

The Preprocessor

- any line beginning with a # is a preprocessor directive
- preprocessor does its job before compilation
- for example, `#include <foo.h>` (system header file) or `#include "foo.h"` (user written header file) literally includes the contents of the file at the place where the directive appears as if you had typed the contents of the file `foo.h` there
- there are other more or less used directives:
 - `#ifdef`
 - `#ifndef`
 - `#define`
 - `#endif`
- use of the above coming later in the context of header files

Macro I

- a macro is created with a `#define` command:

```
#define SMALL_LEN 5
```

- now wherever `SMALL_LEN` appears in the code the preprocessor would just *literally* substitute 5
- remember: this substitution is done before compilation
- why is this useful?
 - if you were to change the constant 5 to 500 later at some point you don't have to search and replace in all the files 5 appears just edit the following:

```
#define SMALL_LEN 500
```
 - this way you would avoid using “magic numbers” and make much less mistakes and its much more readable

Aside I

- another way to define (*only*) integer constants other than using `#define` as before is to use `enum` statement:

```
enum EscapeChars {
    BELL = '\a',
    BACKSPACE = '\b',
    TAB = '\t',
    NEWLINE = '\n',
    VTAB = '\v',
    RETURN = '\r'
};
```

- one could have used six different `#defines` like the one below to emulate the above:

```
#define BELL '\a'
```

- the difference here is that the `enums` do not vanish before (and after) compilation as do the `#defines` do (remember, all the `#defines` get substituted and hence are all gone before compilation!) and hence can debug

Aside II

- another advantage of using enum, the compiler can count for you:

```
enum Days {  
    MON = 123454321,  
    TUE,  
    WED,  
    THU,  
    FRI,  
    SAT,  
    SUN  
};
```

- in the above the values of TUE, WED, ... would be 123454321 + 1, 123454321 + 2, ... respectively, set automatically for you by the compiler: very convenient for a huge list of integer constants

Aside III

- use of enums:

```
int
main (int argc, char **argv)
{
    int day = MON;

    switch(day) {
    case MON:
        printf("This is a Monday\n");
        break;

    /* other cases here */

    case SUN:
        printf("This is a Sunday\n");
        break;

    default:
        printf("invalid code\n");
    }
    return 0;
}
```

Macro II

- going beyond magic numbers:

```
#define SQR(x) ((x) * (x))

int
main (int argc, char **argv)
{
    int num1 = 2, num2 = 4;

    printf("square of %d: %d\n", num1, SQR(num1));
    printf("square of %d: %d\n", num1 + num2, SQR(num1 + num2));
    return 0;
}
```

- note `SQR(num1)` above is replaced by `((num1) * (num1))` by the preprocessor, hmm, lots of redundant(?) braces, huh?

Macro III

- why do we need those redundant(?) braces, consider:

```
SQR(num1 + num2)
```

- the above would be substituted as:

```
((num1 + num2) * (num1 + num2))
```

- the take out the barces to get:

```
(num1 + num2 * num1 + num2)
```

- the above is wrong, right? so #define SQR(x) (x * x) is wrong!
- use parenthesis generously in defining macros, they don't hurt

Aside I

- the only ternary operator in C: “?:”
- suppose you want to do:

```
int
main (int argc, char **argv)
{
    int num1 = 2, num2 = 4, max;

    if (num1 >= num2)
        max = num1;
    else
        max = num2;
    printf("maximum of %d and %d: %d\n",
        num1, num2, max);
    return 0;
}
```

- you could replace the above if-else by:

```
max = (num1 >= num2) ? num1 : num2;
```

Macro IV

- a little sophisticated example:

```
#define MAX(a, b) (((a) >= (b)) ? (a) : (b))

int
main (int argc, char **argv)
{
    int num1 = 2, num2 = 4;

    printf("maximum of %d and %d: %d\n",
           num1, num2, MAX(num1, num2));
    printf("maximum of %d and %d: %d\n",
           num1 * num2, num2, MAX(num1 * num2, num2));
    return 0;
}
```

- note `MAX(num1, num2)` above is replaced by `(((num1) >= (num2)) ? (num1) : (num2))` by the preprocessor

Macro V

- a sophisticated example:

```
#define PRINT_BINARY(x) (((x) == 1) ? "ONE" : \  
                        (((x) == 0) ? "ZERO" : "NOT A BINARY"))  
  
int  
main (int argc, char **argv)  
{  
    printf("binary: %s\n", PRINT_BINARY(1));  
    printf("binary: %s\n", PRINT_BINARY(0));  
    return 0;  
}
```

