Chapter 3: Variable Names

In English, we put words together to make up sentences. The language is fairly easy to understand when you know what most of the words mean. Even if you don't know some words, you can look them up in the dictionary.

Variables are the “words” for the C language. In a program, variables have a precise definition and usage, but that definition and usage are different for each program. What's worse, some programmers tend to use abbreviations, even for simple things. Shakespeare wrote, “That which we call a rose by any other name would smell as sweet.” (Romeo and Juliet, Act II, Scene 2). But calling a rose an “RZ” creates needless confusion.

Bureaucratize is a prime example of how things get mixed up when people start using their own unique languages. Government agencies don't fire people, they “dehire” them. That probably wouldn't be confusing to the person being dehired, but consider this example: The Army files “Zipper” under “I.” why? Zipper used to be a trade name, making it illegal for Army filing, so they use the generic name “Interlocking cloth fastener.” These are the same people who file furry teddy bears under the label “Bears, fur, Edward.”

Call a spade a spade. Don't call it “spa”, “s1”, or “pronged digging implement.” Simplicity and a firm grasp of the obvious are necessary for good C programming.

Rule 3-1:

Use simple, descriptive variable names.

A Brief History of the Variable

Early computers were initially used for solving complex and repetitive mathematical equations. Not surprisingly, early programming languages looked a lot like algebra. Mathematicians generally use single character variable names because they don't care what the variables stand for. (They're not supposed to; that's what it means to be a mathematician.)

For example, the equation for the area of a triangle is:

$$a = \frac{1}{2}bh$$

where a is the area of the triangle, b is the base, and h is the height.

This sort of notation is highly compact. You can express a lot on a single line.

$$\frac{\frac{1}{n}\sum U_f U_k + 1 - \left(\frac{1}{n}\sum U_f\right)\left(\frac{1}{n}\sum U_f + 1\right)}{\lim_{n \to \infty} \sqrt{\left(\frac{1}{n}\sum U_j^2 - \left(\frac{1}{n}\sum U_f\right)^2\right)\left(\frac{1}{n}\sum {U_f}^2 + k - \left(\frac{1}{n}\sum U_f + k\right)^2\right)}}$$

However, it isn't very clear. Even the simple triangle example requires a somewhat longer line of explanation so we know the meaning of `a`, `b`, and `h`.

In mathematics, you can append notes when you need to break out of the sea of symbols to explain what the symbols stand for. In programming, where code can easily run 10 or more pages and where you care a little more what the symbols stand for, variable names should be more meaningful.

As people discovered that they needed longer variable names, computer languages improved. The first BASIC interpreter limited variables to a single letter and an optional digit (`A`, `B2`, `C3`, etc.)

FORTRAN gave the programmer six characters to play with—really 5 1/2, since the first character denoted the default type. That meant that instead of using I for an index, you could use a name like `INDEX`. This was progress. (One problem with FORTRAN was that it used the first letter of the variable name for type information. So people would use `KOUNT` for "count" so that the type would be integer.)

In C the length of a variable name is unlimited (although the first 31 characters must be unique). So variable names like these:

```
disk_name       total_count          last_entry
```

are legal.

If long is better, then very long must be much better, right? Consider this example:

```
total_number_of_entries_with_mangled_or_out_of_range_dates
```

This is an extremely descriptive name; you know exactly what this variable is used for. But there are problems with names like this. First, they are difficult to type. Second, remembering the exact wording of such a long name is not easy. Finally, look at what happens when you try to use this variable in a statement:

```
total_number_of_entries_with_mangled_or_out_of_range_dates =
    total_number_of_entries_with_mangled_input +
    total_number_of_entries_with_mangled_output +
    total_number_of_entries_with_out_of_range_dates;
```

True, you know what the variables are, but the statement logic is obscured by the excess verbosity of the names.

Choosing the right variable name is a balancing act. It must be long enough to be descriptive, yet short enough to be memorable and useful.

Over the years, the following rule of thumb has evolved.

