The Practice of Programming

Brian W.Kernighan Rob Pike https://github.com/wuzhouhui/tpop

July 3, 2015

Contents

Preface

The presentation is organized into nine chapters, each 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 2, 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 4 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 design: 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 5 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 7 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 8 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 necessary to adapt it to a new environment.

Computing is rich 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 9 presents several 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 certain of doing as well, he will probably 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 extremely 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 the 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 crucial 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 simple -- straightforward login, natural expression, conventional language use, meaningful names, neat formatting, helpful comments -- and it should be avoid clever tricks and unusual constructions. Consistency 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 artificial. 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 distinguish 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 comes 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 conventions 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 relationship and highlight their difference.

Besides being 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 opposite 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.

Exercise 1-1 . 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;
```

□

Exercise 1-2 . Improve this function:

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

□

Exercise 1-3 . 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 desk so you can find things. Unlike your desk, your programs are likely to 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))
        ...
```

Now 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 logical operators(&& and ||).

When mixing unrelated operators, 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 certainly 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-precedence 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 operations 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 contains 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 caution in any expression with side effects.

Exercise 1-4 . 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;
```

□

Exercise 1-5 . What is wrong with this excerpt (摘录)?

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

□

Exercise 1-6 . 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)

Consistency leads to better programs. If formatting varies unpredictably, or loop over an array runs until 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 experienced 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 each 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 update 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`, beware.

Java doesn't suffer from this specific problem, since string are not represented as null-terminated arrays. Array subscripts 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 (condition_i)
    statement_i
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 omitted 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`. Vertical 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-through 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.

Exercise 1-7 . 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);
```

□

Exercise 1-8 . 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 outweigh (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 guarantee 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 occurrence 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 address 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 syntactic trouble while offering whatever performance advantage macros might provide. They are appropriate for short functions that set or retrieve a single value.

Exercise 1-9 . 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 cursor-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 others, 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). Making 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. The C preprocessor 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.

Exercise 1-10 . 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 should add something 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 parenthesis */
?       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: return an integer in the range [0, r-1].
int random(int r)
{
    return (int)(Math.floor(Math.random() * r));
}
```

Sometimes code 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 decisions were made. This comment introduces an extremely 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 dimensional 8*8 Discrete Cosine Transform,
 * Chen-Wang algorithm (IEEE ASSP-32, pp 803-816, Aug 1984)
 *
 * 32-bit integer arithmetic (8-bit coefficients)
 * 11 multiplies, 29 adds per DCT
 *
 * coefficients 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 outweigh 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 agree with code when they are written, but as bugs are fixed and the program evolves, the comments are often left in their original form, resulting in disagreement with the code. This is the likely explanation for the inconsistentency in the example that opens this chapter.

Whatever the source of the disagreement, 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 still 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 about 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 being 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 and 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 often 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.

Exercise 1-11 . 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 habit. 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 habits. 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 discussions 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

In the end, only familiarity with the tools and techniques of the field will provide right solution for a particular problem, and only a certain amount of experience will provide consistently professional result.

Raymond Fielding.*The Technique of Special Effects Cinematography*

The study of algorithms and data structures is one of the foundations of computer science, a rich field elegant techniques and sophisticated mathematical analyses. And it's more than just fun and games for the theoretically inclined: a good algorithm or data structure might make it possible to solve a problem in seconds that could otherwise take years.

In specialized areas like graphics, databases, parsing, numerical analysis, and simulation, the ability to solve problems depends critically on state-of-the-artalgorithms and data structures. If you are developing your programs in a field that's new to you, you *must* find out what is already known, lest (免得) you waste your time doing poorly what others have already done well.

Every program depends on algorithms and data structures, but few programs depend on the invention of brand new ones. Even within an intricate (错综复杂的) program like a compiler or a web browser, most of the data structures are arrays, lists, trees and hash tables. When a program needs something more elaborate(详细), it will likely be based on these simpler ones. Accordingly, for most programmers, the task is to know what appropriate algorithms and data structures are available and to understand how to choose among alternatives.

Here is the story in a nutshell. There are only a handful of basic algorithms that show up in almost every program -- primarily searching and sorting -- and even those are often included in libraries. Similarly, almost every data structure is derived from a few fundamental ones. Thus the material covered in this chapter will be familiar to almost all programmers. We have written working versions to make the discussion concrete, and you can lift code verbatim (逐字的) if necessary, but do so only after you have investigated what the programming language and its libraries have to offer.

2.1 Searching

Nothing beats an array for storing static tabular data. Compile-time initialization makes it cheap and easy to construct such arrays. (In Java, the initialization occurs at run-time, but this is an unimportant

implementation detail unless the arrays are large.) In a program to detect words that are used rather too much in bad prose, we can write