Macro VI

- caring about efficiency in macros:

```
#define MAX_IMPROVED(a, b) ({ \
    typeof (a) _a = (a); \
    typeof (b) _b = (b); \
    (_a > _b) ? _a : _b; })
```

- note the above avoids recomputation of a and b as in:

```
#define MAX(a, b) (((a) >= (b)) ? (a) : (b))
```

- note the operator `typeof` is an esoteric one and only works with `gcc`, `gcc -pedantic` would give you warning and `gcc -pedantic -ansi` wouldn't compile
- note the effect of `max = MAX_IMPROVED(num1 * num2, num2)` would be `max` getting the value of the last expression in the above scope after macro substitution

Macro VII

- how about a mind blower?

```
#define DEBUG(_x) \  
do { \  
    printf("==== BEGIN: DEBUG block ====\\n" \  
          "file:  %s,  line:  %d\\n", \  
          __FILE__, __LINE__); \  
    _x \  
    printf("==== E N D: DEBUG block ====\\n"); \  
} while (0)
```

- note the usage, especially the last semicolon:

```
DEBUG(printf("This is a debugging message\\n");  
      printf("Hope you find it useful\\n"));
```

Macro VIII

- use the `assert` statement generously but judiciously:
 - its a macro, helps you catch your errors pretty quickly
 - use it to verify some condition which you expect to see satisfied at a certain point in your program at runtime:
 - its kills the program if the condition is not satisfied

```
int
main (int argc, char **argv)
{
    int num1 = 2, num2 = 4;

    /*
     * some computation which is not supposed to change the values
     * above variables
     */
    assert(num2 == 2 * num1);
    return 0;
}
```

Macro IX

- some more macro tricks:

```
#define DEBUG_INT_PRINT(expr) printf(#expr " = %d\n", expr)
```

- `#expr` makes `expr` a string and joins it to " = %d\n", so for example

```
DEBUG_INT_PRINT(num1 + num2);
```

would be equivalent to

```
printf("num1 + num2 = %d\n", num1 + num2);
```

Macro vs. Functions I

- consider the following (useless) structure:

```
#define FIXED_LENGTH 2

struct FixedLengthVector {
    int len;
    double vals[FIXED_LENGTH];
};
```

- the macros and functions defined on this structure are:

```
#define FIXED_LENGTH_VECTOR_LEN(vv) ((vv).len)
#define FIXED_LENGTH_VECTOR_PTR_LEN(vv_ptr) ((vv_ptr)->len)
#define FIXED_LENGTH_VECTOR_PTR_VALS(vv_ptr) ((vv_ptr)->vals)

int
flv_get_len (FixedLengthVector *vv_ptr);
double
flv_get_val_at (FixedLengthVector *vv_ptr, int pos);
```

Macro vs. Functions II

- use of the macros:

```
int
main (int argc, char **argv)
{
    int pos = 1;
    FixedLengthVector flvec1 = { 2, { 1, 100 } };
    FixedLengthVector *flvec1_ptr = &flvec1;

    DEBUG_INT_PRINT(FIXED_LENGTH_VECTOR_LEN(flvec1));
    DEBUG_INT_PRINT(FIXED_LENGTH_VECTOR_PTR_LEN(flvec1_ptr));
    printf("flv_get_val_at [%d]: %g\n", pos,
           FIXED_LENGTH_VECTOR_PTR_VALS(flvec1_ptr)[pos]);
    return 0;
}
```

Macro vs. Functions III

- use of the above functions:

```
int
main (int argc, char **argv)
{
    int pos = 1;
    FixedLengthVector flvec1 = { 2, { 1, 100 } };
    FixedLengthVector *flvec1_ptr = &flvec1;

    printf("flv_get_len: %d\n", flv_get_len(flvec1_ptr));
    printf("flv_get_val_at [%d]: %g\n", pos,
           flv_get_val_at(flvec1_ptr, pos));
    return 0;
}
```

Macro vs. Functions IV

- general advice: use functions unless its absolutely necessary to use macros
e.g.

```
#define DEBUG_INT_PRINT(expr) printf(#expr " = %d\n", expr)
```

- use macros to avoid using magic numbers all the time

Code Files

prog10.c