More about Macros and Header Files

coursera.org/learn/writing-running-fixing-code/supplement/BWulO/more-about-macros-and-header-files

More about Macros

The standard header file limits.h contains constants specifically for portability. These constants describe the maximum and minimum value of various types on the current platform. For example, if a program needs to know the maximum and minimum values for an **int,** it can use INT_MAX and INT_MIN respectively. On platforms where an **int** is 32 bits, these would be defined like this:

```
#define INT_MAX  2147483647
#define INT_MIN -2147483648
```

On a platform with a different size for the **int** type, these would be defined to whatever value is appropriate for the size of the **int** on that platform.

Macros can also take arguments, however, these arguments behave differently from function arguments. Recall that function calls are evaluated when the program runs, and the values of the arguments are copied into the function's newly created frame. Macros are expanded by the preprocessor (while the program is being compiled, before it has even started running), and the arguments are just expanded textually. In fact, the macro arguments do not have any declared types, and do not even need to be valid C expressions —only the text resulting from the expansion needs to be valid (and well typed).

We could (though as we shall see shortly, we shouldn't) define a SQUARE macro as follows:

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

The preprocessor would then expand SQUARE(3) to 3 * 3, or SQUARE(45.9) to 45.9 * 45.9. Note that here, we are using the fact that macro arguments do not have types to pass it an **int** in the first case and a **double** in the second case. However, what happens if we attempt SQUARE(z-y)? If this were a function, we would evaluate z-y to a value and copy it into the stack frame for a call to SQUARE, however, this is a macro expansion, so the preprocessor works only with text. It expands the macro by replacing x in the macro definition with the text z-y, resulting in z-y * z-y. Note that this will compute z- (y*z) -y, which is not z-y squared.

We could improve on this macro by defining it like this:

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

Now, SQUARE(z-y) will expand to ((z-y) * (z-y)), giving the correct result independent of the arguments and location of the macro. We can still run into problems if the macro argument has side effects. For example if we type SQUARE(f(i)), then the function f will be called twice. If f prints something, it will be printed twice, once for each call to f in the macro expansion.

The SQUARE macro is a bit contrived—we would be better to write the multiplication down where we need it—but highlights the nature (and dangers) of textual expansion of macros. They are quite powerful and can be used for some rather complex things, but you will not need them for most things you write in this specialization.

More about Header Files

Header files may also contain type declarations. For example, stdio.h contains a type declaration for a FILE type, which is used by a variety of functions which manipulate files. The functions also have their prototypes in stdio.h, and we will discuss them later when we learn about accessing files and reading input.

Another example of type declarations in standard header files are the integer types in stdint.h. As mentioned previously, integers come in different sizes, and the size of an **int** varies from platform to platform. Often programmers do not care too much about the size of an int, but sometimes using a specifically sized **int** is important. stdint.h defines types such as int32_t (which is guaranteed to be a 32-bit signed **int** on any platform), or uint64_t (which is always a 64-bit **unsigned int**).

With all of that in mind, we can revisit our Hello World program from earlier:

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

int main (void) {
  printf ("Hello World\n");
  return EXIT_SUCCESS;
}
```

The preprocessor would take this code and basically transform it into this:

```c
int printf(const char *, ...);

int main (void) {
  printf ("Hello World\n");
  return 0;

}
```

In actuality, the result of the preprocessor is a few thousand lines long, since it includes the entire contents of stdio.h and stdlib.h, which have many other function prototypes and type declarations. However, these do not affect our code (the compiler will know those types and functions exist, but since we do not use them, they do not matter). We will note that the prototype for printf contains a few features that we have not seen yet. You can think of **const char** * as being the type for a literal string, and the ... means that printf takes a variable number of arguments—a feature of C that we will not delve into beyond using it to call printf and similar functions.