```
/*lookup: sequential search for word in array */
int lookup(char *word, char *array[])
{
    int i;
    for (i = 0; array[i] != NULL; i++)
       if (strcmp(word, array[i] == 0))
            return i;
    return -1;
}
```

In C and C++, a parameter that is an array of strings can be declared as `char *array[]` or `char *+array`. Although these forms are equivalent, the first makes it clearer how the parameter will be used.

This search algorithm is called ***sequential search*** because it looks at each element in turn to see if it's the desired one. When the amount of data is small, sequential search is fast enough. There are standard library routines to do sequential search for specific data types; for example, functions like `strchr` and `strstr` search for the first instance of a given character or substring in a C or C++ string. The Java `String` class has an `indexOf` method, and the generic C++ `find` algorithms apply to most data types. If such a function exists for the data type you've got, use it.

Sequential search is easy but the amount of work is directly proportional to the amount of data to be searched; doubling the number of elements will double the time to search if the desired item is not present. This is a linear relationship -- run-time is a linear function of data size -- so this method is also known as ***linear search***.

Here's an excerpt from an array of more realistic size from a program that parses HTML, which defines textual names for well over a hundred individual characters:

```
typedef struct Nameval Nameval;
struct Nameval{
    char *name;
    int value;
};
/* HTML characters, e.g. AElig is ligature of A and E. */
/* Values are Unicode/ISO10646 encoding. */
Nameval htmlchars[] = {
    "AElig",    0x00c6,
    "Aacute",   0x00c1,
    "Acirc",    0x00c2,
    /* ... */
    "zeta",     0x03b6,
};
```

For a larger array like this, it's more efficient to use `binary` search. The binary search algorithm is an orderly version of the way we look up words in a dictionary. Check the middle element. If that value is bigger than what we are looking for, look in the first half; otherwise, look in the second half. Repeat until the desired item is found or determined not to be present.

For binary search, the table must be sorted, as it is here (that's good style anyway; people find things faster in sorted tables too), and we must know how long the table is. The `NELEMS` macro from Chapter 1 can help:

```
printf("The HTML table has %d words\n", NELEMS(htmlchars));
```

A binary search function for this table might look like this:

Chapter 3

Design and Implementation

Chapter 4

Interfaces

Chapter 5

Debugging

Bug, a defect or fault in a machine, plan, or the like. orig.U.S. 1889 Pall Mall Gaz. 11 Mar: 1/1 Mr. Edison, I was informed, has been up the two previous nights discovering 'a bug' in his phonograph -- an expression for solving a difficulty, implying that some imaginary insect has secreted itself inside and is causing all the trouble.

Oxford English Dictionary. 2nd Edition

We have presented a lot of code in the past four chapters, and we've pretended that it all pretty much worked the first time. Naturally this wasn't true; there were plenty of bugs. The word "bug" didn't originate with programmers, but it is certainly one of the most common terms in computing. Why should software be so hard?

One reason is that the complexity of a program is related to the number of ways that its components can interact, and software is full of components and interactions. Many techniques attempt to reduce the connections between components so there are fewer pieces to interact; examples include information hiding, abstraction and interfaces, and the language features that support them. There are also techniques for ensuring the integrity of a software design -- program proofs, modeling, requirements analysis, formal verification -- but none of these has yet changed the way software is built; they have been successful only on small problems. The reality is that there will always be errors that we find by testing and eliminate by debugging.

Good programmers know that they spend as much time debugging as writing so they try to learn from their mistakes. Every bug you find can teach you how to prevent a similar bug from happening again or to recognize it if it does.

Debugging is hard and can take long and unpredictable amount of time, so the goal is to avoid having to do much of it. Techniques that help reduce debugging time include good design, good style, boundary condition tests, assertions (断言) and sanity (完整性) checks in the code, defensive (防御性) programming, well-designed interfaces, limited global data, and checking tools. An ounce (盎司) of prevention really is worth a pound (磅) of cure.

What is the role of language? A major force in the evolution of programming language has been the attempt to prevent bugs through language features. Some features make classes of errors less likely: range checking on subscript, restricted pointers or no pointers at all, garbage collection, string data types, typed UO, and strong type-checking. On the opposite side of the coin, some features are prone (倾向的) error, like

`goto` statements, global variables, unrestricted pointers, and automatic type conversions. Programmers should know the potentially risky bits of their languages and take extra care when using them. They should also enable all compiler checks and heed (注意) warnings.

Each language feature that prevents some problem has a cost of its own. If a higher-level language makes the simple bugs disappear automatically, the price is that it makes it easier to create higher-level bugs. No language prevents you from making mistakes.

Even though we wish it were otherwise, a majority of programming time is spent testing and debugging. In this chapter, we'll discuss how to make your debugging time as short and productive as possible; we'll come back to testing in Chapter 6.

5.1 Debuggers

Compiler for major languages usually come with sophisticated (复杂的) debuggers, often packaged as part of a development environment that integrates creation and edition of source code, compilation, execution, and debugging, all in a single system. Debuggers include graphical interfaces for stepping through a program one statement or function at a time, stopping at particular lines or when a specific condition occurs. They also provide facilities for formatting and displaying the values of variables.

A debugger can be invoked directly when a problem is known to exist. Some debuggers take over automatically when something unexpectedly goes wrong during program execution. It's usually easy to find out where the program was executing when it died, examine the sequence of functions that were active (the stack trace), and display the values of local and global variables. That much information may be sufficient to identify a bug. If not, breakpoints and stepping make it possible to re-run a failing program one step at a time to find the first place where something goes wrong.

In the right environment and in the hands of an experienced user, a good debugger can make debugging effective and efficient, if not exactly painless. With such powerful tools at one's disposal, why would anyone ever debug without them? Why do we need a whole chapter on debugging?

There are several good reasons, some objective and some based on personal experience. Some languages outside the mainstream have no debugger or provide only rudimentary (基本的) debugging facilities. Debuggers are system-dependent, so you may not have access to the familiar debugger from one system when you work on another. Some programs are not handled well by debuggers: multi-process or multi-thread programs, operating systems, and distributed systems must often be debugged by lower-level approaches. In such situations, you're on your own, without much help besides print statements and your own experience and ability to reason about code.

As a personal choice, we tend not to use debuggers beyond getting a stack trace or the value of a variable or two. One reason is that it is easy to et lost in details of complicated data structures and control flow; we find stepping through a program less productive than thinking harder and adding out statements and self-checking code at critical places. Clicking over statements takes longer than scanning the output of judiciously-placed (精心放置的) displays. It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is. More important, debugging statements stay with the program; debugger sessions are transient.

Blind probing with a debugger is not likely to be productive. It is more helpful to use the debugger to discover the state of the program when it fails, then think about how the failure could have happened.

Debuggers can be arcane (晦涩难解的) and difficult programs, and especially for beginners may provide more confusion than help. If you ask the wrong question, they will probably give you an answer, but you may not know it's misleading.

A debugger can be of enormous (庞大的) value, however, and you should certainly include one in your debugging toolkit; it is likely to be the first thing you turn to. But if you don't have a debugger, or if you're stuck on an especially hard problem, the techniques in this chapter will help you to debug effectively and efficiently anyway. They should make your use of your debugger more productive as well, since they are largely concerned with how to reason about errors and probable causes.

5.2 Good Clues, Easy Bugs

Oops! Something is badly wrong. My program crashed, or printed nonsense, or seems to be running forever. Now what?

Beginners have a tendency to blame the compiler, the library, or anything other than their own code. Experienced programmers would love to do the same, but they know that, realistically, most problems are their own fault.

Fortunately, most bugs are simple and can be found with simple techniques. Examine the evidence in the erroneous output and try to infer (推论) how it could have been produced. Look at any debugging output before the crash; if possible get a stack trace from a debugger. Now you know something of what happened, and where. Pause to reflect. How could that happen? Reason back from the state of the crashed program to determine what could have caused this.

Debugging involves backwards reasoning, like solving murder mysteries. Something impossible occurred, and the only solid information is that it really did occur. So we must think backwards from the result to discover the reasons. Once we have a full explanation, we'll know what to fix and, along the way, likely discover a few other things we hadn't expected.

Look for familiar patterns. Ask yourself whether this is familiar pattern. "I've seen that before" is often the beginning of understanding, or even the whole answer. Common bugs have distinctive signatures. For instance, novice (新手) C programmers often write:

