Including Files and Declaration Order

Programming Fundamentals

Dr Kevin Chalmers

School of Computing
Edinburgh Napier University
Edinburgh

k.chalmers@napier.ac.uk

Notes

Outline

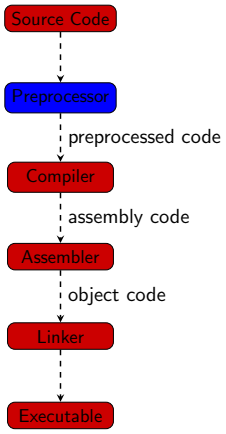
Notes

Outline

Notes

C Preprocessor

- A macro preprocessor
- C preprocessor often referred to as cpp
  - Don't confuse this with C++
- Often a separate program
  - Depends on the implementation you are using
- Invoked by the compiler as the first part of code *translation*



Notes

Macros???

- Short for *"macroinstruction"*
  - Sometimes called macro language
- Generally macro means *"large"*
- Macros are essentially pattern matching and substitution mechanisms
  - Define a rule specifying how an input sequence should be mapped to an output sequence
  - A large block of code can be expanded from a small sequence of characters
- Mapping process is called *"macro expansion"*

Notes

Lexical Preprocessors

- Lowest level of pre-processor
- Does lexical analysis
- Operates on the source text (*"code"*) before parsing (stage where code is checked for correctness)
  - Parsing is the stage where code is checked for correctness
- Performs simple substitution of tokenized character sequences for other tokenized character sequences
- Typically performs
  - Macro substitution
  - Textual / file inclusion
  - Condition compilation or inclusion
- Another type is syntactic preprocessors which operate on syntax trees. We won't be looking at these

Notes

The C Preprocessor

- Looks for lines beginning with # as *directives*
  - A directive is an instruction to the preprocessor
- The C preprocessor actually knows nothing about the underlying C language - *it simply substitutes text segments as instructed*
- As such the C preprocessor can be used by other languages
  - e.g. JavaScript
- Effectively, the C preprocessor just processes source files

Notes

Questions?

Notes

Outline

Notes

Phases of Preprocessing

- The phases of the C preprocessor are defined in the C standard
- The first four (of eight) phases of translation are:

  Trigraph replacement replaces sequences of 3 characters with their representative characters (don't worry about this)

  Line splicing sources lines continued with escaped newline sequences are spliced to form logical lines (don't worry about this)

  Tokenization break the lines into preprocessing tokens and whitespace (replace comments with whitespace)

  Macro expansion & directive handling :

  1. Preprocessing of directive lines
  2. File inclusion (what we are interested in next)
  3. Conditional compilation

Notes

File Inclusion

- Any line like:
  - `#include "filename"`
  - `#include <filename>`
- is replaced by the contents of *filename*

  `"filename"` search for the file from location of source program, e.g. the same directory as the source file

  `<filename>` search standard compiler include paths
- For example, we include the C standard library using the line
  - `#include <stdio.h>`
  - Search is undertaken in standard compiler include paths

Notes

Example

File Inclusion Example

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello Napier!\n");
    return 0;
}
```

- Preprocessor replaces the line `#include <stdio.h>` with the text in the file `stdio.h`
- `stdio.h` defines the `printf` function, allowing us to use it in our code
- `stdio.h` also includes other function definitions that we can further use

Notes

Conditional Compilation

- Conditional compilation allows us to only compile certain sections of code based on values defined (in code or by the compiler)
- Basically, conditional compilation is a collection of if-else directives
- For example, we have
  - `#if`
  - `#ifdef`
  - `#ifndef`
  - `#else`
  - `#elif`
  - `#endif`
- These are very useful for using OS specific features in our code. For example
  - `#ifdef _WIN32`
  - Code following this statement is only compiled when the OS is Windows

Notes

Examples

Using the Preprocessor to Determine a Debug Build

```
#ifdef DEBUG
    printf("Debug mode\n");
#endif
```

Using the Preprocessor to Determine OS

