CSSE2310 Week 3 – Preprocessor

1 Preprocessor

I recommend reading the GNU page on macros. I'll cover some of the basics here, but the GNU documentation is much more thorough and very digestible.

1.1 Object-like macros

We can use `#define` to create *object-like* macros. These take the form

```
#define <identifier> <fragment>
```

Wherever the identifier appears in the source file, it is replaced with the code fragment.

Usually, object-like macros are used to avoid magic numbers. For example

```
#define MAX_SIZE 1024
#define MIN_SIZE 0
```

Now `MAX_SIZE` or `MIN_SIZE` can be used rather than typing `1024` or `0` everywhere. This makes it much easier to read, because you don't have to guess what the numbers mean.

Which do you think is easier to read?

```
if (size > 1024) {
    /* do something */
}
```

Or

```
if (size > MAX_SIZE) {
    /* do something */
}
```

Another benefit is that if you want to change the value, you don't have to find every place it was used, you can just change it once in the macro definition.

These kinds of macros are usually put at the top of your source code or header.

1.2 Function-like macros

Function-like macros look a bit like functions. Just like functions, they take some arguments. But they are much dumber than functions and you have to be very careful that, when they are expanded, they don't cause any weird issues.

Parentheses must be used extensively in macros to ensure they work as expected. Operator precedence and associativity can have unexpected consequences if parentheses are not used properly.

1.2.1 Always put parens around the macro definition

```
#define SUM(X, Y) X + Y
...
int c = 5 * SUM(a, b)
==> 5 * a + b
```

To fix this, use `SUM(X, Y) (X + Y)`.

1.2.2 Always put parens around the arguments

```
#define DIV(X, Y) (X / Y)
...
int c = DIV(a + 2, b)
==> a + 2 / b
```

To fix this, use `DIV(X, Y) ((X) / (Y))`.

1.3 Multiple evaluation

If the argument to a macro is used multiple times, then it will be evaluated multiple times. For example, consider this macro

```
#define SQUARE(X) ((X) * (X))
```

Imagine that the function `sequence` returns `0` on the first call, then `1`, `2`, and so on. Then `SQUARE(sequence())`, will expand to `((sequence()) * (sequence()))`, which might be `((1) * (2))`, which is clearly not what we want.

Another thing to watch out for is if the argument has *side effects* (that is, has an effect on the state of the program or the environment). For example, if we call `SQUARE(readInt(file))` where `readInt` is some function that reads from the file and returns a number, then `readInt` will get called twice.

1.4 Include guards

Larger projects (such as later assignments) will often benefit by being split up into separate files. Functions, types, and variables that are closely related to eachother will be put into a pair of files: a source file (with extension `.c`) and a *header* file (with extension `.h`).

The *public* interface (the function prototypes, structs, etc that are meant to be visible to other modules) are placed in the header file and the function definitions into the source file.

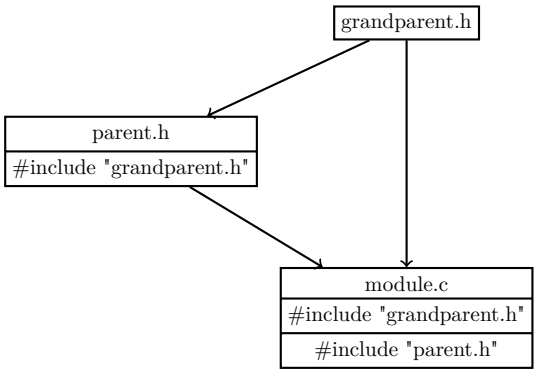
The source file and any other files that need to use the functions or types from the header must include the header file, like so

```
#include "otherFile.h"
```

This `#include` literally pastes the contents of the header file `otherFile.h` into the current file. Now that the header has been included, the file knows about those function prototypes and so on that are in the header.

1.5 The problem

The problem is that sometimes we include the same header file *twice*. Consider this diagram, where the arrows show includes.



The preprocessor pastes `grandparent.h` into `module.c` twice: once because `module.c` includes it directly and once indirectly through `parent.h`.

To stop this from happening, we use a clever trick: we wrap our header files in a preprocessor directive that stops them from being copied more than once.

```
#ifndef GRANDPARENT_H
#define GRANDPARENT_H
/* ... contents of grandparent.h ... */
#endif
```

Everything between the `ifndef` ("if not defined") and the `endif` directives is ignored if the symbol `GRANDPARENT_H` is defined.

The first time the preprocessor reads the file, the symbol won't yet exist and everything will be included as normal, but when the preprocessor reads the file for the second time, the `#define` will have created `GRANDPARENT_H` and so the `ifndef` check will fail and the preprocessor will ignore everything.

These are called **include guards** and it's good practice to always use them for all your headers.