The Practice of Programming

Brian W.Kernighan Rob Pike

June 28, 2015

Contents

Preface

The presentation is organized into nine chapters, ecah focusing on one major aspect of programming practice.

Chapter 1 discusses programming style. Good style is so important to good programming that we have chosen to cover it first. Well-written programs are better than badly-written ones, they have fewer errors and are easier to debug and to modify, so it is import to think about style from the beginning. This chapter also introduces an important theme in good programming, the use of idioms(惯用语法) appropriate to the language being used.

Algorithms and data structures, the topics of Chapter **??**, are the core of the computer science curriculum(课程) and a major part of programming courses. Since most readers will already be familiar with this material, our treatment is intended as a brief review of the handful of algorithms and data structures that show up in almost every program. More complex algorithms and data structures usually evolve from these building blocks, so one should master the basics.

Chapter 3 describes the design and implementation of a small program that illustrates algorithm and data structure issues in a realistic setting. The program is implemented in five language; comparing the versions shows how the same data structures are handled in each, and how expressiveness(表现力) and performance vary across a spectrum(系列) languages.

Interfaces between users, programs, and parts of programs are fundamental in programming and much of the success of software is determined by how well interfaces are designed and implemented. Chapter **??** show the evolution of a small library for parsing a widely used data format. Even though the example is small, it illustrates many of the concerns of interfaces degisn: abstraction, information hiding, resource management and error handling.

Much as we try to write a programs correctly the first time, bugs, and therefore debugging are inevitable. Chapter **??** gives strategies and tactics(策略) for systematic and effective debugging. Among the topics are the signatures of common bugs and the importance of "numerology"(命理学), where patterns in debugging often indicates where a problem lies.

Testing is an attempt to develop a reasonable assurance that a program is wording correctly and that it stays correct as it evolves. The emphasis in Chapter 6 is on systematic testing by hand and machine. Boundary condition tests probe at potential weak spots. Mechanization(机械化) and test scaffolds(脚手架) make it easy to do extensive testing with modest effort. Stress tests provide a different kind of testing than typical users do and ferret out(揪出) a different class of bugs.

Computers are so fast and compilers are so good that many programs are fast enough the day they are written. But others are too slow, or they use too much memory, or both. Chapter **??** presents an orderly way to approach the task of making a program use resources efficiently, so that the program remains correct and sound as it is made more efficient.

Chapter **??** covers portability. Successful programs live long enough that their environment changes, or they must be moved to new system or new hardware or new countries. The goal of portability is to reduce the maintenance of a program by minimizing the amount of change neccessary to adapt it to a new environment.

Computing is righ in languages, not just the general-purpose ones that we use for the bulk of programming, but also many specialized languages that focus on narrow domains. Chapter **??** presents serveral examples of the importance of notation in computing, and shows how we can use it to simplify programs, to guide implementations, and even to help us write programs that write programs.

Chapter 1

Style

It is an old observation that the best writers sometimes disregard the rules of rhetoric (修辞学). When they do so, however, the reader will usually find in the sentence some compensating (补偿) merit (优点), attained at the cost of the violation (违反). Unless he is centain of doing as well, he will probaly do best to follow the rules.

William Strunk and E. B. White, *The Elements of Style*

This fragment of code comes from a large program written many years ago:

```
if ((country == SING) || (country == BRNI) ||
    (country == POL) || (country == ITALY))
{
    /*
     * If the country is Singapore, Brunei or Poland
     * then the current time is the answer time
     * rather than the off hook time.
     * Reset answer time and set day of week.
     */
     ...
}
```

It's carefully written, formatted, and commented, and the program it comes from works extremly well; the programmers who create these system are rightly proud of what they built. But this except is puzzling to the casual reader. What relationship links Singapore, Brunei, Poland and Italy? Why isn't Italy mentioned in the comment? Since the comment and the code differ, one of them must be wrong. Maybe both are. The code is what gets executed and tested, so it's more likely to be right; probably the comment didn't get updated when the code did. The comment doesn't say enough about the relationship among the three countries it does mention; if you had to maintain this code, you would need to know more.

The few lines above are typical of much real code: mostly well done, but with some things could be improved.

This book is about te practice of programming -- how to write programs for real. Our purpose is to help you to write software that works at least as well as the program this example was take from, while avoiding trouble spots and weakness. We will talk about writing better code from the beginning and improving it as it evolves.

We are going to start in an unusual place, however, by discussing programming style. The purpose of style is to make the code easy to read for yourself and others, and good style is curcial to good programming. We want to talk about it first so you will be sensitive to it as you read the code in the rest of the book.

These is more to writing a program than getting the syntax right, fixing the bugs, and making it run fast enough. Programs are read not only by computers but also by programmers. A well-written program is easier to understand and to modify than a poorly-written one. The discipline of writing well leads to code that is more like to be correct. Fortunately, this discipline is not hard.

The principles of programming style are based on common sense guided experience, not on arbitrary rules and prescriptions (命令). Code should be clear and simle -- straightforward login, natural expression, conventional language use, meaningful names, neet formatting, helpful comments -- and it should be avoid clever tricks and unusual constructions. Consitency is important because others will find it easier to read your code, and you theirs, if you all stick to the same style. Detail many be imposed by local conventions, management edict (法令), or a program, but even if not, but if even not, it is best to obey a set of widely shared conventions. We follow the style used in the book *The C Programming Language*, with minor adjustments for C++ and Java.

We will often illustrate rules of style by small example of bad and good programming, since the contrast between two ways of saying the same thing is instructive. These example are not artifical. The "bad" ones are all adapted from real code, written by ordinary programmers (occasionally yourselves) working under the common pressures of too much work and too little time. Some will be distilled (提取) for brevity (简短), but they will not be misrepresented (错误的叙述). Then we will rewrite the bad excerpts (摘录) to show how they could be improved. Since they are real code, however, they may exhibit (展现) multiple problems. Addressing every shortcoming would take us too far off topics, so some of the good examples will still harbor (隐藏) other, unremarked flaws (缺点).

