Chapter 6: Preprocessor

The C preprocessor provides many additional features not found in the language itself. You can use these to create constants, to include data from other files, to shoot yourself in the foot.

Problems with preprocessors are difficult to spot because they are not obvious Even the compiler may misreport preprocessor errors. For example, the following program generates an error on Line 5 when the problem is really a bad **#define** statement on Line 1.

```
1 #define VALUE_MAX 300 ++ 5   /* Problem is here */
2
3 void check(int value)
4 {
5     if (value > VALUE_MAX) {
6         printf("Value %d is out of range\n", value);
7      abort();
8     }
9 }
```

Good style is the best defense against preprocessor efforts. It is extreme important. By religiously following the rules discussed here, you can catch errors before they happen.[1]

Simple Define Statements

One of the uses of the **#define** statement is to define simple constant format is this:

```
#define SYMBOL value     /* comment */
```

The *SYMBOL* is any valid C symbol name (by convention, **#define** names are all uppercase). The *value* can be a simple number or an expression.

Like variable declarations, a constant declaration needs a comment explains it. This comment helps create a dictionary of constants.

Some examples:

```
/* Max number of symbols in a procedure */
#define SYMBOL_MAX 500

/* The longest name handled by this system */
#define NAME_LENGTH 50
```

Rule 6-1:

> **#define** *constants are declared like variables. Always put a comment describes the constant after each declaration.*

---

1. Religion, noun. Something a programmer gets after working until two in the morning only find a bug that wouldn't have been there had he or she religiously followed the rules.

Rule 6-2:

Constant names are all upper-case.

Constant expressions

If the *value* of a **#define** statement is a compound expression, you can run problems. The following code looks correct, but it hides a fatal flaw.

```
/* Length of the object (inches) (partl=10, part2=20) */
#define LENGTH 10 + 20     /* Bad practice */

#define WIDTH 30           /* Width of table (in inches) */

/*..... */
/* Prints out an incorrect width */
printf("The area is %d\n", LENGTH * WIDTH);
```

Expanding the *printf* line, you get:

```
printf("The area is %d\n", LENGTH * WIDTH);
printf("The area is %d\n", 10 + 20 * WIDTH);
printf("The area is %d\n", 10 + 20 * 30);
```

This another example of how the C preprocessor can hide problems. Clearly `LENGTH` is 10 + 20, which is 30. *S*o `LENGTH` is 30, right? Wrong. `LENGTH` literally `10 + 20`, and:

```
10 + 20 * 30
```

is vastly different from:

```
30 * 30
```

To avoid problems like this, always surround all **#define** expressions with parenthesis (`()`). Thus, the statement:

```
/* Length of the object (inches) (partl=10, part2=20) */
#define LENGTH 10 + 20       /* Bad Practice */
```

Becomes:

```
/* Length of the object (inches) (partl=10, part2=20) */
#define LENGTH (10 + 20)       /* Good Practice */
```

Rule 6-3:

If the value of a constant is anything other than a single number, enclose it in parentheses.

#define constants vs. consts

In ANSI C constants can be defined two ways: through the **#define** statement and through use of the **const** modifier. For example, the following two statement, are equivalent:

```
#define LENGTH 10       /* Length of the square in inches */
const int length = 10;  /* Length of the square in inches */
```

Which statement should you use? The **const** declaration is better because it is in the main part of the C language and provides more protection against mistakes.

Consider the following example:

```
#define SIZE 10 + 20      /* Size of both tables combined */
const int size = 10 + 20; /* Size of both tables combined */
```

As you've already seen, the **#define** statement is a problem. *SIZE* is a macro and always expands to 10 + 20. The const int size is an integer. It has the value 30. So while the statement:

```
area = SIZE * SIZE:           /* Mistake */
```

generates the wrong number, the statement:

```
area = size * size:           /* Works */
```

generates the right number. So the **const** declaration is less error-prone. Also, if you make a mistake in defining a **const,** the compiler generates an error message that points at the correct line. With a **#define,** the error appears when the symbol is used, not when it is defined.

Then why do we have the **#define**? Because early compilers did not recognize const declarations. There is still a lot of code out there that was written for these compilers and that should be modernized.

Rule 6-4:

The use of **const** *is preferred over* **#define** *for specifying constants.*

#define vs. typedef

The **#define** directive can be used to define types, such as:

```
#define INT32 long int /* 32 bit signed integer type */
```

The **typedef** clause can be used in a similar manner.

```
typedef long int int32; /* 32 bit signed integer */
```

The typedef is preferred over the **#define** because is better integrated into the C language, and it can create more kinds of variable types than a mere define.