Rule 3-2:

Good variable names are created by using one word or by putting two or three words together, separated by "_". For example:

```
/* Good variable names */
start_time     start_date     late_fee
current_entry  error_count    help_menu
AllDone        ModemName      LastNumberCalled
```

Capitalization

Shortly after the invention of moveable type, printers began to arrange their letters in specially designed boxes, or cases. Soon a standard arrangement emerged: two drawers were used for each typeface, the top one holding the capital letters, and the bottom for all the others. Thus the terms uppercase and lowercase.

In many programming languages case is ignored, but in C, uppercase is distinguished from lowercase. This means, for example, that Count, count and COUNT are three different names. This can lead to problems, but it also gives you another tool for making variable names meaningful.

Over the years programmers have devised special naming and capitalization conventions for variables, functions, and constants.

System A

`total_count`	Variable and function names	All lowercase words separated by underscores
`NAME_MAX`	Constants	All uppercase words separated by underscores

One of the advantages of this system is that all the component words (total, count, name, max) are separated from each other. This allows you to run the program through a spelling checker.

System B

`TotalCount`	Variable and function names	Upper/Lower case with no separators.
`NAME_MAX`	Constants	All uppercase words separated by underscores

This system uses a different style for variables and functions. Research shows, incidentally, that people find upper- and lowercase words easier to read than lowercase only. System B is not very common.

System C

`total_count`	Local variable and function names	All lowercase words separated by underscores
`TotalCount`	Global variables and functions	Uppercase and lowercase with no separators.
`NAME_MAX`	Constants	All uppercase words separated by underscores

This system uses a different format for local and global names, which provides additional programming information.

Each system has its own advantages. System A used to be the universal standard, but System C is quickly growing in popularity. Choose the one that suits you best and stay with it.

Names You Must Never Use

A programmer once came up with a brilliant way to avoid ever getting a traffic ticket. He submitted a request for a personalized license plate with the choices "`OOOOOO`", "`1l1l1l`", and "`llOO11`". He figured that if his license plate read "`OOOOOO`", the police would find it difficult to tell the difference between the letter "`O`" and the digit "`O`". The problem was, the DMV clerk had the same problem, so he got a personalized license plate that read "`000000`".

The uppercase letter "`O`" and the digit "`O`" can easily be confused. So can the lowercase letter "`1`" and the digit "`1`".

Rule 3-3:

Never use `I` *(lowercase* `L`*) or* `O` *(uppercase* `O`*) as variable or constant names.*

Other Names Not To Use

Don't use names already in the C library. You'll never know who calls them. I recently ported a program that defined its own version of *getdate*. The program worked under UNIX because although the C library has a *getdate* function, the program never expected to use it.

When the application was ported to the PC, I discovered that *getdate* called the library function *time*. This function had an internal call to *getdate*. It expected to call the system *getdate*, not a local function defined in the program. But the program overrode the library's *getdate*, which resulted in *getdate* calling *time* calling *getdate* calling *time*—until the stack overflowed.

A quick global rename was done to turn *getdate* into `get_current_date`, and the porting problem went away. But it would have never occurred in the first place if the programmer hadn't used an existing C library function:

Rule 3-4:

Don't use the names of existing C library functions or constants.

Avoid Similar Names

Subtle differences in variable names should be avoided. For example, the variable names total and totals can be easily confused. Differences between variables should be blatant, such as `entry_total` and `all_total`.

Rule 3-5:

Don't use variable names that differ by only one or two characters. Make every name obviously different from every other name.

Consistency in Naming

Consistency and repetition are extremely powerful programming tools. Use similar names for similar functions. In the following example, you can easily guess the name of the missing variable:

```
int start_hour;    /* Hour when the program began */
int start_minute;  /* Minute when the program began */
int ??????;        /* Second when the program began */
```

If `start_hour` is the hour when the program began and `start_minute` is the minute, you can easily figure out the name of the variable that holds the seconds. Think how confusing it would be if the programmer had written this:

```
int start_hour;    /* Hour when the program began */
int begin_minute;  /* Program start time. minutes only */

/* Seconds on the clock at program commencement */
int commence_seconds;
```

Rule 3-6:

Use similar names for variables that perform similar functions.