To distiguish bad examples from good, throughout the book we will place question marks in the questionable code, as in this real excerpt (摘录):

```
?   #define ONE 1
?   #define TEN 10
?   #define TWENTY  20
```

Why are these `#define`s questionable? Consider the modification that will be necessary if an array of `TWENTY` elements must be made larger. At the very least (至少), each name should be replaced by one that indicates the role of the specific value in the program:

```
#define INPUT_MODE  1
#define INPUT_BUFSIZE   10
#define OUTPUT_BUFSIZE  20
```

1.1 Names

What's in a name? A variable or function name labels an object and conveys information about its purpose. A name should be informative, concise, memorable, and pronounceable if possible. Much information coomes from context and scope; the broader the scope of a variable, the more information should be conveyed by it's name.

Use descriptive names for globals, short names for locals. Global variables, by definition, can crop up (突然出现) anywhere in a program, so they need names long enough and descriptive enough to remind the reader of their meaning. It's also helpful to include a brief comment with the declaration of each global:

```
int npending = 0;   // current length of input queue
```

Global functions, classes and structures should also have descriptive names that suggest their role in a program.

By contrast, shorter names suffice (足够) for local variables; within a function, `n` may be sufficient, `npoints` is fine, and `numberOfPoints` is overkill.

Local variables used in conventional way can have very short names. The use of `i` and `j` for loop indices, `p` and `q` for pointers, and `s` and `t` for strings is so frequent that there is little profit and perhaps some loss in longer names. Compare

```
?   for (theElementIndex = 0; theElementIndex < numberOfElements;
?           theElementIndex++)
?       elementArray[theElementIndex] = theElementIndex;
```

to

```
    for (i = 0; i < nelems; i++)
        elem[i] = i;
```

Programmers are often encouraged to use long variable names regardless of context. That is s mistake: clarity is often achieved through brevity.

There are many naming convensions and local customs. Common ones include using names that begin or end with `p`, such as `nodep`, for pointer; initial capital letters for `Globals`; and all capital for `CONSTANTs`. Some programming shop use more sweeping (彻底的) rules, such as notation to encode type and usage information in the variable, perhaps `pch` to mean a pointer to a character and `strTo` and `strFrom` to mean strings that will be written to and read from. As for the spelling of the names themselves, whether to use `npending` or `numPending` or `num_pending` is a matter of taste; specific rules are much less important than consistent adherence (坚持) to a sensible convention.

Naming conventions make it easier to understand your own code, as well as code written by others. They also make it easier to invent new names as the code is being written. The longer the program, the more important is the choice of good, descriptive, systematic names.

Namespaces in C++ and packages in Java provide ways to manage the scope of names and help to keep meanings clear without unduly (过度的) long names.

Be consistent. Give related things related names that show their relationshipt and highlight their difference.

Besides beging much too long, the member names in this Java class are wildly (鲁莽地) inconsistent:

```
?   class UserQueue {
?       int noOfItemsInQ, frontOfTheQueue, queueCapacity;
?       public int noOfUserInQueue() { ... }
?   }
```

The word "queue" appears as Q, Queue and queue. But since queues can only be accessed from a variable of type `UserQueue`, member names do not need to mention "queue" at all; context suffices, so

```
?   queue.queueCapacity
```

is redundant. This version is better:

```
    class UserQueue {
        int nitems, front, capacity;
        public int nusers() { ... }
    }
```

since it leads to statements like

```
    queue.capacity++;
    n = queue.nusers();
```

No clarity is lost. This example still needs work, however, "items" and "users" are the same thing, so only one term should be used for a single concept.

Use active names for functions. Function names should be based on active verbs, perhaps followed by nouns:

```
    now = data.getTime();
    putchar('\n');
```

Functions that return boolean(true or false) value should be named so that return value is unambiguous. Thus

```
?   if (checkoctal(c)) ...
```

does not indicate which value is true and which is false, while

```
    if (isoctal(c)) ...
```

makes it clear that the function return true if argument is octal and false if not.

Be accurate. A name not only labels, it conveys information to the reader. A misleading names can result in mystifying (模糊的) bugs.

One of us wrote and distributed for years a macro called `isoctal` with this incorrect implementation:

```
?   #define isoctal(c) ((c) >= '0' && (c) <= '8')
```

instead of the proper

```
    #define isoctal(c) ((c) >= '0' && (c) <= '7')
```

In this case, the name conveyed the correct intent but the implementation was wrong; it's easy for a sensible name to disguise a broken implementation.

Here is an example in which the name and the code are in complete contradiction (矛盾):

```
?   public boolean inTable(Object obj) {
?       int j = this.getIndex(obj);
?       return(j == nTable);
?   }
```

The function `getIndex` returns a value between zero and `nTable-1` if it finds the object, and returns `nTable` if not. The boolean value returned by `inTable` is thus the oppsite of what the name implies. At the time code is written, this might not cause trouble, but it the program is modified later, perhaps by a different programmer, the name is sure to confuse.

Exercise1-1 . Comment on the choice of names and values in the following code.

```
?   #define TRUE    0
?   #define FALSE   1
?
?   if ((ch = getchar()) == EOF)
?       not_eof = FALSE;
```

□

Exercise1-2 . Improve this function:

```
?   int smaller(char *s, char *t) {
?       if (strcmp(s, t) < 1)
?           return 1;
?       else
?           return 0;
?   }
```

□

Exercise1-3 . Read this code aloud:

```
?   if ((falloc(SMRHSHSCRTCH, S_IFEXT|0644, MAXRODDHSH)) < 0)
?       ...
```

□

1.2 Expression and Statements

By analogy (类比) with choosing names to aid the reader's understanding, write expressions and statements in a way that makes their meaning as transport as possible. Write the clearest code that does the job. Use spaces around operators to suggest grouping; more generally, format to help readability. This is trivial (琐细的) but valuable, like keeping a neat dest so you can find things. Unlike your dest, your programs are likely bo be examined by others.