Consider the following:

```
#define INT_PTR int    /* Define a pointer to integer */
typedef int *int_ptr;  /* Define a pointer to an integer */
INT_PTR ptr1, ptr2;    /* This contains a subtle problem */
int_ptr ptr3, ptr4;    /* This does not */
```

What's the problem with the line `INT_PTR ptr1, ptr2`? The problem is that `ptr2` of type integer, not a pointer to integer. If you expand this line, the problem, comes apparent:

```
INT_PTR ptr1, ptr2;    /* This contains a subtle problem */
int *  ptr1, ptr2;     /* This contains a subtle problem */
```

Problems like this can be avoided by using **typedef**.

Rule 6-5:

When possible, use **typedef** *instead of* **#define.**

Abuse of #define directives

It is possible to use #define directives for things other than constants. For example, the macro:

```
#define FOR_EACH_ITEM for (i = first; i < last; ++i)
```

can define a standard for loop. This can be used in place of a regular for.

```
FOR_EACH_ITEM
    process_item(i);
```

You can even go so far as to create macros that make your C code look like Pascal.

```
#define BEGIN I
#define END I

/*...    */
if (x == Y)
  BEGIN
    / *...   * /
  END;
```

The problem with this approach is that you are obscuring the C language itself. The maintenance programmer who comes after you will know C, not a half-Pascal half-C mongrel.

Even the simple `FOR_EACH_ITEM` macro hides vital C code. Someone else reading the program would have to go back to the definition of `FOR_EACH_ITEM` to figure out what the code does. By using the code instead of a macro, no lookup is necessary,

You can easily understand the C code that goes into this:

```
for (i - first; i < last; ++i)
    process_item(i);
```

Rule 6-6:

Don't use **#define** *to define new language elements.*

Keywords and standard functions

Defining new language elements is one problem. A far more difficult problem occurs when a programmer redefines existing keywords or standard routines. For example, in one program, the author decided to create a safer version of the string copy routine:

```
#define strcpy(s1, s1) \
x_strcpy(s1, s2, sizeof(s1), sizeof(s2))
```

This worked great until the program was ported. Then the program mysteriously bombed at the code:

```
/* This lines hangs the system */
strcpy(name, "noname.c");
```

The programmer performing the port was baffled. There was nothing wrong with the parameters to *strcpy.* And of course, because *strcpy* is a standard function, there Shouldn't be a problem with it.

But in this case, *strcpy* is not a standard function. It's a non-standard macro that results in a great deal of confusion.

Think about how difficult it would be to find your way if someone gave you directions like these: “When I say north I mean west, and when I say west I mean north. Now, go north three blocks, turn west for one (when I say one I mean four), and then east two. You can't miss it.”

Rule 6-7:

Never use **#define** *to redefine C keywords or standard functions.*

Parameterized Macros

The **#define** may have arguments. For example, the following macro doubles a number:

```
/* Double a number */
#define DOUBLE_IT(number) (2 * (number))
```

Enclosing the entire macro in parenthesis avoids a lot of trouble similar to the problems with simple **#define**s.

Rule 6-8:

Enclose parameterized macros in parentheses.

In the next example, the macro *SQUARE* is supposed to square a number:

```
/* Square a number */
#define SQUARE(X) (x * x)
/* Bad practice, no () around parameter */
```

The invocation of the macro:

```
a = SQUARE(1 + 3);
```

expands to:

```
a = (1 + 3 * 1 + 3);
```

which is not what was expected. If the macro is defined as:

```
/* Square a number */
#define SQUARE(X) ((x) * (x))
```

Then the expansion will be:

```
a = ((l + 3) * (1 + 3));
```

Rule 6-9:

Enclose each argument to a parameterized macro in parenthesis.

Multi-line Macros

The **#define** statement can be used to define code as well as constants. For example:

```
/* Print current values of registers (for debugging) */
#define PRINT_REGS printf("Registers AX=%x BX=%x\n", AX,BX);
```

This is fine as long as the target of the **#define** is a single C statement. Problems occur when multiple statements are defined. The following example defines a macro ABORT that will print a message and exit the system. But it doesn't work when put inside an **if** statement.

```
/* Fatal error found, get out of here */
#define ABORT print("Abort\n"); exit(8);

/*....   */
if (value > LIM)
    ABORT;
```

problem can easily be seen when we expand the macro:

```
if (value > LIM)
    printf("Abort\n"); exit(8);
```

Properly indented, this is:

```
if (value > LIM)
    printf("Abort\n");
exit(8);
```

This is obviously not what the programmer intended. A solution is to enclose multiple statements in braces.

```
/* Fatal error found, get out of here */
#define ABORT { printf("Abort\n"); exit(8); )
    /* Better, but not good */
```

This allows you to use the ABORT macro in an **if,** like this:

```
if (value > LIMIT)
    ABORT;
```

Unfortunately, it causes a syntax error when used with an **else:**

```
if (value > LIMIT)
    ABORT;
else
    do_it();
```

The **do/while** statement comes to the rescue. The statement:

```
do {
    printf("Abort\n");
    exit(8);
} while (0);
```

executes the body of the loop once and exits. C treats the entire **do/while** as a single statement, so it's legal inside a **if/else** set.