Which Word First

Suppose you have a variable that denotes the maximum entry in a series of numbers. You could call it `max_entry` or `entry_max`. How do you decide which name to use?

Picking one at random is does not work, because you might at one time pick `max_entry` for one program and `entry_max` for another. Experience shows that too often we forget which one we picked for a particular program, which results in confusion. More often than I care to mention, I've had to do a global search and replace to change `max_entry` to `entry_max`.

You need a selection rule. What happens when you put the most important word first (in this case, entry)? A cross reference listing will group all the entry related variables together, such as (`entry_count`, `entry_min`, `entry_mnax`).

If you choose to begin with the word `max`. then all the maximum limits will be grouped together (`max_count`, `max_entry`, `max_list`).

Sorting by an important relation (all variables related to entries) is more important than sorting by type (all maximums).

Rule 3-7:

> *When creating a two word variable name where the words can be put in any order, always put the more important word first.*

Standard Prefixes and Suffixes

Over the years a few standard prefixes and suffixes have developed for variable names. These include the following:

`_ptr` Suffix for pointer

Examples:

```
int *entry_ptr; /* Pointer to current entry */
char *last_ptr; /* Pointer to last char in str */
```

`_p` Another suffix for pointer. This can be a little confusing to people who are not familiar with it, so the suffix `_ptr` is preferred.

Examples:

```
event *next_p;/* Pointer to next event in queue */
char *word_p; /* Pointer to start of next word */
```

`_file` A variable of type FILE *, or a C++ I/O stream.

Examples:

```
FILE *in_file; /* Input data file */
FILE *database_file;/*Where we put the database */
```

`_fd` File descriptor (returned by the open function.)

Examples:

```
/* The dictionary file descriptor */
int dictionary_fd;

/* File where we put the memory dump */
int dump_fd;
```

`n_` Number of. For example, if you store a set of events in the array events, the `n_events` is the number of entries in the events array. Does this violate the rule about putting the most important word first? Yes, but it's established usage.

Examples:

```
/* A list of events */
int events[EVENT_MAX];

/* Number of items in event array */
int n_events = 0;

/* A list of accounts */
struct account account;

/* Number of accounts seen so far */
int n_accounts = 0;
```

Rule 3-8:

Standard prefixes and suffixes are `_ptr`, `_p`, `_file`, `_fd`, *and* `n_`.

Module Prefixes

When creating large modules or libraries (more than 10 functions), a prefix is sometimes added to each variable and function in the library. For example, everything in a database library might start with the prefix `Db`.

Example:

```
int DbErrorNumber;
extern int DbOpen(char *name);
extern int DbClose(int handle);
```

This serves two purposes: first, it identifies the module containing the name; and second, it limits name conflicts. A symbol table module and a database both might have a lookup function, but the names `SymLookup` and `DbLookup` do not conflict.

The X Windows system uses this naming convention extensively. All X Windows functions begin with the letter X. However, the system is so complex that it has been further divided into "tool kits," each of which has its own prefix. For example, Xt is the Andrew Tool kit, Xv is the X-view tool kit, etc.

Special Prefixes and Suffixes

Sometimes you need to use special names, names that you can be sure don't conflict with a large body of existing code. Such cases call for unusual naming conventions.

For example, the C preprocessor had been around a number of years before the ANSI Committee decided on a standard set of predefined symbols. In order to avoid conflict, they decided that each symbol would look like this: (`__SYMBOL__`).

Some of the predefined symbols include:

```
__LINE__        __FILE__  __STDC__
```

Compiler manufacturers have now jumped on this bandwagon and defined their own special symbols using this convention.

The utilities lex and yacc solve the naming problem in a different way: they begin everything with `yy`. Thus we get names like `yylex`, `yytext`, and `yylength` in the code generated by these utilities. It may look a little strange at first, but after a yywhile yyou yyget yyused to it.

If you do need to define a name that's widely used and you want to minimize the possibility of a naming conflict, begin it with an underscore (_). Very few programmers use this character at the beginning of normal variable or constant names.