Indent to show structure. A consistent indentation style is the lowest-energy way to make a program's structure self-evident (不言自明的). This example is badly formatted:

```
?   for(n++;n<100;field[n++]='\0');
?   *i = '\0'; return('\n');
```

Reformatting improves it somewhat:

```
?   for (n++; n < 100; field[n++] = '\0')
?       ;
?   *i = '\0';
?   return('\n');
```

Even better is to put the assignment in the body and separate the increment, so the loop takes a more conventional form and is thus easier to grasp (抓取):

```
    for (n++; n < 100; n++)
        field[n] = '\0';
    *i = '\0';
    return '\n';
```

Use the natural form for expressions. Write expressions as you might speak them aloud. Conditional expressions that include negations are always hard to understand:

```
?   if (!(block-id < actblks) || !(block-id >= unblocks))
?       ...
```

Each test is stated negatively, though there is no need for either to be. Turing the relations around lets us state the tests positively:

```
    if ((block-id >= actblks) || (block - id < unblocks))
        ...
```

Node the code reads naturally. *Parenthesize to resolves ambiguity.* Parentheses specify grouping and can be used to make the intent clear event when they are not required. The inner parentheses in the previous example are not necessary, but they don't hurt, either. Seasoned (经验丰富的) programmers might omit them, because the relational operators(`< <= == != >= >`) have higher precedence than the locgical operators(`&&` and `||`).

When mixing unrelated oprators, though, it's a good idea to parenthesize. C and it's friends present pernicious (恶劣的) precedence problems, and it's easy to make a mistake. Because the logical operators bind tighter than assignment, parentheses are mandatory for most expressions that combine them:

```
    while ((c = getchar()) != EOF)
        ...
```

The bitwise operators `&` and `|` have lower precedence than relational operators like ==, so despite it's appearance,

```
?   if (x&MASK == BITS)
?       ...
```

actually means

```
?   if (x & (MASK==BITS))
?       ...
```

which is centainly not the programmer's intent. Because it combines bitwise and relational operators, the expression need parentheses:

```
    if ((x&MASK) == BITS)
        ...
```

Even if parentheses aren't necessary, they can help if the grouping is hard to grasp at first glance. This code doesn't need parentheses:

```
?   leap_year = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
```

but they make it easier to understand:

```
    leap_year = ((y%4 == 0) && (y%100 != 0)) || (y%400 == 0);
```

We also removed some of the blanks: grouping the operands of higher-precendence operators helps the readers to see the structure more quickly.

Break up complex expressions. C, C++ and Java have rich expression syntax and operators, and it's easy to get carried away by cramming (塞满) everything into one construction. An expression like the following is compact (紧凑的) but it packs too many oprations into a single statement:

```
?   *x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

It's easier to grasp when broken into several pieces:

```
    if (2*k < n-m)
        *xp = c[k+1];
    else
        *xp = d[k--];
    *x += *xp;
```

Be clear. Programmers' endless creative energy is sometimes used to write the most concise code possible, or to find clever ways to achieve a result. Sometimes these skills are misapplied, though, since the goal is to write clear code, not clever code.

What does this intricate (复杂的) calculation do?

```
?   subkey = subkey >> (bitoff - ((bitoff >> 3) << 3));
```

The innermost expression shifts `bitoff` three bits to the right. The result is shifted left again, thus replacing the three shifted bits by zeros. This result in turn is subtracted from the original value, yielding the bottom three bits of `bitoff`. These three bits are used to shift `subkey` to the right.

Thus the original expression is equivalent to

```
    subkey = subkey >> (bitoff & 0x7);
```

It takes a while to puzzle out what the first version is doing; the second is shorter and clearer. Experienced programmers make it ever shorter by using assignment operator:

```
    subkey >>= bitoff & 0x7;
```

Some constructs seem to invite abuse. The `?:` operator can lead to mysterious code:

```
?   child=(!LC&&!RC)?0:(!LC?RC:LC);
```

It's almost impossible to figure out what this code without following all the possible paths through the expression. This form is longer, but much easier to follow because it makes the paths explicit:

```
if (LC == 0 && RC == 0)
    child = 0;
else if (LC == 0)
    child = RC;
else
    child = LC;
```

The `?:` operator is fine for short expressions where it can replace four lines of if-else with one, as in

```
max = (a > b) ? a : b;
```

or perhaps

```
printf("The list has %d item%s\n", n, n==1 ? "" : "s");
```

but it is not a general replacement for conditional statements.

Be careful with side effect. Operators like ++ have side effects: besides returning a value, they also modify an underlying variable. Side effects can be extremely convenient, but they can also cause trouble because the actions of retrieving the value and updating the variable might not happen at the same time. In C and C++, the order of execution of side effects is undefined, so this multiple assignment is likely to produce the wrong answer:

```
?   str[i++] = str[i++] = ' ';
```

The intent is to store blanks at the next two position in `str`. But depending on when `i` is updated, a position in `str` could be skipped and `i` might end up increased only by 1. Break it into two statements:

```
str[i++] = ' ';
str[i++] = ' ';
```

Even though it cantains only one increment, this assignment can also give varying results:

```
?   array[i++] = i;
```

If `i` is initially 3, the array element might be set to 3 or 4.

It's not just increment and decrement that have side effects; I/O is another source of behind-the-scenes (幕后的) action. This example is an attempt to read two related numbers from standard input:

```
?   scanf("%d %d", &yr, &profit[yr]);
```

It is broken because part of the expression modifies `yr` and another part uses it. The value of `profit[yr]` can never be right unless the new value of `yr` is the same as the old one. You might think that the answer depends on the order in which the arguments are evaluated, but the real issue is that all the arguments to `scanf` are evaluated before the routine is called, so `&profit[yr]` will always be evaluated using the old value of `yr`. This sort of problem can occur in almost any language. The fix is, as usual, to break up the expression:

```
scanf("%d", &yr);
scanf("%d", &profit[yr]);
```

Exercises cantion in any expression with side effects.

Exercise1-4 . Improves each of these fragments:

```
?   if (!(c == 'y' || c == 'Y'))
?       return;
?   length = (length < BUFSIZE) ? length : BUFSIZE;
?   flag = flag ? 0 : 1;
?   quote = (*line == '"') ? 1 : 0;
?   if (val & 1)
?       bit = 1;
?   else
?       bit = 0;
```

□

Exercise1-5 . What is wrong with this excerpt (摘录)?

