THE UNIVERSITY OF SYDNEY

INFO1910 S2 2023 Week 5 Tutorial

Compilation

The Tao gave birth to machine language.
Machine language gave birth to the assembler.
The assembler gave birth to the compiler.
Now there are ten thousand languages.
Each language has its purpose, however humble.
Each language expresses the Yin and Yang of software.
Each language has its place within the Tao.
But do not program in COBOL if you can avoid it.
- The Tao of Programming

The Compilation Pipeline

We'll be starting this tutorial with a more in depth view of how compilation works in C, broken down into four stages, it can be broken into many more stages, and each stage itself is quite complex, but we will avoid such complexities for now.

These stages are the preprocessor, compilation, assembly and linking, and each is essential for building our C program. Understanding each of these stages will also improve the performance, readability and structure of our code.

The Preprocessor

You have already encountered the pre-processor before; it is a separate language that performs string replacements throughout C files. While a language, it is not Turing complete, and hence there exist problems that can be solved by C that cannot be solved with just the pre-processor. None the less, it is a useful tool for the readability and portability of your program. The preprocessor iteratively scans the C file and performs string manipulations on identifiers, this scanning and manipulation only stops once there are no longer any identifiers to manipulate.

The preprocessor itself performs macro substitution, conditional compilation and inclusion of named files. Lines beginning with # demark preprocessor commands. This is strictly independent of the rest of the C language, and does not appear elsewhere.

The preprocessor also strips comments from the code, which are demarked by the digraph symbol `//` and terminated by the newline, or by the digraph symbol `/*` and ended by the digraph symbol `*/`. The preprocessor replaces all such bounded blocks with a single space character. Note that if these digraphs are encapsulated within quotes (such as in a C string or character), then they are ignored.

Lines that end with the \ character are folded; that is the newline character is deleted along with the backslash character.

So the following statement

```
#include <std\
i\
o.h>
```

Would be folded into:

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

A C program can be compiled using only the pre-processor using the `-E` flag with your compiler of choice. This is then printed to standard output.

Define

Our first preprocessor operator we will encounter is `define`,

This operator takes the form:

```
#define identifier token-sequence
```

We can also undefine a previously defined identifier to forget its definition, this can be done with the `undef` operator.

```
#undef identifier
```

The preprocessor scans the file for instances of the identifier and then replaces them with the token sequence. These token sequences can take 'arguments', as this is strict string manipulation there exist no typing rules within the preprocessor.

Examples include:

```
#define PI 3.14
#define G 9.8

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

When each of these macros is expanded it is either painted blue or red, blue macros will not expand any further, red macros can be rescanned and expanded further. This allows macro nesting and limited recursive expansion.

Perhaps the only extant use of the ternary operator: `(condition)?(true):(false);` is for use within a macro, where the one line if statement

The `-D<identifier>` and `-D<identifier>=<token sequence>` compiler flags can be used to perform define operations on a C file at compile time.

Question 1: Macros Are Not Functions

What is the difference between these two programs, what does this suggest about the dangers of using macros?

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

int main()
{
    int i = 0;
    while (i < 10)
    {
        printf("%d\n", SQUARE(x++));
    }
}

int SQUARE(int x)
{
    return (x * x)
}

int main()
{
    int i = 0;
    while (i < 10)
    {
        printf("%d\n", SQUARE(x++));
    }
}
```

Include

Our next macro operator, is `include`. This includes another C file in the current file. Local files may be distinguished from library files by the encapsulation of the file name.

```c
#include <library.h>
#include "local_file.h"
```

It's a common feature of C programs to forward declare all functions at the start of the file, along with placing macros, includes and other useful features.

Often these are moved to a header file ending with the `.h` extension, then included by the source file as a way of managing these ubiquitous components of files. It also provides reading material wherein just the headers and their comments may be read rather than traversing the entire source file looking at the implementations of the function.

Header files in other locations can be included with the `-I<path>` flag, where the path specifies the path to the directory containing the header files.

Question 2: Headers

Write a `hello.c` and a `hello.h` file with a clear demarcation between the function declaration and the function definition. Do not include a main function in either of these files.