```
?   int n;
?   scanf("%d", n);
```

instead of

```
    int n;
    scanf("%d", &n);
```

and this typically causes an attempt to access out-of-bounds memory when a line of input is read. People who teach C recognize the symptom instantly.

Mismatched types and conversion in `printf` and `scanf` are an endless source of easy bugs:

```
?   int n = 1;
?   double d = PI;
?   printf("%d %f\n", d, n);
```

The signature of this error is sometimes the appearance of preposterous (反常的) values: huge integers or improbably large or small floating-point values. On a Sun SPARC, the output from this program is a huge number and an astronomical (天文的) one[1] (folded to fit):

[1] It's hard to type a long number, so I omit most of digits.

```
107434034 26815 ... 30144.000000
```

Another common error is using `%f` instead of `%lf` to read a `double` with `scanf`. Some compilers catch such mistakes by verifying that the type of `scanf` and `printf` arguments match their format string; if all warnings are enabled. for the `printf` above, the GNU compiler gcc reports that

```
x.c:9: warning: int format, double arg (arg 2)
x.c:9: warning: double format, different type arg (arg 3)
```

Failing to initialize a local variable give rise to another distinctive error. The result often an extremely large value, the garbage left over from whatever previous value was stored in the same memory location. Some compilers will warn you, though you may have enable the compile-time check, and they can never catch all cases. Memory returned by allocators like `malloc`, `realloc`, and `new` is likely to be garbage too; be sure to initialize it.

Examine the most recent change. What was the last change? If you're changing only one at a time as a program evolves, the bug most likely is either in the new code or has been exposed by it. Looking carefully at recent changes helps to localize the problem. If the bug appears in the new version and not in the old, the new code is part of the problem. This means that you should preserve at least the previous version of the program, which you believe to be correct, so that you can compare behaviors. It also means that you should keep records of the changes made and bugs fixed, so you don't have to rediscover this vital information while you're trying to fix a bug. Source code control systems and other history mechanisms are helpful here.

Don't make the same mistake twice. After you fix a bug, ask whether you might have made the same mistake somewhere else. This happened to one of us just days before beginning to write this chapter. The program was a quick prototype for a colleague, and included some boilerplate (样板) for optional arguments:

```
?   for (i = 1; i < argc; i++) {
?       if (argv[i][0] != '-')  /* options finished */
?           break;
?       switch (argv[i][1]) {
?       case 'o':           /* output filename */
?           outname = argv[i];
?           break;
?       case 'f':
?           from = atoi(argv[i]);
?           break;
?       case 't':
?           to = atoi(argv[i]);
?           break;
?       ...
```

Shortly after our colleague tried it, he reported that the output file name always had the prefix `-o` attached to it. This was embarrassing but easy to repair; the code should have read

```
outname = &argv[i][2];
```

So that was fixed up and shipped off, and back came another report that the program failed to handle an argument like `-f123` properly: the converted numeric value was always zero. This is the same error; the next case in the `switch` should have read

```
from = atoi(&argv[i][2]);
```

Because the author was still in a hurry, he failed to notice that the same blunder (错误) occurred twice more and it took another round before all of the fundamentally identical errors were fixed.

Easy code can have bugs if its familiarity causes us to let down our guard. Even when code is so simple you could write it in your sleep, don't fall asleep while writing it.

Debug it now, not later. Being in too much of a hurry can hurt in other situations as well. Don't ignore a crash when it happens; track it down right away, since it may not happen again until it's too late. A famous example occurred on the Mars Pathfinder mission. After the flawless (完美的) landing in July 1977 the spacecraft's computers tended to reset once a day or so, and the engineer were baffled (困惑). Once they tracked down the problem, they realized that they had seed that problem before. During pre-launch tests the resets had occurred, but had been ignored because the engineers were working on unrelated problems. So they were forced to deal with the problem later when the machine was tens of millions of miles away and much harder to fix.

Get a stack trace. Although debuggers can probe running programs, one of their most common uses is to examine the state of a program after death. The source line number of the failure, often part of a stack trace, is the most useful single piece of debugging information; improbable (不太可能的) values of arguments are also a big clue (zero pointers, integers that are huge when they should be small, or negative when they should be positive, character strings that aren't alphabetic).

Here is a typical example, based on the discussion of sorting in Chapter 2. To sort an array of integers, we should call `qsort` with the integer comparison function `icmp`:

```
int arr[N];
qsort(arr, N, sizeof(arr[0]), icmp);
```

but suppose it is inadvertently (粗心地) passed the name of the string comparison function `scmp` instead:

```
?   int arr[N];
?   qsort(arr, n, sizeof(arr[0]), scmp);
```

A compiler can't detect the mismatch of types here, so disasters awaits. When we run the program, it crashes by attempting to access an illegal memory location. Running the `dbx` debugger produces a stack trace like this, edited to fit[2]:

```
0 strcmp(0x1a2, 0x1c2) ["strcmp.s":31]
1 scmp(p1 = 0x10001048, p2 = 0x1000105c) ["badqs.c":31]
2 qst(0x10001048, 0x10001074, 0x400b20, 0x4) ["qsort.c":147]
3 qsort(0x10001048, 0x1c2,0x4, 0x400b20) ["qsort.c",63]
4 main() ["badqs.c":45]
5 __istart() ["crtltinit.s":13]
```

This says that the program died in strcmp; by inspection, the two pointers passed to `strcmp` are much too small, a clear sign of trouble. The stack trace gives a trail (痕迹) of line numbers where each function was called. Line 13 in our test file `badqs.c` is the call