```
#if defined __APPLE__
    // do Mac OSX specific code
#elif defined _linux_
    // do Linux specific code
#elif defined _WIN32
    // do Windows specific code
#endif
```

Notes

Macro Substitution

- Allows us to define and undefine values
  - `#define`
  - `#undef`
- Two types

  Object like `#define` *identifier value*

  Function like `#define` *identifier*(*params*) *replacement*

Notes

Object-like Substitution

- `#define` *`identifier value`*
- Originally used to create symbolic names for constants. e.g.
  - `#define PI 3.14159`
  - `#define EULER 2.71828`
  - `#define NUMBER_OF_STUDENTS 120`
- This was useful instead of hard-coding numbers throughout code
- Standard practice is now to define `const` values, particularly in C++. e.g.
  - `const double PI = 3.14159;`
  - `const double EULER = 2.71828;`
  - `const int NUMBER_OF_STUDENTS = 120;`
- This provides more type checking by the compiler.

Notes

Function-like Substitution

- `#define` *`identifier(params) replacement`*
- Allows us to define function "macros" that are inserted into code. e.g.

  Area circle `#define AREA_CIRCLE(r)(PI * (r) * (r))`
  Radians to degrees `#define RADTODEG(x)((x) * 57.29578)`
- Means we don't have to hard code in calculations
- Also means we don't make a function call
  - Function calls cost cycles
  - `inline` functions get round this
- Macros get round typing problems when doing calculations which functions cannot
- C++ templates get round this limitation of C (not covered in the module)

Notes

Questions?

Notes

Outline

Notes

Header Files and Libraries

- We looked at the preprocessor so we could discuss the important part of this unit - *header files*
- We have used header files since our first program
  - `#include <stdio.h>`
  - Includes the code from the `stdio.h` header file
- Header files contains a collection of already written, reusable code
  - ... and (hopefully) well tested
- `stdio.h` includes standard C language (statements and declarations)
- Using headers enables us to separate and organise our code
  - This becomes a *big thing* when you start building large applications

Notes

Example (part 1)

- On thing we need to ensure is that a header file isn't included more than once
- To do this, we require header guards

Header Guard

```
#ifndef HELLO_GUARD
#define HELLO_GUARD
// hello.h header content
#include <stdio.h>

void hello_world()
{
    printf("Hello World!\n"
        );
}
#endif
```

Using #pragma once

```
#pragma once

// hello.h header content
#include <stdio.h>

void hello_world()
{
    printf("Hello World!\n"
        );
}
```

Notes

Example (part 2)

- To use our header we do the following:

Including Local Headers

```
#include "hello.h"

int main(int argc, char **argv)
{
    hello_world();
}
```

- Notice the use of quotation marks - we are searching for the header in the same directory as the source file

Notes

Example (part 3)

- When we run this code through the pre-processor, what do we get?

Preprocessed Code