Question 3: Implementation and Inclusivity

- Suggest a potential implementation for the behaviour of the `include` operator.
- Consider the case where we have two C files, each of which includes the other. What happens?

Conditional Compilation

In order to resolve the issues presented in the previous problem, we introduce the `ifdef` and `ifndef` operators.

These operators are bound by the `else` and `endif` operators and collapse entire blocks of code when their condition is not met.

In the example below, if the identifier HELLO exists, then the first block is executed, otherwise the second block is executed.

```
#ifdef HELLO

    printf("Hello");

#else

    printf("Goodbye");

#endif
```

This is an incredibly powerful syntax where we can now use define statements to allow a single file to compile against multiple conditions, for example when different operating systems require different C code. Rather than writing two different programs, conditional compilation allows one program to compile differently on different systems.

Question 4: Cond Comp

- Propose a resolution to the circular dependency issue from the previous question by using conditional compilation.
- Use the example of conditional compilation from above to write a C program and a makefile with separate `make hello` and `make goodbye` rules that print hello and goodbye respectively.

Compilation

Compilation is the replacement of the C code with a limited form of assembly. Each function is compiled internally and references to external variables, jump offsets and functions are left as symbols.

The output of the compilation stage can be viewed as a `.s` file using the `-S` flag. This is useful for viewing (and modifying!) human readable assembly code.

Question 5: Flip the Jump

Write a program that accepts a single integer as a command line argument and prints whether the input was less than or greater than 42.

Compile the file with the `-S` flag and modify your jumps (they work just like goto statements) to flip the direction of the if statement. Compile the `.s` file and execute it to ensure that your changes worked.

Assembly

In the assembly stage the symbols left by the compilation step will be matched to other compiled functions or defined variables and replaced with the relevant addresses. The jump offsets within functions are also calculated and put in place. We will discuss this step a bit more in future weeks.

At this stage we are given an object file with all internal symbols connected, but external symbols are still dangling. This includes functions that have been forward declared but not yet defined (hence why we include header files but not source files!), and variables marked with the `extern` keyword.

A program can be assembled using the `-c` flag.

Question 6: Extern

Compile each of these files to object files, then compile the object files together and run the code.

`extern.c`

```c
float pi = 3.14
```

`pi_printer.c`

```c
#include <stdio.h>
int main()
{
    extern float pi;
    float foo = 1.337;
    printf('%f %f\n', pi, foo);
```

```
    return 0;
}
```

Compilation sequence should be:

```
gcc -c extern.c
gcc -c pi_printer.c
gcc extern.o pi_printer.o -o pi_printer.out
```

- How does this differ from using the preprocessor to set a macro for PI?
- Why do we need to specify the type of pi in `pi_printer.c`?
- Compile and look in the `pi_printer.s` file for the pi symbol, how does it differ from the `foo` variable?

Question 7: Flipping Twice

Take your flipped jump compiled file and assemble it, try to flip that jump back now. You may find the `:%!xxd` and `:%!xxd -r` commands in vim very useful for this task. It might be worth knowing that the different jump instructions are defined such that the finding complement to the jump condition is equivalent to flipping the last bit of the conditional jump instruction.

Linking

Linking occurs by connecting any dangling symbols. Unlike the assembly stage, the linking stage links different assembled files together, connecting any remaining symbols.

An example is linking the `stdio.h` file, the header file itself only contains function declarations, while elsewhere is a compiled object file containing the definitions. While the previous stages have reserved memory using the function signatures within the header file, the linker now replaces the relevant symols with the addresses of the functions provided by another object file.

The output of the linking stage can be saved to a binary file using the `-o` flag.

Question 8: A Manual Pipeline

Write a makefile that manually performs each stage of the compilation process using the output of the previous stage `make all` should run each stage as a dependency of the previous one.

Save the output of all the intermediary stages to a `build` directory. You might want to make this a dependency so that it is automatically constructed when you run make.

Question 9: Compiling Together

Write three C files; `hello.h` containing the function signature for a function `void hello();`, the next `hello.c` containing the definition of the `hello` function that prints hello world, and a second source file containing a main function that calls hello.

