INFO1910 Week 5 Tutorial

Compilation

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++));
    }
}
```

```c
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.

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 3: 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 4: 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 5: Extern

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

`extern.c`

```
float pi = 3.14
```

`pi_printer.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?

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 6: 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.