Therefore, properly defined, the macro is:

```
/* Print an error message and get out */
#define ABORT                               \
    do {                                    \
        printf("Abort\n");                  \
        exit(8);                            \
} while (0)                     /* Note: No semicolon */
```

Rule 6-10:

Always enclose macros that define multiple C statements in braces.

Rule 6-11:

If a macro contains more than one statement, use a do/while structure to enclose the macro. (Don't forget to leave out the semicolon of the statement).

When macros grow too long, they can be split up into many lines. The preprocessor uses the backslash (\) to indicate "continue on next line." The latest `ABORT` macro also uses this feature.

Always stack the backslashes in a column. Try and spot the missing backslash in the following two examples:

```
/* A broken macro */
#define ABORT                               \
    do {
        printf("Abort\n");                  \
        exit(8);                            \
} while (0)

/* Another broken macro */
#define ABORT \
    do { \
        printf("Abort\n"); \
        exit(8);
} while (0)
```

The mistake in the first example is obvious. In the second example, the problem is hidden.

Rule 6-12:

When creating multi-line macros, align the backslash continuation characters (\) in a column.

Macros and Subroutines

Complex macros can easily resemble subroutines. It is entirely possible to create a macro that looks and codes exactly like a subroutine. The standard functions `getc` and `getchar` are actually not functions at all, but macros. These types of macros frequently use lower-case names, copying the function-naming convention.

If a macro mimics a subroutine, it should be documented as a function. That involves putting a function-type comment block at the head of the macro:

```
/*****************************************************
 * next_char -- move a buffer pointer up one char    *
 *                                                   *
 * Parameters                                        *
 *      ch_ptr -- pointer to the current character   *
 *                                                   *
 * Returns                                           *
 *      pointer to the next character or NULL if none.  *
 *****************************************************/
#define next_char(ch_ptr) \
    /* ... definition ... */
```

Rule 6-13:

Always comment any parameterized macros that look like functions.

The #include Directive

Include files are used to define data structures, constants, and function prototypes for items used by multiple modules. it is possible to put code in an include file, but this is rarely done.

Style for #Includes

,Most programs put the **#include** directives in a group just after the heading comments. That way they are all together in a known place. System includes are enclosed in <>) come first, followed by any local includes (enclosed in "").

Example:

```
/*********************************************************
 *   ... heading comments ....                           *
 *********************************************************/
/* System includes */
#include <stdio.h>
#include <alloc.h>
#include <string.h>

/* Local includes */
#include "key.h"
#include "table.h"
#include "blank.h"
```

Rule 6-14:

> **#include** *directives come just after the heading comments. Put system includes first, followed by local includes.*

#include directives that use absolute file names, that is specify path and name, such as */user/sam/program/data.h* and *Y:\DEVELOP\PROGRAM\DEFS.H* make your program non-portable. If the program is moved to another machine, even one using the same operating system, the source will have to be changed.

The solution is to never use absolute paths. The compiler can be pointed to the correct directory by using the `-I` option. That way you need to change only one Makefile instead of a bunch of source files.

```
/* Non portable */
#include “/user/sam/program/data.h”

/* Portable, compile with “-I/user/sam/program” */
#include “data.h”
```

Rule 6-15:

> *Do not use absolute paths in* **#include** *directives. Let the -I compile opt*

Protecting against double #Includes

Include files can contain **#include** directives. This means that you can easily include the same file twice. For example, suppose *database.h* and *symbol.h* both need the file *defs.h*. Then, putting these lines:

```
#include “database.h”
#include “symbol.h”
```

in your program brings in two copies of *defs.h.* Defining a structure or type twice can cause errors. So how do you avoid this problem? The solution is t conditional compilation to prevent the double include from causing trouble.

```
#ifndef _DEFS_H_INCLUDED_
#define _DEFS_H_INCLUDED_
```

And at the end, insert this line:

```
#endif /* _DEFS_H_INCLUDED_ */
```

The first time through, *_DEFS_H_INCLUDED_* is not defined.

The **#ifndef** causes the entire body of the file to be included and `_DEFS_H_INCLUDED_` to be defined. Therefore, when the file is include the **#ifndef** kicks in, and the entire body of the file is now **#ifdef**'ed out.

Conditional Compilation

The preprocessor allows you conditionally to compile sections o through the use of **#ifdef**, **#else**, and **#endif** directives.

