywords of chaining multiple if-else statements together

Ternary

- Alternative form of if-else which fits on a single line
- Most languages have something similar, but ternaries seem to be used a lot in C
- Rule-of-thumb: Only use a ternary for simple if-else operations
- You can chain ternaries as much as you want, but please don't

(Java has these, but you might not have seen them)

```
a = (condition) ? true_value :
false_value;
```

```
int x = 4;

int y = 5;

int z;

z = (x < y) ? 2 : 3;

// x is less than y, so z == 2
```

Do not do the following:

```
z = (((w == 5) ? 5 : 4) == ((w+x < y) ?
4 : 5)) ? ((y > 4) ? 3 : 1) : ((x > 3) ?
((v < 0) ? 1 : 2) : 3);
```

Switch/Case

- Works like Java
- Switch/Case statements are a fast way of performing a bunch of If-Else's providing that you're evaluating one expression against a list of constants
- If you're checking against sequential integers, the compiler can optimize a switch/case statement to something called a jump table, which runs very quickly
- Note the break statements. If you omit break, it will "fall through". In the example, cases 6 and 7 will do the same thing.
- default specifies the default action, if the condition doesn't evaluate to one of the case's
- If there's no default and no match, switch does nothing

```
int x = 5;

int y = 2;

switch (x + y) {

  case 6:

  case 7:

    // do something for 6+7

    break;

  case 8:

    break;

  default:

    // default action

}
```

For Loop

- Works like Java!
- The header contains an initialiser, a condition check, and an update
- On starting the loop it runs the initialiser
- At the end of each loop it checks the condition
- If the condition is True, it runs the update and starts again
- If the condition is False, the loop ends
- Can also exit the loop using `break;`

```
for (int x = 0; x < 10; x = x + 1) {
   // do this 10 times
}

int y;

for (y = 100; y > 0; y = y - 1) {
// this is generally a bad idea
   y = y - 10;
}
```

While and Do While Loops

- Again, works like Java!
- These loops work using a conditional only
- While checks the conditional first
- Do-While checks the conditional last, so the loop body will execute at least once
- Can use break; to exit the loop

```
int x = 100;
while (x > 0) {
  // do some stuff
  x = x - 1; // update condition
};
```

```
int x = 0;
do {
  // do some stuff
  x = x - 1;
} while (x > 0); // will still run once
```

Functions

```
// prototype
int add(int x, int y);


// implementation
int add(int x, int y) {
  return x + y;
}
```

- Functions in C have two parts
- The Function Prototype, which specifies the name and signature of the function, and promises that there exists an implementation.
- The Function Implementation, which specifies the name, signature and implementation of the function
- All functions which are not main must have a Function Prototype

const Arguments

- The add function shouldn't modify its arguments
- We can declare the arguments to be const and the compiler enforces this
- This still might not seem too useful, but we'll soon see an example where it is *very* useful

```
// prototype
int add(const int x,
const int y);


// implementation
int add(const int x,
const int y) {
  return x + y;
}
```

Header Files

- Function Prototypes are normally stored in Header files
- These files have the .h extension
- Are included using `#include "some_header.h"`
  - Double quotes mean look in current directory, Angle braces standard library
- Header files might also contain constants and some macros, but they do not contain any implementation