When You Can Use Short Names

In some cases you can use short variable names. For example, when dealing with a graphic position, the variables `x` and `y` are descriptive.

Also, the variable named "`i`" is frequently used as a general purpose, handy dandy, local index. Its popularity makes it acceptable as a variable name, even though the name index is more descriptive.

Rule 3-9:

> *Short names such as* `x`, `y`, *and* `i` *are acceptable when their meaning is clear and when a longer name would not add information or clarity.*

argv, argc

The main function of a C program takes two arguments. In 99 percent of the programs, the arguments are named `argv` and `argc`. In the other 1 percent, we wish the programmer had used `argc` and `argv` instead of `ac` and `av`.

A lot of history has gone into these names. When programmers see them, they immediately think "command line arguments." Don't confuse the issue by using these names for anything else.

Rule 3-10:

> *Use* `argc` *for the number of command line arguments and* `argv` *for the argument list. Do not use these names for anything else.*

Microsoft Notation

When Microsoft introduced Windows, it also introduced a new variable naming notation called Hungarian Notation. (There are two reasons why it's called that. First, Charles Simonyi, the man who invented it, is Hungarian. Second, most people looking at it for the first time think that it might as well be written in Hungarian.) It's also known as Microsoft Notation.

The idea is simple: prefix each variable name with a letter denoting its type; for example, `w` for a 16-byte integer (**word**), and `l` for a 32-byte integer (**long**). That way, you can easily prevent programming problems caused by type conflicts. For example:

```
wValue = lParam; /* Obvious type conflict */
```

There is no complete list of prefixes. The following was gathered from several sources:

Prefix	Type
b	Boolean (true or false)
w	Word, 16-bit integer
i	Integer, 16-bit integer (conflicts with w)
n	Short, 16-bit integer (conflicts with w)
n	Near pointer (ambiguous, can be used for "**short**")
p	Pointer
d	Double, 32-bit integer
dw	Double word, 32-bit integer (conflicts with d)
l	Long, 32-bit integer (conflicts with d)
fn	Function (or pointer to function)
g	Global
s	String
sz	String terminated with zero (conflicts with s)
c	character
by	byte (unsigned character)
h	Window handle
hn	Window handle (conflicts with h)

There are some problems with this notation. First, the list of prefixes is confusing and incomplete, and the order of prefixes is not clear. For example, does a pointer to a word start with pw or wp?

Second, variables with type prefixes get sorted by type in the cross reference, which is not the most useful ordering.

Of course, sometimes a programmer really needs to put type information into a variable name. For example, it's very important to know the difference between things and pointers to Wings. The suffix _ptr does this job well.

Example:

```
char name[30]: /* Name of the user */
char *name_ptr: /* Pointer to user's name */
```

Suffixes easily do the job of Microsoft's prefixes without getting in the way. The only advantage of Microsoft Notation is that it makes type conflicts obvious. For example:

```
wValue = lParam: /* Obvious type conflict */
```

However, most good compilers will produce a warning message for potential problems like this. So while it may be hard to spot the potential problem in the following line:

```
count = index:
```

it's a lot easier when the compiler chimes in with:

```
Line 86: Warning: Assignment may lose significant digits
```

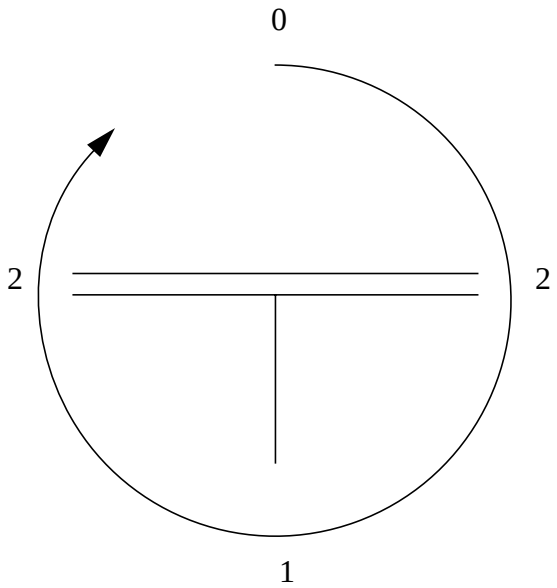
Imaginative Solutions

PC class machines come with a line drawing character set that allows the programmer to construct things like single and double lined boxes. One of the problems PC programmers face is what to name these curious characters when they are referred to in a program. One solution is to begin every single line character with S. followed by the character type: C for corner, L for line, T for T and X for cross, followed by a name. The result is:

