More about the C Pre-Processor

`#define`

We have already seen `#define` used for defining constants: `#define MB (1024*1024)`

Wherever the symbol `MB` appears in the source, outside a string literal, it will be replaced with the text `(1024*1024)`

In C, we often define several constants at the start of a file in this fashion:

```
#define MAX_FILES 200
#define BUFFER_SIZE 2*1024 // Dangerous
#define BUFFER_SIZE (2*1024) // Much better
```

What happens if our symbol happens to appear as a function name? Compiler Error

To stop this from occurring, the convention is to write `#define` symbols in **upper-case**, making it more obvious where they are being used

`#define` with parameters

`#define` is also capable of taking parameters - we call these **macros**

Suppose we are classifying instruction words in an emulator and wanted to check whether an arbitrary set of bits in the word were set to a specific value: `#define BITS_SET(value, bits, mask) ((value & mask) == bits)`

We could use it to check a mask as follows:

```
void do_action( uint32_t opcode ) {
        if( BITS_SET(opcode, 0x01400000, 0x03e000000) )
                process_cmp(opcode);
        ...
}
```

The previous macro could have been written as a function

One of the obvious reasons for writing a macro rather than a function is to avoid the overhead of calling a function

`#define` for customised functions

What about a macro that declares something that couldn't have been written as another function?

For example, this macro writes a function each time it is invoked:

```
#define OPERATOR_FUNCTION(name, operator)\
static int name(int a, int b)\
{\
  return a operator b;\
}
```

We use backslashes to escape newlines in our macro. When we invoke this macro, as in:
`OPERATOR_FUNCTION(add, +)`

(note: **no semi-colon**) it writes a function called `add` whose body returns `a + b`

We can use this to simplify the function pointer example we saw in a previous lecture:

```
#include <stdio.h>

#define OPERATOR_FUNCTION(name, operator)\
static int name(int a, int b)\
{\
  return a operator b;\
}

OPERATOR_FUNCTION(add, +)
OPERATOR_FUNCTION(mul, *)
OPERATOR_FUNCTION(div, /)
OPERATOR_FUNCTION(mod, %)

typedef int (*opfunc)(int, int);

static void print_result( opfunc func, int a, int b ) {
  printf( "func(%d, %d) = %d\n", a, b, func(a, b) );
}

int main(void) {
  int a = 42, b = 37;
  print_result( &add, a, b );
  print_result( &mul, a, b );
  print_result( &div, a, b );
  print_result( &mod, a, b );
  return 0;
}
```

More Pre-Processor Features

The pre-processor keeps track of the source filename and line number, giving us two constants (`__FILE__` and `__LINE__`):

```
#include <stdio.h>
#include <stdlib.h>
#define FATAL(m) fprintf( stderr, "Fatal error at %s, line %d:
%s\n", __FILE__, __LINE__, m ); exit(1)

int main( void ) {
        int x = 0;
```

```
        if( x < 100 ) { FATAL( "x<100" ); }
        return 0;
}
```

Occasionally in macros, you need to turn a macro argument into a string literal. You do that using `#` as shown:

```
#include <stdio.h>
#define STRINGIFY(m) #m

int main( void ) {
        puts( STRINGIFY(hello) );
        return 0;
}
```

Still More Pre-Processor Features

You sometimes want to concatenate two macro arguments into a single C lexical token, using `##` as in:

```
#include <stdio.h>
#define CONCAT(x,y) x##y

int main( void ) {
        int CONCAT(x,y) = 10;
        printf( "xy=%d\n", xy );
        return 0;
}
```

Finally, the pre-processor automatically merges immediately adjacent string literals into a single bigger string literal:

```
#include <stdio.h>
#define STRINGIFY(m) #m

int main( void ) {
        puts( STRINGIFY(hello) " " STRINGIFY(there) );
```

```
        return 0;
}
```