- Compile both of these files together,
- Compile both files to object files, then compile the object files together, what advantages does this pose over the previous method.

Makefiles Revisited

Now that we've significantly increased the complexity of our compiling, we should also acquire a more versatile build script for this; in the form of improving our makefiles and directory structure.

Of course now that we're building all these object files, it seems a bit messy to leave them lying around; we really should put them all in a build directory (handily titled build). Typically creating this directory will also be the work of the makefile.

Having moved our object files out of the way, it's now time to do the same with our source files to a `src` directory and our header files to a `lib` directory. We can include our headers for forward declaration purposes with the `-Ilib` flag, and our source files are all being built to object files anyway.

We've now separated our compilation into two steps, and so we need two separate rules. The first, to build the object files should be very similar to the previous compilation rules you have used, however instead of `-o ${TARGET}$` we're now building to the object file in question.

```
%.o : %.c
    ${CC} ${CFLAGS} ${LIBS} $^ -c
```

Here the `$^` digraph is a shorthand for the name of the dependencies while `%` is used for string matching, so in the above rule, `hello.o` would be used to pattern match to `hello.c`, which would then replace the digraph `$^`, leaving our final command:

```
    ${CC} ${CFLAGS} ${LIBS} hello.c -c
```

Next we want to build our executable, the dependencies for this are all of our object files. Given we have a standard pattern for our object file format, we can avoid hard coding all the names of these files.

Makefiles can make use of wildcards and substitutions when setting variables, this makes it very easy to collect all our source files and all of our object files rather than needing to specify them manually.

```
SRCFILES := $(wildcard *.c)
OBJFILES := $(patsubst %.c,%.o, ${SRCFILES})
```

So our rule to build our target becomes:

```
${TARGET} : ${OBJFILES}
    ${CC} ${CFLAGS} ${LIBS} -o $@ $^
```

Where the digraph `$@` is replaced with the name of the rule, which here is our target, and `$^` is expanded as before.

Lastly, sometimes you may have problems with the names of your files and the names of your makefile rules, for example if you wanted `make build` but already had a `build` directory. For this we have the `.PHONY` rule, which declares that certain rule names will never be interpreted as file names.

So imagine that there exists a file named `clean` in your directory, which conflicts with your `make clean` rule.

```
.PHONY: clean
clean:
    rm -rf ${BUILDDIR}$/*.o
```

Would indicate that the clean rule is not associated with the file `clean`.

Question 10: Setup

Write a makefile rule that creates a build directory if it doesn't already exist, consider the similarities between this and the dependencies in other rules and think about where you should set the build directory as a dependency itself.

Question 11: Enterprise Hello World

Let's get a minimum working example going; we're going to use five different files for our hello world function.

The first is a `hello.c` file containing a function `hello`. This function should return a random greeting string from a collection of such greetings. The function should be forward declared within the `hello.h` file. Consider very carefully how such strings are declared and stored. You will want to use the `rand` function for this. You may consider modifying `RAND_MAX` in some manner, but you should probably return it to its original value after use. Alternatively the modulus operator is very useful for setting bounds quickly (neither of these approaches provide particularly uniform random distributions).

Next we need a greeter in `greeter.c` and `greeter.h`, this should contain a greeter function that should take the random greeting from the hello function and print it to standard output.

Lastly, we need a main function to call the greeter. This should exist within a `main.c` file.

All C files should exist within your `src` directory, all header files should exist within your `lib` directory and your topmost directory should contain both of these directories, your `build` directory and your makefile.

Try to make this makefile as general as possible for this particular directory layout.

Multi Dimensional Arrays

Question 12: $N = 1$

- Write a collection of helper macros that assist in mapping a one dimensional array to a two dimensional array.
- Do the same for three dimensions.
- How does mapping an N dimensional array to a one dimensional array assist in passing the array to other functions?

Question 13: Ragged Arrays

A ragged array is a multi-dimensional array where not all elements are of the same length. Do your macros from before help you with ragged arrays? What is a better representation for such ragged arrays?