Character	Name
┐	`S_C_UR` (Single, Corner, Upper Right)
┌	`S_C_UL` (Single, Corner, Upper Left)
┘	`S_C_LR` (Single, Corner, Lower Right)
└	`S_C_LL` (Single, Corner, Lower Left)
─	`S_L_A` (Single, Line, Across)
|	`S_C_UR` (Single, Line Down)
╪	`SD_X_DA` (Single Down, crossing Double Across)

After a while this system tends to make you sick. The problem with this system is that it's complex and somewhat error prone. At the time it was the best we could come up with, but that didn't stop us from trying something new.

Then someone figured out a system where you started at the top and worked your way around, counting the number of lines (0, 1, or 2).



So the character turns into `L_0212`. The table is now:

Character	Name
┐	`L 0011`
┌	`L_0110`
┘	`L_1001`
└	`L_1100`
─	`L_0101`
│	`L_1010`
┼	`L_1212`

Now, these names break all the rules we have discussed so far. They are short and somewhat cryptic. But put them in a file with a very good comment block explaining them, and they bring order to a very messy problem.

Case studies

Over the years different groups have developed their own standard naming conventions. Each has its own advantages and disadvantages. This section will look at some of the standard programs and how they use their names.

The C runtime library

The C runtime library traces its roots back to the first C compiler. It evolved over the years, and the ANSI Committee standardized it.

Naming conventions:	
Private variable names	All lowercase
Public variable names	All lowercase
Constant names	Uppercase only

The C library is full of short, cryptic names like these:

```
creat       stdin     stdout    open
strcpy      printf    memcmp    malloc
```

Initially names were restricted because of severe limitations in the compiler, which ran on some extremely small machines. Also, the early programmers didn't place any value on long names.

These names are somewhat cryptic. For example, the function `strcpy` copies a string. A much better name would have been `string_copy`. And using `creat` instead of `create` is pretty silly.

The language does not make good use of prefix and suffix letters. For example, here is the *printf* family of functions:

printf	Print to standard output
fprintf	Print to a file

sprintf	Print to a string
vfprintf	Print to a file, argument list is a vector

Also, the library is fairly consistent. For example, every string function *str...* has a corresponding *strn...* function.

Example:

strcpy	copy a string
strncpy	copy n characters of a string
strcat	append one string to another
strncat	append n characters of one string to another

Constant names are also somewhat cryptic. For example, some of the error codes (from standard library *errno.h*) are:

```
#define EZERO   0   /* Error 0 */
#define EINVFNC 1   /* Invalid function number */
#define ENOFILE 2   /* File not found */
#define ENOPATH 3   /* Path not found */
#define ECONTR  7   /* Memory blocks destroyed */
#define EINVMEM 9   /* Invalid memory block address */
```

However, the C library is good about grouping similar constants together. All error numbers begin with `E`, all open flags begin `O_`, and so on.

All in all, the C 1ibrary is fairly well designed; and the naming, though short, is regular and reasonable, with limitations tractable to the days when C compilers were much more limited.

The UNIX kernel

The UNIX operating system was one of the very first to be written in a high level language. It was the first to be widely ported. Today, almost every computer that's not a PC clone runs UNIX.

Naming Conventions	
Private variable names	All lowercase
Public variable names	All lowercase
Constant names	Uppercase only

UNIX is the king of the 1 to 3 character variable names. Some typical names are:

```
u    bp   i    bn
pid  uid  gid  fd
```

After a while, UNIX operating system programmers learn the meaning of most of the abbreviations. The know that `pid` stands for process `id` and `bp` is a buffer pointer. But it takes time and effort to learn this code. In fact, UNIX internal programming is not for the inexperience or the faint of hear.t Most programmers must be introduced into the world of UNIX internals by an experience guru.