```
?   int read(int *ip) {
?       scanf("%d", ip);
?       return *ip;
?   }
?       ...
?   insert(&graph[vert], read(&val), read(&ch));
```

□

Exercise1-6 . List all the different output this could produce with various orders of evaluation:

```
?   n = 1;
?   printf("%d %d\n", n++, n++);
```

Try it on as many compilers as you can, to see what happens in practice. □

1.3 Consistency and Idioms (习惯用法)

Consitency leads to better programs. If formatting varies unpredictably, or loop over an array runs upill this time and downhill the next, or strings are copied with `strcpy` and a `for` loop there, the variations make it harder to see what's really going on. But if the same computation is done the same way every time it appears, any variation suggests a genuine (真正的) difference, one worth nothing.

Use a consistent indentation and brace style. Indentation show structure, but which indentation style is best? Should the opening brace go one the same line as the `if` for on the next? Programmers have always argued about the layout of program, but the specific style is much less important than its consistent application. Pick one style, preferably (更合意地) ours, use it consistently, and don't waste time arguing.

Should you include braces even when they are not needed? Like parentheses, braces can resolve ambiguity and occasionally make the code clearer. For consistency, many experienced programmers always put braces around loop or `if` bodies. But if the body is a single statement they are unnecessary, so we tend to omit them. If you also choose to leave them out, make sure you don't drop them when they are needed to resolve the "dangling (悬挂的) else" ambiguity exemplified (示例) by this excerpt (摘录):

```
?   if (month == FEB) {
?       if (year%4 == 0)
?           if (day > 29)
?               legal = FALSE;
?       else
?           if (day > 28)
?               legal = FALSE;
?   }
```

The indentation is misleading, since the `else` is actually attached to the line

```
?   if (day > 29)
```

and the code is wrong. When one `if` immediately follows another, always use braces:

```
?   if (month == FEB) {
?       if (year%4 == 0) {
?           if (day > 29)
?               legal = FALSE;
?       } else {
?           if (day > 28)
?               legal = FALSE;
?       }
?   }
```

Syntax-driven tools make this sort of mistake less likely.

Even with the bug fixed, though, the code is hard to follow. The computation is easier to grasp if we use a variable to hold the number of days in February:

```
?   if (month == FEB) {
?       int nday;
?
?       nday = 28;
?       if (year%4 == 0)
?           nday == 29;
?       if (day > nday)
?           legal = FALSE;
?   }
```

The code still wrong -- 2000 is a leap year, while 1900 and 2100 are not -- but this structure is much easier to adapt to make it absolutely right.

By the way, if you work on a program you didn't write, preserve the style you find there. When you make a change, don't use your own style even though you prefer it. The program's consistency is more important than your own, because it makes life easier for those who follow.

Use idioms for consistency. Like natural language, programming languages have idioms, conventional ways that exprerienced programmers write common pieces of code. A central part of learning any language is developing a familiarity with its idioms.

One of the most common idioms is the form of a loop. Consider the C, C++, or Java code for stepping through the `n` elements of an array, for example to initialize them. Someone might write the loop like this:

```
?   i = 0;
?   while (i <= n-1)
?       array[i++] = 1.0;
```

or perhaps like this:

```
?   for (i = 0; i < n; )
?       array[i++] = 1.0;
```

or even:

```
?   for (i = 0; --i >= 0; )
?       array[i] = 1.0;
```

All of these are correct, but the idiomatic form is like this:

```
    for (i = 0; i < n; i++)
        array[i] = 1.0;
```

This is not an arbitrary choice. It visits ecah member of an n-element array indexed from `0` to `n-1`. It places all the loop control in the `for` itself, runs in increasing order, and uses the very idiomatic `++` operator to undate the loop variable. It leaves the index variable at a known value just beyond the last array element. Native speakers recognize it which study and write it correctly without a moment's thought.

In C++ or Java, a common variant includes the declaration of the loop variable:

```
    for (int i = 0; i < n; i++)
        array[i] = 1.0;
```

Here is the standard loop for walking along a list in C:

```
    for (p = list; p != NULL; p = p->next)
        ...
```

Again, all the loop control is in the `for`.

For an infinite loop, we prefer

```
for (;;)
    ...
```

but

```
while (1)
    ...
```

is also popular. Don't use anything other than these forms.

Indentation should be idiomatic, too. This unusual vertical layout detracts (贬低) from readability; it looks like three statements, not a loop:

```
?   for (
?       ap = arr;
?       ap < arr + 128;
?       *ap++ = 0
?       )
?   {
?       ;
?   }
```

A standard loop is much easier to read:

```
for (ap = arr; ap < arr+128; ap++)
    *ap = 0;
```

Sprawling (蔓延的) layout also force code onto multiple screens or pages, and thus detract from readability.

Another common idiom is to nest an assignment inside a loop condition, as in

```
while ((c = getchar()) != EOF)
    putchar(c);
```

The `do-while` statement is used much less often than `for` and `while`, because it always executes at least once, testing at the bottom of the loop instead of the top. In many cases, that behavior is a bug waiting to bite, as in this rewrite of the `getchar` loop:

```
?   do {
?       c = getchar();
?       putchar(c);
?   } while (c != EOF);
```

It write a spurious (假的) output character because the test occurs after the call to `putchar`. The `do-while` loop is the right one only when the body of the loop must always be executed at least once; we'll see some examples later.

One advantage of the consistent use of idioms is that it draws attentions to non-standard loops, a frequent sign of trouble:

```
int i, *iArray, nmemb;

iArray = malloc(nmemb * sizeof(int));
for (i = 0; i <= nmemb; i++)
    iArray[i] = i;
```

Space is allocated for `nmemb` items, `iArray[0]` through `iArray[nmemb-1]`, but since the loop test is <= the loop walks off the end of array and overwrites whatever is stored next in memory. Unfortunately, error like this are often not detected until long after the damage has been done.

C and C++ also have idioms for allocating space for strings and then manipulating it, and code that doesn't use them often harbors (隐藏) a bug:

```
?   char *p, buf[256];
?
?   gets(buf);
?   p = malloc(strlen(buf));
?   strcpy(p, buf);
```

One should never use `gets`, since there is no way to limit the amount of input it will read. This leads to security problems that we'll return to in Chapter 6, where we will show that `fgets` is always a better choice. But there is another problem as well: `strlen` does not count the `'\0'` that terminates a string, while `strcpy` copies it. So not enough space is allocated, and `strcpy` writes past the end of allocated space. The idiom is