For example:

```
#ifdef DOS
#define NAME "C:\ETC\DATA”
#else /* DOS */
#define NAME "/etc/data”
#endif /* DOS */
```

Actually, the **#else** and **#endif** directives take no arguments. The following them is entirely a comment, but a necessary one. It serves to match **#else** and **#endif** directive with the initial **#ifdef**.

Note: Some strict ANSI compilers don't allow symbols after **#else** or **#endif** directives. In these cases, the comment DOS must be formally written as `/* DOS */`.

Rule 6-16:

> *Comment* **#else** *and* **#endif** *directives with the symbol used in the initial* **#ifdef** *or* **#endif** *directive.*

Use conditional compilation sparingly. It easily confuse the code.

```
#ifdef SPECIAL
float sum(float a[])
#else /* SPECIAL */
int sum(int bits)
#endif SPECIAL
{
#ifdef SPECIAL
    float total;    /* Total so far */
#else /* SPECIAL */
    int total;      /* Total number of bits */
#endif /* SPECIAL */
    int i;          /* General index */

#ifdef SPECIAL
    total = 0.0;
#else /* SPECIAL */
     total = 0;
#endif /* SPECIAL */

#ifdef SPECIAL
    for (i = 0; a[i] != 0.0; ++i)
        total += (( bits & i) != 0);
#else /* SPECIAL */
    for (i = Ox8O; i != 0: i >> 1)
        total += a[i];
#endif /* SPECIAL */

    return (total);
}
/*
 * A comment explaining that this
 * is bad code would be redundant
 */
```

The structure of this function is nearly impossible to pick out. Actually, it consists of two completely different functions merged together. There are a few lines of common code, but not many.

```
float sum(float a[])
{
    float total;    /* Total so far */
    int i;          /* General index */

    total = 0.0;

    for (i = 0; a[i] != 0.0; ++i)
        total += (( bits & i) != 0);

    return (total);
}

int sum(int bits)
{
    int total;      /* Total number of bits */
    int i;          /* General index */

     total = 0;

    for (i = Ox8O; i != 0: i >> 1)
        total += a[i];

    return (total);
}
```

Avoid complex conditional sections. C is difficult enough to understand without confusing the issue. Usually it is better to write two entirely separate but clearer functions.

Rule 6-17:

Use conditional compilation sparingly. Don't let the conditionals obscure the code.

Where to define the control symbols

The control symbols for conditional compilation can be defined through **#define** statements in the code or the -D compiler option.

If the compiler option is used, the programmer must know how the program was compiled in order to understand its function. If the control symbol is defined in the code, the programmer needs no outside help. Therefore, avoid the compiler option as much as possible.

Rule 6-18:

Define (or undefine) conditional compilation control symbols in the code rather than using the `-D` *option to the compiler.*

Put the **#define** statements for control symbols at the very front of the file. After all, they control how the rest of the program is produced.

Use the **#undef** statement for symbols that are not defined. This serves several functions. It tells the program that this symbol is used for conditional compilation. Also, **#undef** contains a comment that describes the symbol Finally, to put the symbol in, all the programmer needs to do is change the **#undef** to **#define.**

```
#define CHECK    /* Internal data checking enabled */
#undef DEBUG    /* Not the debug version of the program */
```

Rule 6-19:

*Put **#define** and **#undef** statements for compilation control symbols at the beginning of the program.*

Commenting out code

Sometimes a programmer wants to get rid of a section of code. This may be because of an unimplemented feature, or some other reason. One trick is to comment it out, but this can lead to problems:

```
/*-------Begin commented out section ------
open_database();
update_symbol_table();     /* Add our new symbols */
close_database();
/*-------End commented out section------*/
```

Unless your compiler has been extended for nested comments, this code will not compile. The commented-out section ends at the line */* *Add our new symbols */,* not at the bottom of the example.

Conditional compilation can accomplish the same thing, with much less hassle.

```
#ifdef UNDEF
open_database();
update_symbol_table();     /* Add our new symbols */
close_database();
#endif /* UNDEF */
```

Note: This will not work if the programmer defines the symbol (However, any programmer who defines this symbol should be shot.)

Rule 6-20:

Do not comment out code. Use conditional compilation (`#ifdef UNDEF`*) to get rid of unwanted code.*

Sometimes the programmer wants to take out a section of code for a minutes for debugging. This can be done in a similar way:

```
#ifdef QQQ
    erase_backups();
#endif QQQ
```

The symbol *QQQ* was chosen because it is probably not defined and is easy spot with an editor. This allows all *QQQ* lines to be quickly removed when the is found.

Rule 6-21:

Use `#ifdef QQQ` *to temporarily eliminate code during debugging.*