As UNIX evolved, more and more people added to its code. While the core of the system remains cryptic, much of the new code is better. Longer variable names came into use, such as:

```
physical_io          dispatch
signal               tty_select
```

The Microsoft library

Microsoft Windows provides PC programmers with a graphics programming environment. It also allows programmers to better use the power of the more advanced 80386 and 80486 processors.

Naming conventions:

Private variable names	Up to the application programmer
Public variable names	Upper and lowercase
Constant names	Uppercase only

Function names in Microsoft Windows are nicely done, consisting of several words put together. Examples:

```
GetFreeSpace          UnlockSegment  CreateBitmap
CloseClipboard        GetWindow      AppendMenu
```

However, there is no special prefix or suffix for Windows functions, so it's impossible to tell at a glance whether or not `OpenFile` is a Windows function. Constants are all Uppercase.

Examples:

```
LB_SETSEL          WM_MOUSE       WM_MOVE
WM_CLOSE           EN_UPDATE      LB_MSGMAX
```

Each constant contains a short group prefix. For example, all “List box” related constants begin with `LB_`. Almost all Windows constants contain only one “_”, which is used to separated the prefix from the rest of the constant. Multiple words are run together, making it difficult to read. Therefore a constant like `WM_WINDOWPOSCHANGED` would be much more readable if it was written as `WM_WINDOW_POS_CHANGED`.

In general, the Windows naming convention makes programs more readable than the UNIX code or C library. Although not perfect, it is a step forward.

The X Windows System

The X Windows is a popular UNIX windowing system available from MIT. Its low cost and relative availability make it the windowing system of choice for most UNIX systems.

Naming conventions:

Private variable names	Up to the application programmer
Public variable names	Uppercase and lowercase
Constant names	Most Uppercase only, some upper and lowercase

One of the most refreshing things about X Windows programming is that it actually looks like someone thought about the design and style of the system before beginning the coding.

Public function and variable names in X Windows are upper and lowercase words, with no underscores.

Examples:

```
XDrawString          XNextEvent      XGrabPointer
XCreateGC            XMapWindow      XFlush
```

All public names start with the letter X. The programmer can select among many different tool kits for code. Each of these has its own prefix. For example, all routines in the X-View tool kit start with Xv.

Constants begin with a type prefix. For example, all button related constants begin with `BUTTON_`. Constants are a series of words separated by underscores.

```
XA_POINT                XA_BITMAP       XA_ATOM
XrmOptionNoArg          XSetSizeHints   Focusin
```

The X Windows system shows what can happen when some thought is given to style before coding begins. It is well designed and presents the programmer with one of the best interfaces available.

Variable Declaration Comments

Choosing good variable names helps create programs that are easy to read and maintain. But names can't do the job alone. The programmer needs a definition for each variable. Technical books have glossaries that explain all the special terms. We also need a glossary for our programs.

Writing a glossary is a lot of work. Maintaining it and keeping it up to date is even more work. A better solution is to follow each variable with a comment that describes it.

Examples:

```
int window;       /* Current window index */
int words;        /* Number of words in the document */
int line_number;  /* Current input file line number */
char *in_name;    /* Current input file name */
```

Now, if you want to know the definition of line_number, go to your cross reference and look for the first reference to the variable, thus locating the line:

```
int line_number; /* Current input file line number */
```

Using this method, you can quickly determine the type and definition of any variable.

Rule 3-11:

Follow every variable declaration with a comment that defines it.

Units

Weather services generally measure rainfall in hundredths of inches, referring to half an inch of rain as 50. However, one night the weather service computer used inches as input. Someone forgot a decimal point and entered 50 instead of 0.50.

Now, 50 inches of rain is a lot of rain. The computer caught the effort and printer this message:

```
Build an Ark. Gather the animals two by two.
```

Units of measure are important. It's one thing to describe dist as the distance between two objects, but what are the units? They could be inches, centimeters, yards, meters, or light years. It makes a difference.

Rule 3-12:

Whenever possible, include the units of measure in the description of a variable.