```
p = malloc(strlen(buf) + 1);
strcpy(p, buf);
```

or

```
p = new char[strlen(buf) + 1];
strcpy(p, buf);
```

in C++. If you don't see the `+1`, bewary.

Java doesn't suffer from this specific problem, since string are not prepresented as null-terminated arrays. Array substripts are checked as well, so it is not possible to access outside the bounds of an array in Java.

Most C and C++ environments provide a library function, `strdup`, that creates a copy of a string using `malloc` and `strcpy`, making it easy to avoid this bug. Unfortunately, `strdup` is not part of the ANSI C standard.

By the way, neither the original code nor the corrected version check the value returned by `malloc`. We omitted this improvement to focus on the main point, but in a real program the return value from `malloc`, `realloc`, `strdup`, or any other allocation routine should always be checked.

Use else-ifs for multi-way decisions. Multi-way decisions are idiomatically expressed as a chain of `if .. else if .. else`, like this:

```
if (condition1)
    statement1
else if (condition2)
    statement2
...
else if (conditioni)
    statementi
else
    default-statement
```

The *conditions* read from top to bottom; at the first *condition* that is satisfied, the *statement* that follows is executed, and then rest of the construct is skipped. The *statement* part may be a single statement or group of statements enclosed in braces. The last `else` handles the "default" situation, where none of the other alternatives was chosen. This trailing (拖尾的) `else` part may be ommited if there is no action for the default, although leaving it with an error message may help to catch conditions that "can't happen".

Align all of the `else` clauses (子句) vertically rather than lining up (排列) each `else` with the corresponding `if`. Vertial alignment emphasizes that test are done in sequence and keep them from marching off (步进, 越来越近) the right side of the page.

A sequence of nested `if` statement is often a warning of awkward (笨拙的) code, if not outright (显示) errors:

```
?   if (argc == 3)
?       if ((find == fopen(argv[1], "r")) != NULL)
?           if ((fout = fopen(argv[2], "w")) != NULL) {
?               while ((c = getc(find)) != EOF)
```

```
?                   putc(c, fout);
?               fclose(fin); fclose(fout);
?           } else
?               printf("Can't open output file %s\n", argv[1]);
?       else
?           printf("Can't open input file %s\n", argv[1]);
?   else
?       printf("Usage: cp inputfile outputfile\n");
```

The sequence of `if`s requires us to maintain a mental (心理的) pushdown (推进) stack of what tests were make, so that at the appropriate point we can pop them until we determine the corresponding action (if we can still remember). Since at most one action will be performed, we really want an `else if`. Changing the order in which the decisions are made leads to a clearer version, in which we have also corrected the resource leak (泄漏) in the original:

```
if (argc != 3)
    printf("Usage: cp inputfile outputfile\n");
else if ((fin = fopen(argv[1], "r")) == NULL)
    printf("Can't open input file %s\n", argv[1]);
else if ((fout = fopen(argv[2], "w")) == NULL) {
    printf("Can't open output file %s\n", argv[2]);
    fclose(fin);
} else {
    while ((c = getc(fin)) != EOF)
        putc(c, fout);
    fclose(fin);
    fclose(fout);
}
```

We read down the tests until the first one that is true, do the corresponding action, and continue after the last `else`. The rule is to follow each decision as closely as possible by it's associated action. Or, to put it another way, each time you make a test, do something.

Attempts to re-use pieces of code often lead to tightly knotted (纠缠的) programs.

```
?   switch (c) {
?   case '-': sign = -1;
?   case '+': c = getchar();
?   case '.': break;
?   default: if (!isdigit(c))
?               return 0;
?   }
```

This uses a tricky sequence of fall-throughs in the `switch` statement to avoid duplicating one line of code. It's also not idiomatic; cases should almost always end with a `break`, with the rare exceptions commented. A more traditional layout and structure is easier to read, though longer:

```
?   switch (c) {
?   case '-':
?       sign = -1;
?       /* fall through */
?   case '+':
?       c = getchar();
?       break;
?   case '.':
?       break;
?   default:
?       if (!isdigit(c))
?           return 0;
?       break;
?   }
```

The increase in size is more than offset by the increase in clarity. However, for such an unusual structure, a sequence of `else-if` is even clearer:

```
if (c == '-') {
    sign = -1;
    c = getchar();
} else if (c == '+') {
    c = getchar();
} else if (c != '.' && !isdigit(c)) {
    return 0;
}
```

The braces around the one-line blocks highlight the parallel structure.

An acceptable use of a fall-through occurs when several cases have identical code; the conventional layout is like this:

```
case '0':
case '1':
case '2':
    ...
    break;
```

and no comment is required.

Exercise1-7 . Rewrite these C/C++ excerpts more clearly:

```
?       if (istty(stdin)) ;
?       else if (istty(stdout)) ;
?           else if (istty(stderr)) ;
?               else return(0);

?       if (retval != SUCCESS)
?           return(retval);
?       /* All went well */
?       return SUCCESS;

?       for (k = 0; k++ < 5; x += dx)
?           scanf("%lf", %dx);
```

□

Exercise1-8 . Identify the errors in this Java fragment and repair it by rewriting with an idiomatic loop:

```
?       int count = 0;
?       while (count < total) {
?           count++;
?           if (this.getName(count) == nametable.userName()) {
?               return (true);
?           }
?       }
```

□

1.4 Function Macros

There is a tendency among older C programmers to write macros instead of function for very short computations that will be executed frequently; I/O operations such as `getchar` and character test like `isdigit` are officially sanctioned (认可的) examples. The reason is performance: a macro avoids the overhead of a function call. This argument was weak even when C was first defined, a time of slow machines and expensive

function calls; today it is irrelevant. With modern machines and compilers, the drawbacks of function macros outweight (超过) their benefits.

Avoid function macros. In C++, `inline` functions render (补偿) function macros unnecessary; in Java, there are no macros. In C, they cause more problems than they solve.

One of the most serious problem with function macros is that a *parameter* that appears more than once in the definition might be evaluated more than once; if the argument in the call includes an expression with side effect, the result is a subtle (微妙的) bug. This code attempts to implement one of the character tests from `<ctype.h>`:

```
?   #define isupper(c) ((c) >= 'A' && (c) <= 'Z')
```

Note that the parameter `c` occurs twice in the body of the macro. If `isupper` is called in a context like this,