simple_math.h

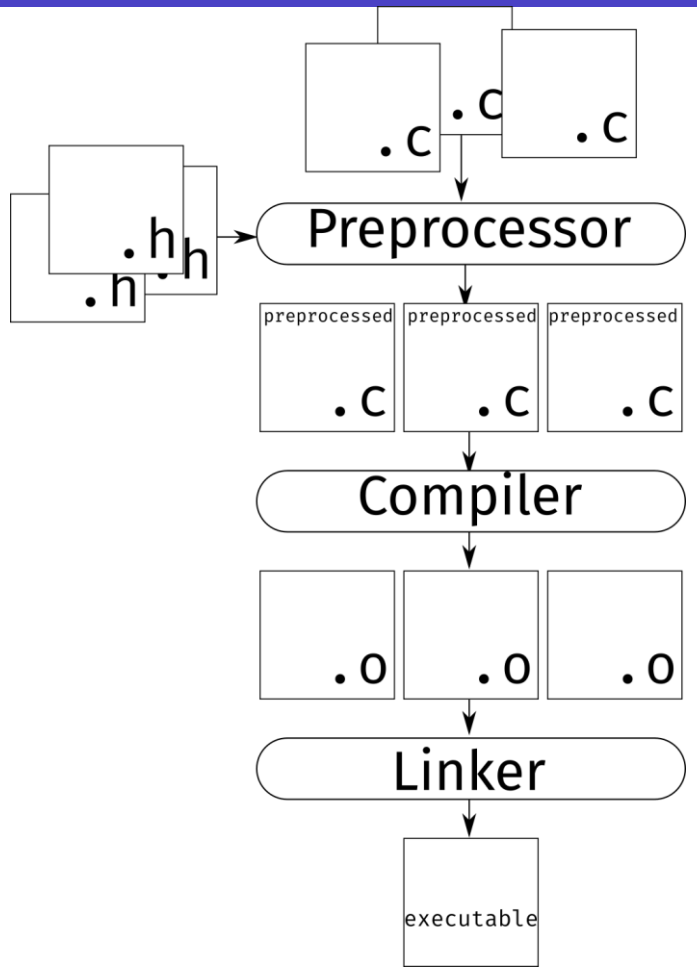
```
int add(int x, int y);

int sub(int x, int y);

int mult(int x, int y)'

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

How does C Compilation Work?



You give the C Compiler Chain a set of C files

The C Pre-processor resolves all Pre-processor directives

- e.g. it looks for files referenced by #include and replaces the directive with the content of that file

The C Compiler compiles all the C files to Object (.o) files

The C Linker combines the .o files into an executable, along with any linked libraries

If this seems arcane and overly complicated, remember that C is about 50 years old

The C Pre-processor

Enables us to use #include to insert header files

The function prototypes in a header file let us use a function, even though the implementation is in another file that might not be compiled yet

#ifdef / #else / #endif allow code to be included / omitted based on environment variables

#define lets you do replacements in your code e.g. “#define X 2” replaces all instances of X with 2

After the pre-processor is done, all the pre-processor directives will be resolved

On Linux: Try running the cpp command on a c file to see the output of the C preprocessor.

Pre-processor Example

```
// Before


#include “simple_math.h”
#ifdef WINDOWS
  PATH_SEP = ‘\\’
#else
  PATH_SEP = ‘/’
#endif
#define CONST 4
int main(){int x; add(x,
CONST); return 0;}
```