Examples:

```
int distance-left; /* Distance we've got left (in miles) */
int owed;    /* Amount owed in current account (in cents) */

/* Acceleration of gravity (in furlongs/fortnight**2) */
float gravity;
```

I once had to write a graphics conversion program. Many different units were used throughout the system, including inches, thousandths of an inch, plotter units, digitizer units, etc. Figuring out which units to use was a nightmare. Finally, I gave up and put the following comment in the program:

```
/*******************************************************
 * Warning: This program uses a lot of different types  *
 *      of units.  I have no idea what the input units  *
 *      or nor do I have any idea what the output       *
 *      units should be, but I do know that if you      *
 *      divide by 3 the plots look about the right size.*
 *******************************************************/
```

Structures and unions

A structure is simply a group of related variables tied together to form a convenient package. Each field in a structure should be treated like a variable, with a carefully chosen name. A descriptive comment is necessary as well.

Example:

```
/*
 * A square of space on the screen enclosed by
 * a border
 */
struct box {
    int x;     /* X loc. of upper left corner (in pixels) */
    int y;      /* Y loc. of upper left corner (in pixels) */
    int length;  /* Length of the box in pixels */
    int width;   /* Width of the box in pixels */
};
```

The structure itself is described in a comment just before its definition. This example uses a multi-line comment to describe the box. Single-line comments tend to get lost in the clutter. White space before and after the definition separates the structure from the rest of the code (much like blank lines separate paragraphs in this book).

Rule 3-13:

Name and comment each field in a structure or union like a variable.

Rule 3-14:

Begin each structure or union definition with a multi-line comment describing it.

Rule 3-15:

Put at least one blank line before and after a structure or union definition.

Long declarations and comments

Sometimes a variable declaration and its initializer leave little room for a comment. In the following example, we need to describe *last_entry*, but where do we put the comment?

```
int first_entry;          /* First entry to process */
int last_entry = (GOOD-ENTRIES + BAD-ENTRIES + FUDGE);
int current_entry;        /* Entry we are working on */
```

There's no room at the end of the line. The solution is to put the description on a separate line in front of the variable:

```
int first_entry;     /* First entry to process */
/* Last entry number to process */
int last_entry = (GOOD-ENTRIES + BAD-ENTRIES + FUDGE);
int current_entry;       /* Entry we are working on */
```

But this is still not good enough. This section of code looks like a big gray blob. It's not easy to locate the description for *last_entry.* Adding white space not only breaks up the blob, it also helps group *last_entry's* comment and declaration as shown here:

```
int first_entry;     /* First entry to process */

/* Last entry number to process */
int last_entry = (GOOD-ENTRIES + BAD-ENTRIES + FUDGE);

int current_entry;       /* Entry we are working on */
```

Rule 3-16:

When you can't put a descriptive comment at the end of a variable declaration, put it on a separate line above. Use blank lines to separate the declaration/comment pair from the rest of the code.

Group similar declarations

Repetition and consistency are powerful organizing tools. When declaring variables, group similar variables together and use similar names.

```
int errors_out;    /* Total number of output errors */
int errors_in;     /* Total number of input errors */

int max_out;       /* Max output error rate (errors/hour) */
int max_in;        /* Max input error rate (errors/hour) */

int min_out;       /* Min output error rate (errors/hour) */
int min_in;        /* Min input error rate (errors/hour) */
```

This example uses the prefix *errors_* for the counters that accumulate a running total of the input/output errors. The variables that hold the limits start with the fixes *max_* and *min_*. Common suffixes are also used. All output-related variables end with *_out*, and input variables with *_in*.

Notice that each group of variables consists of two declarations, the first one for the output and the second one for the input.

This example shows only one of several possible groupings. Another possible method is this:

```
int errors_out;    /* Total number of output errors */
int errors_in;     /* Total number of input errors */

int max_out;       /* Max output error rate (errors/hour) */
int max_in;        /* Max input error rate (errors/hour) */

int min_out;       /* Min output error rate (errors/hour) */
int min_in;        /* Min input error rate (errors/hour) */
```

Rule 3-17:

Group similar variables together. When possible, use the same structure for each group.