```
?   while (isupper(c = getchar()))
?       ...
```

then each time an input character is greater than or equal to `A`, it will be discarded and another character read to be tested against `Z`. The C standard is carefully written to permit `isupper` and analogous functions to be macros, but only if they gurantee to evaluate the argument only once, so this implementation is broken.

It's always better to use the ctype function than to implement them yourself, and it's safer not to nest routine like `getchar` that have side effects. Rewriting the test to use two expressions rather one makes it clearer and also gives an opportunity to catch end-of-file explicitly:

```
    while ((c = getchar()) != EOF && isupper(c))
        ...
```

Sometimes multiple evaluation causes a performance problem rather than an outright error. Consider this example:

```
?   #define ROUND_TO_INT(x) ((int) ((x)+(((x)>0)?0.5:-0.5)))
?       ...
?   size = ROUND_TO_INT(sqrt(dx*dx + dy*dy));
```

This will perform the square root computation twice as often as necessary. Even given simple arguments, a complex expression like the body of `ROUND_TO_INT` translates into many instructions, which should be housed (收藏) in a single function to be called when needed. Instantiating a macro at every occurence makes the compiled program larger. (C++ `inline` functions have this drawback, too.)

Parenthesize the macro body and arguments. If you insist on using function macros, be careful. Macros work by textual substitution: the parameters in the definition are replaced by the arguments of the call and the result replaces the original call, as text. This is a troublesome difference from function. This expression

```
    1 / square(x)
```

works fine if square is a function, but if it's a macro like this,

```
?   #define square(x) (x) * (x)
```

the expression will be expanded to the erroneous

```
?   1 / (x) * (x)
```

The macro should be rewritten as

```
    #define square(x) ((x) * (x))
```

All those parentheses are necessary. Even parenthesizing the macro properly does not addres the multiple evaluation problem. If an operation is expensive or common enough to be wrapped up, use a function.

In C++, `inline` functions avoid the syntatic trouble while offering whatever performance advantage macros might provide. They are appropriate for short functions that set or retrieve a single value.

Exercise1-9 . Identify the problems with this macro definition:

```
?       #define ISDIGIT(c) ((c >= '0') && (c <= '9')) 1 : 0
```

□

1.5 Magic Numbers

Magic numbers are the constants, array sizes, character positions, conversion factors, and other literal numeric values that appear in programs.

Give names to magic numbers. As a guideline, any number other than 0 or 1 is likely to be magic and should have a name of its own. A raw number in program source gives no indication of its importance or derivation (来历), making the program harder to understand and modify. This excerpt from a program to print a histogram of letter frequencies on a 24 by 80 cirsor-addressed terminal is needlessly opaque (不透明的) because of a host of magic numbers:

```
?   fac = lim / 20;     /* set scale factor */
?   if (fac < 1)
?       fac = 1;
?
?                       /* generate histogram */
?   for (i = 0, col = 0; i < 27; i++, j++) {
?       col += 3;
?       k = 21 - (let[i] / fac);
?       star = (let[i] == 0) ? ' ' : '*';
?       for (j = k; j < 22; j++)
?           draw(j, col, star);
?   }
?   draw(23, 2, ' '); /* label x axis */
?   for (i = 'A'; i <= 'Z'; i++)
?       printf("%c ", i);
```

The code includes, among othres, the number 20, 21, 22, 23. They're clearly related... or are they? In fact, there are only three numbers critical to this program: 24, the number of rows on the screen; 80, the number of column; and 26, the number of letters in the alphabet. But none of these appears in the code, which makes the numbers that do even more magical.

By giving names to the principal (主要的) numbers in the calculation, we can make the code easier to follow. We discover, for instance, that the number 3 comes from (80-1)/26 and that `let` should have 26 entries, not 27 (an off-by-one error perhaps caused by 1-indexed screen coordinates). Mking a couple of other simplifications, this is the result:

```
enum {
    MINROW  = 1,                        /* top edge */
    MINCOL  = 1,                        /* left edge */
    MAXROW  = 24,                       /* bottom edge (<=) */
    MAXCOL  = 80,                       /* right edge (<=) */
    LABELROW = 1,                       /* position of labels */
    NLET    = 26,                       /* size of alphabet */
    HEIGHT  = MAXROW - 4,               /* height of bars */
    WIDTH   = (MAXCOL-1)/NLET,          /* width of bars */
};

...
fac = (lim + HEIGHT - 1) / HEIGHT;  /* set scale factor */
if (fac < 1)
    fac = 1;
for (i = 0; i < NLET; i++) { /* generate histogram */
```

```
        if (let[i] == 0)
            continue;
        for (j = HEIGHT - let[i]/fac; j < HEIGHT; j++)
            draw(j+1+LABELROW, (i+1)*WIDTH, '*');
    }
    draw(MAXROW-1, MINCOL+1, ' '); /* label x axis */
    for (i = 'A'; i <= 'Z'; i++)
        printf("%c ", i);
```

Now it's clearer what the main loop does: it's an idiomatic loop from 0 to `NLET`, indicating that the loop is over the elements of the data. Also the calls to `draw` are easier to understand because words like `MAXROW` and `MINCOL` remind us of the order of arguments. Most important, it's now feasible to adapt the program to another size of display or different data. The numbers are demystified (易懂的) and so is the code.

Define numbers as constants, not macros. C programmers have traditional used `#define` to manage magic number values. Te C preprocesser is a powerful but blunt (愚蠢的) tool, however, and macros are a dangerous way to program because they change the lexical structure of the program underfoot (碍事的). Let the language proper do the work. In C and C++, integer constants can be defined with an `enum` statement, as we say in the previous example. Constants of any type can be declared with `const` in C++:

```
const int MAXROW = 24, MAXCOL = 80;
```

or `final` in Java:

```
static final int MAXROW = 24, MAXCOL = 80;
```

C also has `const` values but they cannot be used as array bounds, so the `enum` statement remains the method of choice in C.

Use character constants, not integers. The functions in `<ctype.h>` or their equivalent should be used to test the properties of character. A test like this:

```
?   if (c >= 65 && c <= 90)
?       ...
```

depends completely on a particular character representation. It's better to use

```
?   if (c >= 'A' && c <= 'Z')
?       ...
```

but that may not have the desired effect if the letters are not contiguous in the character set encoding or if the alphabet include other letters. Best is to use the library:

```
    if (isupper(c))
        ...
```

in C or C++, or

```
    if (Character.isUpperCase(c))
        ...
```

in Java.

A related issue is that the number 0 appears often in programs, in many contexts. The compiler will convert the number into the appropriate type, but it helps the reader to understand the role of each 0 if the type is explicit. For example, use `(void *)0` or `NULL` to represent a zero pointer in C, and `'\0'` instead of `0` to represent the null byte at the end of a string. In other words, don't write

```
?   str = 0;
?   name[i] = 0;
?   x = 0;
```

but rather:

```
str = NULL;
name[i] = '\0';
x = 0.0;
```

We prefer to use different explicit constants, reserving 0 for a literal integer zero, because they indicate the use of the value and thus provide a bit of documentation. In C++, however, `0` rather than `NULL` is the acceptable notation for a null pointer. Java solves the problem best by defining the keyword `null` for an object reference that doesn't refer to anything.

Use the language to calculate the size of an object. Don't use an explicit size for any data type; use `sizeof(int)` instead of `2` or `4`, for instance. For similar reasons, `sizeof(array[0])` may be better than `sizeof(int)` because it's one less thing to change if the type of the array changes.

The `sizeof` operator is sometimes a convenient way to avoid inventing names for the numbers that determine array size. For example, if we write

```
char    buf[1024];
fgets(buf, sizeof(buf), stdin);
```

the buffer size is still a magic number, but it occurs only once, in the declaration. It may not be worth inventing a name for the size of a local array, but it is definitely worth writing code that does not have to change if the size or type changes.

Java arrays have `length` that gives the numbers of elements:

```
char buf[] = new char[1024];
for (int i = 0; i < buf.length; i++)
    ...
```

There is no equivalent of `.length` in C and c++, but for an array (not a pointer) whose declaration is visible, this macro computes the number of elements in the array:

```
#define NELEMS(array) (sizeof(array) / sizeof(array[0]))

double dbuf[100];
for (i = 0; i < NELEMS(dbuf); i++)
    ...
```

The array size is set in only one place; the rest of the code not change if the size does. There is no problem with multiple evaluation of the macro argument here, since there can be no side effects, and in fact the computation is done as the program is being compiled. This is an appropriate use for a macro because it does something that a function cannot: compute the size of an array from its declaration.

Exercise1-10 . How would you rewrite these definitions to minimize potential errors?

```
?       #define FTZMETER    0.3048
?       #define METERZFT    328084
?       #define MIZFT       5280.0
?       #define MIZKM       1.609344
?       #define SQMIZSQKM   2.589988
```

□

1.6 Comments

Comments are meant to help the reader of a program. They do not help by saying things the code already plainly (清楚地) says, or by contradicting (矛盾) the code, or by distracting (分心) the reader with elaborate

(详细说明) typographical (印刷的) displays. The best comments aid the understanding of a program by briefly pointing out salient (突出的) details or by providing a larger-scale view of the proceedings (进程, 进行).

Don't belabor (痛打) the obvious. Comments shouldn't report self-evident information, such as the fact that `i++` increments `i`. Here are some of our favorite worthless comments:

```
?   /*
?    * default
?    */
?   default:
?       break;

?   /* return SUCCESS */
?   return SUCCESS;

?   zerocount++;    /* increment zero entry counter */

?   /* initialize "total" to "number_received" */
?   node->total = node->number_received;
```

All of these comments should be deleted; they're just clutter (杂物).

Comments sould add somthing that is not immediately evident from the code, or collect into one place information that is spread through the source. When something subtle (微妙的) is happening, a comment may clarify, but if the actions are obvious already, restating them in words is pointless:

```
?   while ((c = getchar()) != EOF && isspace(c))
?       ;   /* skip white space */
?   if (c == EOF)   /* end of file */
?       type = endoffile;
?   else if (c == '(')  /* left parenthes */
?       type = leftparen;
?   else if (c == ';')  /* semicolon */
?       type = semicolon;
?   else if (isdigit(c))    /* number */
?       ...
```

These comments should also be deleted, since the well-chosen names already convey the information.

Comment functions and global data. Comments can be useful, of course. We comment functions, global variables, constant definitions, fields in structures and classes, and anything else where a brief summary can aid understanding.

Global variables have a tendency to crop up (突然出现) intermittently (间歇地) throughout a program; a comment serves as a reminder to be referred to as needed. Here's an example from a program in Chapter 3 of this book:

```
struct State { /* prefix + suffix list */
    char    *pref[NPREF];   /* prefix words */
    Suffix  *suf;           /* list of suffixes */
    State   *next;          /* next in hash table */
};
```

A comment that introduces each function set the stage for reading the code itself. If the code isn't too long or technical, a single line is enough:

```
// random: retrn an integer in the range [0, r-1].
int random(int r)
{
    return (int)(Math.floor(Math.random() * r));
}
```

Sometimes coe is genuinely (真正地) difficult, perhaps because the algorithm is complicated or the data structure are intricate (复杂的). In that case, a comment that points to a source of understanding can aid the reader. It may also be valuable to suggest why particular dicisions were made. This comment introduces an extremly efficient implementation of an inverse (反转) discrete (离散) cosine transform (DCT) used in JPEG image decoder:

```
/*
 * idct: Scaled integer implementation of
 * inverse two dimentional 8*8 Discrete Cosine Transform,
 * Chen-Wang algorithm (IEEE ASSP-32, pp 803-816, Aug 1984)
 *
 * 32-bit integer arithmetic (8-bit coefficents)
 * 11 multiplies, 29 adds per DCT
 *
 * coefficents extended to 12 bits for IEEE 1180-1990 compliance.
 */
 static void dict(int b[8*8])
 {
     ...
 }
```

This helpful comments cites the reference, briefly describes the data used, indicates the performance of the algorithm, and tells how and why the original algorithm has been modified.

Don't comment bad code, rewrite it. Comment anything unusual or potentially confusing, but when the comment outweight the code, the code probably needs fixing. This example uses a long, muddled (胡乱对付的) comment and a conditional-compiled debugging print statement to explain to a single statement:

```
?    /* if "result" is 0, a match was found, so return true.
?     * otherwise, "result" is non-zero, so return false
?     */
?    #ifdef DEBUG
?    printf("*** isword returns !result = %d\n", !result);
?    fflush(stdout);
?    #endif
?
?    return(!result);
```

Negation is hard to understand and should be avoided. Part of the problem is the uninformative (无法提供信息的) variable name, `result`. A more descriptive name, `matchfound`, makes the comment unnecessary and cleans up the print statement, too.