```
// After


int add(int x, int y);
int sub(int x, int y);
int mult(int x, int y);
float div(int x, int y);
PATH_SEP = ‘/’
int main(){int x; add(x, 4);
return 0;}
```

The C Compiler

Very simple! Converts pre-processed C files into Object (.o) files

Object files contain the machine code that you can use to build an application

But they're not an application yet

Sometimes library developers will distribute libraries as object files and headers – if the source isn't available, you won't be able to see the implementations, but you can still use the library, as well as being able to skip compiling the library

The C Linker

Takes all the object files you're using, and turns them into an executable

Responsible for enforcing things like "only one main function" and "all functions used are defined somewhere"

Needs to know about all the object files and dynamic libraries you might build against

For example, the linker option `-lfoo` tells it to link the foo library

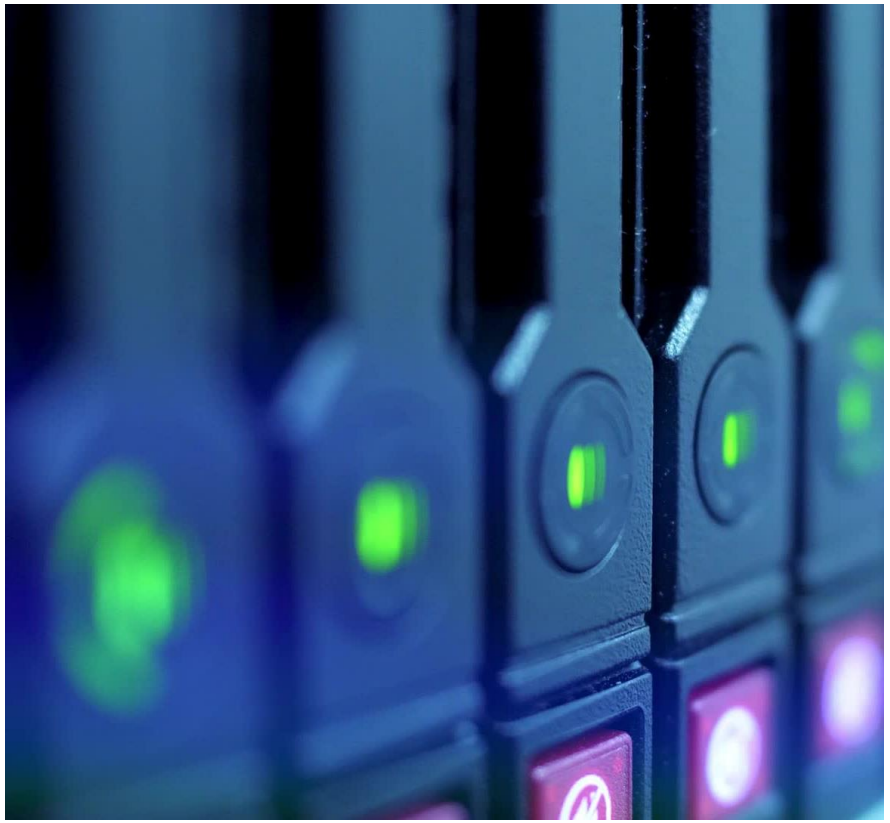
How are all these tools used in big projects?

- There's a whole ecosystem of tools built around the C compiler chain to make it more manageable
- We've already used one: gcc – it handles calling the pre-processor, compiler and linker for you.
- Notable mentions include GNU make and Cmake
- These kinds of tools automate the build process based on a configuration file
  - For example, if you have a GNU make Makefile, you can just run "make" to build your executable
  - Of course, setting up a Makefile can take some time
- IDEs can also be used to manage compiling projects.

C on the Lab Computers under Linux

- Unfortunately, sometimes the IDEs on the lab computers at York might not work!
  - i.e. Most likely, right now.
- This is why the previous practical encouraged you to use a text editor and terminal.
- Terminals are also good for other things, so it's useful to learn them.

GCC – The Gnu C Compiler



- gcc is the C compiler chain used at York
  - Others exist, like msvc (only on Windows) and clang (common on Mac)
- As we've seen, you can compile a C file by giving it to gcc
  - `gcc c_file.c -o executable`
- And then you can run the executable
  - `./executable`
  - (if you didn't specify an output name with the `-o` option, it will default to `a.out`)
- gcc only needs the .c files; it knows where to look for headers

More Compiling/Running from the terminal

If you want to compile more than one C file, just pass in more than one C file	• `gcc file1.c file2.c`
If you want it to include header files from a different directory, use the `-I` option	• `gcc -Iheader_dir custom_headers.c`
If you want to include libraries, use the `-l` option	• `gcc -lm maths_program.c`
If you want to include non-system installed libraries, tell it where to look with the `-L` option	• `gcc -L. -lcustom library_program.c`
If you want extra debugging with GDB, use the -g option	• `gcc -g buggy_program.c`

Debugging with GDB

This is a complex topic which we'll only scratch the surface on

If your program crashes, C will normally not be too helpful about why it crashed

- `Segmentation Fault; Core Dumped`

You can add in debug `printf`'s, but as you have to recompile this takes a while

Instead run the program under gdb

- `gdb buggy_program  # launches a GDB shell with your program loaded`
- `run  # tells gdb to start running the program`

And this will tell you what actually happened

You can do a lot more with gdb, but it can get complex without something to help you

- breakpoints, calling functions, printing variables, inspecting core dumps etc.

GNU Make

- Going back to tools which coordinate C functions, one simple and common one is GNU Make
- GNU Make runs Makefiles, which are files named "Makefile" in the project
- The example Defines two rules: build and clean
- Can run these with `make build` and `make clean`
- Important: The indents must be tab characters, not spaces
  - This can be problematic if your text editor uses spaces instead of tabs
- Uses variables to link in some libraries
- This is just an example! Makefiles can do a lot more than this!

```
LIBS=-lfoo -lbar

build:
	gcc *.c $(LIBS)

clean:
	rm *.o
```

Conclusions

Everyone should know about

The various program flow methods of C	Why Header files are used	How the C compiler toolchain works	What tools exist to manage the complexities of the C compiler toolchain

Now we can move onto some more complicated C stuff, like actual algorithms and IO

Assignment

This assignment is about program flow

And compiling something using multiple files

There's going to be lots of ways of doing that last part. I'd recommend using raw commands, but you could create a makefile if you're feeling adventurous.