Hidden Variables

Hidden variables occur when a variable is declared in a global scope and is then declared again in a local scope. The second declaration “hides” the first.

In the following example, the second declaration of location hides the first.

```
/* Bad programming practice */

/* Distance traveled by the car in miles */
float location;

/*..... */
void display_location(void)
{
    /* Location of current cursor */
    int location;   /* *** Hides previous declaration *** */
```

The problem is that we've now used the same word for two different things. Is location a global or a local? Is it a **float** or an **int**? Without knowing which version of the variable is being referred to, we can't answer these questions.

There are enough variable names in the universe that there's no reason to use the same name twice. We could just as easily have used a different name for the second declaration:

```
/* Good programming practice

/* Distance traveled by the car in miles
float car_location;

/*..... */

void display_location(void)
{
    /* Location of current cursor */
    int cursor_location;
```

Rule 3-18:

Don't use hidden variables.

Portable Types

The C compiler runs on many different machines. Making portable programs that can run on all these machines is an art. One trick used to define portable types. For example, Novell uses the type WORD *and DWORD* in all its header files. But what is a *WORD?* Is it 8, 16, or 32 bits? Is it signed or unsigned? You can't tell from the name.

A better set of portable names is:

```
INT16   INT32
UINT16  UINT32
```

These names clearly define the type and size of the data.

Rule 3-19:

Use the names INT16, INT32, UINT16, and UINT32 for portable application

Numbers

C uses a wide variety of numbers, and it's easy to get them confused. Be careful to make numbers clear and unambiguous.

Floating-point numbers

Here are some examples of floating-point numbers:

```
   0.5        .3       6.2          10.
32E4     1e+10         0.333331   5E-5
```

A zero in front of the decimal point is optional. For example, C treats `0.8` and `.8` same. But there is a difference. `.8` looks a lot like the integer `8`, while the number `0.8` is obviously floating-point. Similarly, you should write numbers like `5.` as `5.0`.

Rule 3-20:

Floating-point numbers must have at least one digit on either side f the decimal point.

Large floating-point numbers are written using exponent format. The exponent's “E” can be written in upper or lowercase. Which is better? Well, all digits are full-height characters. The uppercase E is also a full-height character and can easily get lost in a string of digits.

```
321418312354321lEl32809932
```

The E is important and shouldn't get lost. The lowercase is easier to spot:

```
321418312354321lel32809932
```

It's even easier if you always include the sign.

```
321418312354321e+132809932
```

So a lowercase `e` and a sign make these important elements of a floating-point number stand out.

Rule 3-21:

The exponent in a floating-point number must be a lowercase `e`*. This is always followed by a sign.*

Here are some examples of good floating-point numbers:

```
3.1415          3.0        0.5        0.0
1.0e+33    1.0e-333.       33.0       1230.0
```

Hex numbers

C uses the prefix Ox for hexadecimal numbers. A uppercase or lowercase `x` may be used, but as discussed, lowercase letters stand out better.

Rule 3-22:

Start hexadecimal numbers with `0x`. *(Lowercase* `x` *only.)*

Hexadecimal digits include the letters A through F. Again, uppercase or lowercase may be used, so `0XACE` is the same as `0Xace`. Lowercase digits create numbers that are easily confused with variable names. Uppercase digits create numbers that look like constants.

```
0xacde      ace         face        0Xdead
0xACE       X_ACE       BEEF        0xBEEF
```

Numbers are a type of constant, so confusing a number and a constant is not too problematic. Mistaking a number for a variable is worse, so it is preferable to use uppercase digits.

Rule 3-23:

Use uppercase A through F when constructing hexadecimal constants.

Long integers

Long integers end with the letter `L`. Again, C is case insensitive, so lowercase can be used. But lowercase `l` looks a lot like the number `1` and should be avoided. For example, the following two constants look very much alike:

```
34l                 341
```

But when the long integer is written using an uppercase `L`, the confusion clears up:

```
34L                 34l
```

Rule 3-24:

Long constants should end with an uppercase `L`.