```
#ifdef DEBUG
printf("*** isword returns matchfound = %d\n", matchfound);
fflush(stdout);
#endif

return(matchfound);
```

Don't contradict the code. Most comments argee with code when they are written, but as bugs are fixed and the program evolves, the comments are foten left in their original form, resulting in disagremment with the code. This is the likely explanation for the inconsistentency in the example that opens this chapter.

Whatever the source of the disagremment, a comment that contradicts the code is confusing, and many a debugging session has been needlessly protracted (拖延) because a mistaken comment was taken as truth. When you change the code, make sure the comments are stil accurate.

Comments should not only agree with code, they should support it. The comment in this example is correct -- it explains the purpose of the next two lines -- but it appears to contradict the code; the comment talks aboue newline and the code talks about blanks:

```
?   time(&now);
?   strcpy(date, ctime(&now));
?   /* get rid of trailing newline character copied from ctime */
?   i = 0;
?   while (date[i] != ' ') i++;
?   date[i] = 0;
```

One improvement is to rewrite the code more idiomatically:

```
?   time(&now);
?   strcpy(date, ctime(&now));
?   /* get rid of trailing newline character copied from ctime */
?   i = 0;
?   while (date[i] != '\n') i++;
?   date[i] = '\0';
```

Code and comment now agree, but both can be improved by beging made more direct. The problem is to delete the newline that `ctime` puts on the end of the string it returns. The comment should say so, and the code should do so:

```
    time(&now);
    strcpy(date, ctime(&now));
    /* ctime() puts newline at end of string, delete it */
    date[strlen(data) - 1] = '\0';
```

This last expression is the C idiom for removing the last character from a string. The code is now short, idiomatic and clear, and the comment support it by explaining why it needs to be there.

Clarify, don't confuse. Comments are supposed to help readers over the hard parts, not create more obstacles. This example follows our guidelines of commenting the function and explaining unusual properties; on other hand, the function is `strcmp` and the unusual properties are peripheral(无关紧要的) to the job at hand, which is the implementation of a standard and familiar interface:

```
?   int strcmp(char *s1, char *s2)
?   /* string comparison routine returns -1 if s1 is
?    * above s2 in an ascending order list, 0 if equal,
?    * 1 if s1 below s2
?    */
?   {
?       while (*s1 == *s2) {
?           if (*s1 == '\0') return(0);
?           s1++;
?           s2++;
?       }
?       if (*s1 > *s2) return(1);
?       return(-1);
?   }
```

When it takes more than a few words to explain what's happening,it's often and indication that the code should be rewriting. Here, the code could perhaps be improved but the real problem is the comment, which is nearly as long as the implementation ans confusing. too (which way is "above"?). We're stretching (伸展) the point to say this routine is hard to understand, but since it implements a standard function, its comment can help by summarizing the behavior and telling us where the definition originates; that's all that's needed:

```
    /*
     * strcmp: return <0 if s1 < s2, > 0 if s1 > s2, 0 if equal.
     * ANSI C, section 4.11.4.2
     */
    int strcmp(const char *s1, const char *s2)
    {
        ...
    }
```

Students are taught that it's important to comment everything. Professional programmers are foten require to comment all their code. But the purpose of commenting can be lost in blindly following rules. Comments are meant to help a reader understand pans (全景) of the program that are not readily understood from the code itself. As much as possible, write code that is easy to understand; the better you do this, the fewer comments you need. Good code needs fewer comments than bad code.

Exercise1-11 . Comment on these comments.

```
?       void dict::insert(string& w)
?       // returns 1 if w in dictionary, otherwise returns 0.

?       if (n > MAX || n % 2 > 0) // test for even number

?       // write a message
?       // Add to line counter for each line written
?       void write_message()
?       {
?           // increment line counter
?           line_number = line_number + 1;
?           fprintf(fout, "%d %s\n%d %s\n%d %s\n",
?               line_number, HEADER,
?               line_number + 1, BODY,
?               line_number + 2, TRAILER);
?           // increment line counter
?           line_number = line_number + 2;
?       }
```

□

1.7 Why Bother?

In this chapter, we've talked about the main concerns of programming style: descriptive names, clarity in expressions, straightforward control flow, readability of code and comments, and the important of consistent use of conventions and idioms in achieving all of these. It's hard to argue that these are bad things.

But why worry about style? Who cares what a program looks like if it works? Doesn't it take too much time to make it look pretty? Aren't the rules arbitrary anyway?

The answer is that well-written code is easier to read and to understand, almost surely has fewer errors, and is likely to be smaller than code that has been carelessly tossed (摇摆) and never polished. In the rush to get programs out the door to met some deadline, it's easy to push style aside, to worry about it later. This can be a costly decision. Some of the examples in this chapter show what can go wrong if there isn't enough attention to good style. Sloppy (粗心的) code is bad code -- not just awkward and hard to read, but often broken.

The key observation is that good style should be a matter of habbit. If you think about style as you write the code originally, and if you take the time to revise and improve it, you will develop good habbits. Once they become automatic, your subconscious will take care of many of the details for you, and even the code you produce under pressure will be better.

Supplementary (补充的) Reading

As we said at the beginning of the chapter, writing good code has much in common with writing good English. Strunk and White's *The Elements of Style* (Allyn & Bacon) is still the best short book on how to write English well.

This chapter draw on the approach of *The Elements of Programming Style* by Brian Kernighan and P. J. Plauger (McGraw-Hill, 1978). Steve Maguire's *Writing Solid Code* (Microsoft Press, 1993) is an excellent source of programming advise. There are also helpful descussions of style in Steve McConnell's *Code Complete* (Microsoft Press, 1993) and Peter van der Linder's *Expert C Programming: Deep C Secrets* (Prentice Hall, 1994).

Chapter 2

Algorithms and Data Structures

Chapter 3

Design and Implementation

Chapter 4

Interfaces

Chapter 5

Debugging

Chapter 6

Testing

Chapter 7

Performance

Chapter 8

Portability

Chapter 9

Notation