```
#include <stdio.h>

void hello_world()
{
    printf("Hello World\n");
}

int main(int argc, char **argv)
{
    hello_world();
}
```

- *caveat* - we get more than this. For example `#include <stdio.h>` is also replaced.

Notes

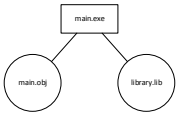
Compiling Multiple Files

- With headers we can now split code between multiple files using headers
- **Aim** - to break up our source code into simple, sensible, logical blocks
- Ideally we want to simplify conceptual model, aid understanding, and support reuse
- Role of Integrated Development Environment (IDE) & make files:
  - The IDE manages the creation of make files for us (and building and running applications)
  - With make files we are doing manually what the IDE does for us
- Basically, to compile multiple files we just list them after the compile command. For example
  - `cl <file1.c> <file2.c> <file3.c>`

Notes

Header Summary

- Headers can be thought of as *bridges* between different code (.c) files
- Compilation process (particularly the C preprocessor and the linker) and other tools do all the hard work for us
- We just need each code *unit* to know enough about the code it calls so the linker can join the parts into a whole



Notes

Creating and Linking Libraries

- So ...
- Our program isn't necessarily a monolithic tangle of code
- Can be broken up
- Can be made manageable
- Can be made reusable
- The key concept is the idea of code **libraries**

Notes

Libraries

- A library is not executable - *it has no `main` function*
- Is linked to our application (which does have a `main`) to build an executable
- Affects the build process
  - Compile-only the source code to build only object files
  - Use other program (e.g. `lib` with Microsoft, `ar` with GNU) to produce library file from object file(s)

Notes

Using Libraries

- We need header files (`.h`) to create a *bridge* between the main application and the library.
  - The header *declares* (but does not *define*) the functionality needed
- We need library files (`.lib` or `.o`) to contain the implementation
  - Pre-compiled *definition* of the functionality
- Understanding the difference between declaration and definition is important when developing software:

  declaration stating that some code will exist (e.g. a function, `struct` or `class`)

  definition actually specifying the internal parts of code declared

Notes

Questions?

Notes

Outline

Notes

File IO

- One of the fundamental parts of computation
  - $input \rightarrow process \rightarrow output$
  - $input \rightarrow transformation \rightarrow output$
- We can't just have user input all the time
- We need to read and write data to other places...
- ... especially files
- C treats many things as a *file* (including the standard input) - this is *fundamental metaphor* when working in some operating systems

Notes

File IO Using Library Functions

- C provides nearly all of its input-output functionality in the Standard IO library
  - `#include <stdio.h>`
- So far we have focused on input-output using the Standard Input (`stdin`)
- We did however touch on file input when reading in from the command line
- To read from a file we typically use File Read
  - `fread(...)`
- To write to a file we typically use File Print Formatted
  - `fprintf(...)`
- So everything should be simple!
- Well, not quite...

Notes

It's Complicated...

- When working with files we need to first open the file
- This means defining *how* (which mode) to open the file. Some options are:

read	opens the file for reading from
write	opens the file for writing to
append	opens the file for writing to the end
text	opens the file as text data (e.g. character data)
binary	opens the file as binary data (e.g. could be anything)

- We also need to remember to close the file
  - If you don't close the file, any changes can be lost due to buffering
- We might also have to seek to a particular location in the file
  - For example we might want the 42nd student record in the file
- We also need to worry about allocating memory to store read in data
- In fact, most of `stdio.h` consists of declarations for file input-output

Notes

Reading Binary Files

- Reading a binary file involves a number of stages
  1. Open the file (remember - binary read mode)
  2. Allocate memory to store the contents of the file
     - How much memory do we need?
     - File structure needs to be known
     - For example, the first value could store the number of records in the file (this is what we do in the workbook - **but this is not standard**)
  3. Once you know how much data you are dealing with, read the values into allocated memory using `fread`
     - `fread(`*`data, type size, number of values, file`*`)`

data	location in memory to read file into
type size	size of individual data type we are reading (e.g. `int` is typically 4)
number of values	number of values of the type we are reading in
file	file we are reading from
  4. Close the file using `fclose`

Notes

Example

Reading from Binary File

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

int readfile(int **data)
{
    FILE *file = fopen("numbers.dat", "rb");
    int size;
    fread(&size, sizeof(int), 1, file);
    *data = (int*)malloc(sizeof(int) * size);
    fread(*data, sizeof(int), size, file);
    fclose(file);
    return size;
}

int main(int argc, char **arv)
{
    // Calls readfile
}
```

Notes

Writing Files

- A bit simpler that reading files
- If binary then still have to deal with size of the data types (e.g. `struct`) being written
- Same basic stages
  1. Open file in correct mode
  2. Loop through values
  3. For each value output to file
  4. Close the file

Notes

<stdio.h>

- fopen()
- fflush()
- fclose()
- fread()
- fwrite()
- fgetc()
- fgets()
- fputc()
- fputs()
- getchar()
- putchar()
- puts()
- scanf()
- fscanf()
- fprintf()
- ftell()
- fgetpos()
- fseek()
- fsetpos()
- rewind()
- remove()
- rename()
- tmpfile()
- tmpnam()

Notes

Questions?

Notes

Outline

Notes

Summary

- You should now have a clearer idea of what the C preprocessor is and its purpose in the compilation pipeline
- You should also understand what a header file is and why we need them
- You should now have a better understanding of how we can go about organising our programs across multiple files
- You should have some idea of how we go about making, using, and exploiting libraries
- We have also covered file IO in C - *this is an area you should practice as it is pretty standard across programming languages*

Notes

To do...

1. **Remember** - do the end of unit exercises. Don't ignore these. They help understanding.
   - Some of you are slipping in the practical material already. The lab time is there for you to get help. *You need to work outside the timetabled classes.*
2. Workbook - applied examples of working with header files, libraries, and file I/O.
3. Tutorial
4. Next lecture - will cover call conventions. In particular - how do we pass values into functions (pointer and reference time!).

Notes

Notes