Implement your own ragged arrays and pass them to a function to print all the values stored within. You might wish to consider defining special regions of memory in your array to more easily handle passing the length of the array to the function.

Question 14: Trigraphs

Compile the following code both with and without the `-trigraph` flag. Explain what is happening here.

```c
#include <stdio.h>

int main()
{

    int x = 0;

    // What will be the value of x??????????????/
     x++;

     printf("%d\n", x);

    return 0;
}
```

Question 15: Backtrack

Go back through all the C code you have previously written for this unit and convert your programs and makefiles to the new format, you should find that for the vast majority of cases you will now have one very generic makefile that just needs to have its target changed between programs.

Question 16: (Extension) Your Own Basic Preprocessor

Write your own basic implementation of the preprocessor in C, it read a C file and perform define and include operations. You do not need to do more complex macros, `ifdef` or `ifndef` (but you are welcome to if you wish!), however you should consider how rescanning works. You should print your pre-processed C file to standard output.

You will need to look into how to read from a file in C if you wish to complete this extension problem. You may assume (erroneously) that no line will ever be longer than 80 characters.

Alternatively, attempt to write your preprocessor in bash, this will make file reading much easier, however you will likely want to look into either `awk` or `sed` along with regular expressions or regex for handling the string manipulation.

Question 17: (Extension) Preprocessor Loops

There exists an additional digraph operator in the preprocessor, `##`. This operator concatenates two tokens, but halts further expansion on the result, even if it would produce new tokens. Its use is best described as 'arcane'. To be specific, if the token sequence contains the digraph `##` then after expansion, the digraph and all preceding and trailing whitespace is deleted, causing the concatenation of the adjacent tokens. I would suggest looking first at "The C Programming Language" appendix A, section 12 for more details about this operator. It may also be worth delving into the ANSI C standard.

- Use this in conjunction with the rest of your knowledge of the preprocessor and red/blue painting to construct a preprocessor for loop that can count down from ten using the expansion rules.
- What is interesting about the construction of such loops, and what does this indicate about the limitations of the preprocessor as a language?
- Suggest why such a limitation is necessary; what problems would arise within the preprocessor if such limitations did not exist?

Question 18: (Extension) Lunacy

The preprocessor is powerful and silly. Go and have a look at some of the entries in the international obfuscated C code contest. Then endevour to never write C code in this particular style, except for fun.

Using the C preprocessor rewrite one of your previous C programs to be something interesting, this could be poetry, ascii art, an english description of its own function etc. Include a makefile and provide a link to your artwork for others to run and admire. You should try to preserve the original functionality of the code.

An example produced by Simon Koch has been reproduced below.

lunacy.c

```
#include "lunacy.h"
#include iostuff
#include stringstuff

num p_str takes strlit str only
begin
    ret printf takes stringify(%s\n) and str only go
end

num main takes no arguments
begin
    // make sure str is editable/mutable
    alph str array is stringify(first:second:third) go
    strlit delim is stringify(:) go

    // prints "first"
    p_str takes strtok takes str and delim only only go

    // prints "second"
    p_str takes strtok takes nothing and delim only only go

    // prints "third"
    p_str takes strtok takes nothing and delim only only go

    // stores [5, 4, 3] in numbers
    num numbers array is begin five and four and three and two end go
    // prints numbers[1] = 4
    printf takes stringify(%d\n) and numbers at one memory lane only go

    ret numerical nothing go
end
```

lunacy.h

```
#define begin {
#define end }
#define num int
#define strlit char*
#define p_str(_str) printf("%s\n", _str)
#define ret return
#define array []
#define alph char
#define no void
#define arguments )
#define go ;
#define takes (
#define and ,
#define only )
#define after (
#define before )
#define times *
#define plus +
#define follow *
#define at [
#define memory ]
#define lane
#define unless(condition) if(! condition )
#define have (
#define equal ==
#define is =
#define nothing NULL
#define numerical (int)
#define iostuff <stdio.h>
#define stringstuff <string.h>
#define stringify(_str) #_str
#define one 1
#define two 2
#define three 3
#define four 4
#define five 5

#include iostuff
#include stringstuff
```