```
return strcmp(v1, v2);
```

which identifies the failing call and points towards the error.

A debugger can also be used to display values of local or global variables that will give additional information about what went wrong.

[2]The source pdf is blurry (模糊的), so the stack trace may contain some typing errors

Read before typing. One effective but under-appreciated debugging technique is to read the code very carefully and think about it for a while without making changes. There's a powerful urge to get to the keyboard and start modifying the program to see if the bug goes away. But chances are that you don't know what's really broken and will change the wrong thing, perhaps breaking something something else. A listing of the critical part of program on paper can give a different perspective than what you see on the screen, and encourages you to take more time for reflection. Don't make listings as a matter of routine, though. Printing a complete program wastes trees since it's hard to see the structure when it's spread across many pages and the listing will be obsolete (荒废的) the moment you start editing again.

Take a break for while; sometimes what you see in the source code is what you meant rather than what you wrote, and an interval away from it can soften your misconception (误解) and help the code speak for itself when you return.

Resist the urge to start typing; thinking is a worthwhile alternative.

Explain your code to someone else. Another effective technique is to explain your code to someone else. This will often cause you to explain the bug to yourself. Sometimes it takes no more than a few sentences, followed by an embarrassed "Never mind, I see what's wrong. Sorry to bother you." This works remarkably well; you can even use non-programmer as listeners. One university computer center kept a teddy bear near the help desk. Students with mysterious bugs were required to explain to the bear before they could speak to a human counselor (顾问).

5.3 No Clues, Hard Bugs

"I haven't got a clue. What on earth is going on?" If you really haven't any idea what could be wrong, life get tougher.

Make the bug reproducible. The first step is to make sure you can make the bug appear on demand. It's frustrating to chase down a bug that doesn't happen every time. Spend some time constructing input and parameter settings that reliably cause the problem, then wrap up the recipe (处方) so it can be run with a button push or a few keystrokes. If it's a hard bug, you'll be making it happen over and over as you track down the problem, so you'll save yourself time by making it easy to reproduce.

If the bug can't be made to happen every time, try to understand why not. Does some set of conditions make it happen more often than others? Even if you can't make it happen every time, if you can decrease the time spent waiting for it, you'll find it faster.

If a program provides debugging output, enable it. Simulation program like the Markov chain program in Chapter 3 should include an option that produces debugging information such as the seed of the random number generator so that output can be reproduced; another option should allow for setting the seed. Many programs include such options and it is a good idea to include similar facilities in your own programs.

Divide and conquer (分而治之). Can the input that causes the program to fail be made smaller or more focused? Narrow down the possibilities by creating the smallest input where the bug still shows up. What changes make the error go way? Try to find crucial (决定性的) test cases that focus on the error. Each test case should aim at a definitive outcome that confirms or denies hypotheses (假说) about what is wrong.

Proceed by binary search. Throw away half the input and see if the output is still wrong; if not, go back to the previous state and discard the other half of the input. The same binary search process can be used

on the program text itself: eliminate some part of the program that should have no relationship to the bug and see if the bug is still there. An editor with undo is helpful in reducing big test cases and big programs without losing the bug.

Study the numerology (命理) of failures. Sometimes a pattern in the numerology of failing examples give a clue that focus the search. We found some spelling mistakes in a newly written section of this book, where occasional letters had simply disappeared. this was mystifying (迷惑的). The text had been created by cutting and pasting from another file, so it seemed possible that something was wrong with the cut or paste commands in the text editor. But where to start looking for the problem? For clues we looked at the data, and noticed that the missing characters seemed uniformly (均匀地) distributed through the text. We measured the intervals and found that the distance between dropped characters was always 1023 bytes, a suspiciously non-random value. A search through the editor source code for numbers near 1024 found a couple of candidates. One of those was in new code, so we examined that first, and the bug was easy to spot, a classic off-by-one error where a null byte overwrote the last character in a 1024-byte buffer.

Studying the patterns of numbers related to the failure pointed us right at the bug. Elapsed time? A couple of minutes of mystification, five minutes of looking at the data to discover the pattern of missing characters, a minute to search for likely places to fix, and another minute to identify and eliminate the bug. This one would have been hopeless to find with a debugger, since it involved two multiprocess programs, driven by mouse clicks, communicating through a file system.

Display output to localize your search. If you don't understand what the program is doing, adding statements to display more information can be the easiest, most cost-effective way to find out. Put them in to verify your understanding or refine your ideas of what's wrong. For example, display "can't get here" if you think it's not possible to reach a certain point in the code; then if you see that message, move the output statements back towards the start to figure out where things first begin to go wrong. Or show "got here" messages going forward, to find the last place where things seem to be working. Each message should be distinct so you can tell which one you're looking at.

Display messages in a compact fixed format so they are easy to scan by eye or with programs like the pattern-matching tool `grep`. (A grep-like program is invaluable (无价的) for searching text. Chapter 9 includes a simple implementation.) If you're displaying the value of a variable, format it the same way each time. In C and C++, show pointers as hexadecimal numbers with `%x` or `%p`; this will help you to see whether two pointers have the same value or are related. Learn to read pointer values and recognize likely and unlikely ones, like zero, negative numbers, odd numbers, and small numbers. Familiarity with the form of address will pay off (获益) when you're using a debugger, too.

If output is potentially voluminous (长篇的), it might be sufficient to print single-letter outputs like `A`, `B`, `...`, as a compact display of where the program went.

Write self-checking code. If more information is needed, you can write your own check function to test a condition, dump relevant variables, and abort the programs:

```
/* check: test condition, print and dir */
void check(char *s)
{
    if (var1 > var2) {
        printf("%s, var1 %d var2 %d\n", s, var1, var2);
        fflush(stdout); /* make sure all output is out */
        abort();        /* signal abnormal termination */
    }
}
```

We wrote `check` to call `abort`, a standard C library function that causes program execution to be terminated abnormally for analysis with a debugger. In a different application, you might want `check` to carry on after printing.

Next, add calls to `check` wherever they might be useful in your code:

```
check("before suspect");
/* ... suspect code ... */
check("after suspect");
```

After a bug is fixed, don't throw `check` away. Leave it in the source, commented out or controlled by a debugging option, so that it can be turned on again when the next difficult problem appears.

For harder problems, `check` might evolve to do verification and display of data structures. This approach can be generalized to routine that perform ongoing (进行中) consistency checks of data structures and other information. In a program with intricate (复杂的) data structures, it's good idea to write these checks ***before*** problems happen, as components of the program proper (适当的), so they can be turned on when trouble starts. Don't use then only when debugging; leave them installed during all stages of program development. If they're not expensive, it might be wise to leave them always enables. Large programs like telephone switching systems often devote a significant amount of code to "audit" (审计) subsystems that monitor information and equipment, and report or even fix problems if they occur.

Write a log file. Another tactic (策略) is to write a ***log file*** containing a fixed-format stream of debugging output. When a crash occurs, the log records what happened just before the crash. Web servers and other network programs maintain extensive logs of traffic so they can monitor themselves and their clients; this fragment (edited to fit) comes from a local system:

```
[Sun Dec 27 16:19:24 1998]
HTTPd: access to /usr/local/httpd/cgi-bin/test.html
    failed for m1.cs.bell-labs.com,
    reason: client denied by server (CGI non-executable)
    from http://m2.cs.bell-labs.com/cgi-bin/test.pl
```

Be sure to flush I/O buffers so the final log records appear in the log file. Output functions like `printf` normally buffer their output to print it efficiently; abnormal termination may discard this buffered output. In C, a call to `fflush` guarantees that all output is written before the program dies; there are analogous `flush` function for output streams in C++ and Java. Or, if you can afford the overhead, you can avoid the flushing problem altogether by using unbuffered I/O for log files. The standard functions `setbuf` and `setvbuf` control buffering; `setbuf(fp, NULL)` turns off buffering on the stream `fp`. The standard error streams (`stderr`, `cerr`, `System.err`) are normally unbuffered by default.

Draw a picture. Sometimes pictures are more effective than text for testing and debugging. Pictures are especially helpful for understanding data structures, as we say in Chapter 2, and of course when writing graphics software, but they can be used for all kinds of programs. Scatter (散点) plots display misplaced values more effectively than columns of numbers. A histogram of data reveals anomalies in exam grades, random numbers, bucket sizes in allocators and hash tables, and the like.

If you don't understand what's happening inside your program, try annotating the data structures with statistics and plotting the result. The following graphs plot[3], for the C `markov` program in Chapter 3, hash chain lengths on the x axis and the number of elements in chains of that length on the y axis. The input data is our standard test, *The Book of Psalms* (42685 words, 22482 prefixes). The first two graphs

[3]Lack of graphs, because the graph is too blurry and ugly to represent.

are for the good hash multipliers of 31 and 37 and the third is for the awful multiplier of 128. In the first two cases, no chain is longer than 15 or 16 elements and most elements are in chains of length 5 or 6. In the third, the distribution is broader, the longest chain has 187 elements, and there are thousands of elements in chains longer than 20.

Use tools. Make good use of the facilities of the environment where you are debugging. For example, a file comparison program like `diff` compares the outputs from successful and failed debugging runs so you can focus on what has changed. If your debugging output is long, use `grep` to search it or an editor to examine it. Resist the temptation to send the debugging output to a printer: computers scan voluminous output better than people do. Use shell scripts and other tools to automate the processing of the output from debugging runs.

Write trivial programs to test hypotheses or confirm your understanding of how something works. For instance, is it valid to free a `NULL` pointer?

```
int main(void)
{
    free(NULL);
    return 0;
}
```

Source code control programs like RCS keep track of versions of code so you can see what has changed and revert to previous to restore a known state. Besides indicating what has changed recently, they can identify sections of code that have a long history of frequent modification; these are often a good place for bugs to lurk (潜伏).

Keep records. If the search for a bug goes on for any length of time, you will begin to lose track of what you tried and what you learned. If you record your tests and results, you are less likely to overlook something or to think that you have checked some possibility when you haven't. The act of writing will help you remember the problem the next time something similar comes up, and will also serve when you're explaining it to someone else.

5.4 Last Resorts (手段)

What do you do if none of this advice help? This may be the time to use a good debugger to step through the program. If your mental (精神上的) model of how something works is just pain wrong, so you're looking in the wrong place entirely, or looking in the right place but not seeing the problem, a debugger forces you to think differently. These "mental model" bugs are among the hardest to find; the mechanical aid is invaluable.

Sometimes the misconception (误解) is simple: incorrect operator precedence, or the wrong operator, or indentation that doesn't match the actual structure, or a scope error where a local name hides a global name or a global name introduces into a local scope. For example, programmers often forget that `&` and `|` have lower precedence than == and `!=`. They write

```
?   if (x & 1 == 0)
?       ...
```

and can't figure out why this is always false. Occasionally a slip of the finger converts a single = into two or vice versa:

```
?   while ((c == getchar()) != EOF)
?       if (c = '\n')
?           break;
```

Or extra code is left behind during editing:

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

Or hasty typing creates a problem:

```
?   switch (c) {
?       case '<':
?           mode = LESS;
?           break;
?       case '>':
?           mode = GREATER;
?           break;
?       default:
?           mode = EQUAL;
?           break;
?   }
```

Sometimes the error involves arguments in the wrong order in a situation where type-checking can't help, like writing

```
?   memset(p, n, 0);    /* store n 0's in p */
```

instead of

```
    memset(p, 0, n);    /* store n 0's in p */
```

Sometimes something changes behind your back -- global or shared variables are modified and you don't realize that some other routine can touch them.

Sometimes your algorithms or data structure has a fatal flaw (缺陷) and you just can't see it. While preparing material on linked lists, we wrote a package of list functions to create new elements, link them to the front or back of lists, and so on; these functions appear in Chapter 2. Of course we wrote a test program to make sure everything was correct. The first few tests worked but then one failed spectacularly (惊人地). In essence (基本), this was the testing program:

```
?   while (scanf(%s %d", name, &value) != EOF) {
?       p = newitem(name, value);
?       list1 = addfront(list1, p);
?       list2 - addend(list2, p);
?   }
?   for (p = list1; p != NULL; p = p->next)
?       printf("%s %d\n", p->name, p->value);
```

It was surprisingly difficult to see that the first loop was putting the same node p on both lists so the pointers were hopelessly scrambled (扰乱) by the time we got to printing.

It's tough to find this kind of bug, because your brain takes you right around the mistake. Thus a debugger is a help, since it forces you to go in a different direction, to follow what the program is doing, not what you think it is doing. Often the underlying problem is something wrong with the structure of the whole program, and to see the error you need to return to your starting assumptions.

Notice, by the way, that in the list example the error was in the test code, which made the bug that much harder to find. It is frustratingly easy to waste time chasing bugs that aren't there, because the test program is wrong, or by testing the wrong version of the program, or by failing go update or recompile before testing.

If you can't find a bug after considerable work, take a break. Clear your mind, do something else. Talk to a friend and ask for help. The answer might appear out of the blue, but if not, you won't be stuck in the same rut (凹槽) in the next debugging session.

Once in a long while, the problem really is the compiler or a library or the operating system or even the hardware, especially if something changed in the environment just before a bug appeared. You should never start by blaming one of these, but when everything else has been eliminated, that might be all that's left. We once had to move a large text-formatting program from its original Unix home to a PC. The program compiled without incident, but behaved in an extremely odd way: it dropped roughly every second (次要的) character of its input. Our first thought was that this must be some property of using 16-bit integers instead of 32-bit, or perhaps some strange byte-order problem. But by printing out the characters seen by the main loop, we finally tracked it down to an error in the standard header file `ctype.h` provided by the compiler vendor. It implemented `isprint` as a function macro:

```
?   #define isprint(c) ((c) >= 040 && (c) < 0177)
```

and the main input loop was basically

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

Each time an input character was blank (octal 40, a poor way to write ' ') or greater, which was most of the time, `getchar` was called a second time because the macro evaluated its argument twice, and the first input character disappeared forever. The original code was not as clean as it should have been -- there's too much in the loop condition -- but the vendor's header file was inexcusably (无法容许地) wrong.

One can still find instances of this problem today; this macro comes from a different vendor's current header files:

```
?   #define __iscsym(c) (isalnum(c) || ((c) == '_'))
```

Memory "leaks" -- the failure to reclaim memory that is no longer in use -- are a significant source of erratic behavior. Another problem is forgetting to close files, until the table of open files is full and the program cannot open any more. Programs with leaks tend to fail mysteriously because they run out of some resource but the specific failure can't be reproduced.

Occasionally hardware itself goes bad. The floating-point flaw (缺陷) in the 1994 Pentium processor that caused certain computations to produce wrong answers was a highly publicized (广为宣传的) and costly bug in the design of the hardware, but once it had been identified, it was of course reproducible. One of the strangest bugs we ever saw involved a calculator program, long ago on a two-processor system. Sometimes the expression $1/2$ would print 0.5 and sometimes it would print some consistent but utterly (完全) wrong value like 0.7432; there was no pattern as to whether one got the right answer or the wrong one. The problem was eventually traced to a failure of the floating-point unit in one of the processors. As the calculator program was randomly executed on one processor or the other, answers were either correct or nonsense.

Many years ago we used a machine whose internal temperature could be estimated from the number of low-order bits it got wrong in floating-point calculations. One of the circuit cards was loose; as the machine

got warmer, the card tilted (倾斜的) further out of its socket, and more data bits were disconnected from the backplane (基架).

5.5 Non-reproducible Bugs

Bugs that won't stand still are the most difficult to deal with, and usually the problem isn't as obvious as failing hardware. The very fact that the behavior is non- deterministic is itself information, however; it means that the error is not likely to be a flaw (缺陷) in your algorithm but that in some way your code is using information that changes each time the program runs.

Check whether all variables have been initialized; you may be picking up a random value from whatever was previously stored in the same memory location. Local variables of functions and memory obtained from allocators are the most likely culprits (罪魁祸首) in C and C++. Set all variables to known values; if there's a random number seed that is normally set from the time of day, force it to a constant, like zero.

If the bug changes behavior or even disappears when debugging code is added, it may be a memory allocation error -- somewhere you have written outside of allocated memory, and the addition of debugging code changes the layout of storage enough to change the effect of the bug. Most output functions, from `printf` to dialog windows, allocate memory themselves, further muddying the waters (进一步把水搅混).

If the crash site seems far away from anything that could be wrong, the most likely problem is overwriting memory by storing into a memory location that isn't used until much later. Sometimes this is a dangling (悬挂) pointer problem, where a pointer to a local variable is inadvertently (不慎地) returned from a function, then used. Returning the address of a local variable is a recipe (秘诀) for delayed disaster:

```
?   char *msg(int n, char *s)
?   {
?       char buf[100];
?
?       sprintf(buf, "error %d: %s\n", n, s);
?       return buf;
?   }
```

By the time the pointer returned by `msg` is used, it no longer points to meaningful storage. You must allocate storage with `malloc`, use a static array, or require the caller to provide the space.

Using a dynamically allocated value after it has been freed has similar symptoms. We mentioned this in Chapter 2 when we wrote `freeall`. This code is wrong:

```
?   for (p = listp; p != NULL; p = p->next)
?       free(p);
```

Once memory has been freed, it must not be used since its contents may have changed and there is no guarantee that `p->next` still points to the right place.

In some implementations of `malloc` and `free`, freeing an item twice corrupts the internal data structures but doesn't cause trouble until much later, when a subsequent call slips (滑过) on the mess made earlier. Some allocators come with debugging options that can be set to check the consistency of the arena (竞技场) at each call; turn them on if you have a non-deterministic bug. Failing that, you can write your own allocator that does some of its own consistency checking or logs all calls for separate analysis. An allocator that doesn't have to run fast is easy to write, so this strategy is feasible when the situation is dire (可怕的). There are also excellent commercial products that check memory management and catch

errors and leaks: writing your own `malloc` and `free` can give you some of their benefits if you don't have access to them.

When a program works for one person but fails for another, something must depend on the external environment of the program. This might include files read by the program, file permissions, environment variables, search path for commands, defaults, or startup files. It's hard to be a consultant for these situations, since you have to become the other person to duplicate the environment of the broken program.

Exercise 5-1 . Write a version of `malloc` and `free` that can be used for debugging storage-management problems. One approach is to check the entire workspace on each call of `malloc` and `free`; another is to write logging information that can be processed by another program. Either way, add markers to the beginning and end of each allocated block to detect overruns (超限) at either end. □

5.6 Debugging Tools

Debuggers aren't the only tools that help find bugs. A variety of programs can help us wade through voluminous output to select important bits, find anomalies, or rearrange data to make it easier to see what's going on. Many of these programs are part of the standard toolkit; some are written to help find a particular bug or to analyze a specific program.

In this section we will describe a simple program called `strings` that is especially useful for looking at files that are mostly non-printing characters, such as executables or the mysterious binary formats favored by some word processors. There is often valuable information hidden within, like the text of a document, or error messages and undocumented options, or the names of files and directories, or the names of functions a program might call.

We also find `strings` helpful for locating text in other binary files. Image files often contain ASCII strings that identify the program that created them, and compressed files and archives (such as zip files) may contain file names; `strings` will find these too.

Unix systems provide an implementation of `strings` already. although it's a little different from this one. It recognizes when its input is a program and examines only the text and data segments, ignoring the symbol table. Its `-a` option forces it to read the whole file.

In effect, `strings` extracts the ASCII text from a binary file so the text can be read or processed by other programs. If an error message carries no identification, it may not be evident what program produced it, let alone why. In that case, searching through likely directories with a command like

```
% strings *.exe *.dll | grep 'mystery message'
```

might locate the producer.

The `strings` function reads a file and prints all run of at least `MINLEN = 6` printable characters:

```
/* strings: extract printable strings from stream */
void strings(char *name, FILE *fin)
{
    int c, i;
    char buf[BUFSIZ];

    do {    /* once for each string */
        for (i = 0; (c = getc(fin)) != EOF;  ) {
            if (!isprint(c))
```

```
                break;
            buf[i++] = c;
            if (i >= BUFSIZ)
                break;
        }
        if (i >= MINLEN) /* print if long enough */
            printf("%s: %.*s\n", name, i, buf);
    } while (c != EOF);
}
```

The `printf` format string `%.*s` takes the string length from the next argument (`i`), since the string (`buf`) is not null-terminated.

The `do-while` loop finds and then prints each string, terminating at `EOF`. Checking for end of file at the bottom allows the `getc` and `string` loops to share a termination condition and lets a single `printf` handle end of string, end of file, and string too long.

A standard-issue outer loop with a test at the top, or a single `getc` loop with a more complex body, would require duplicating the `printf`. This function started life that way, but it had a bug in the `printf` statement. We fixed that in one place but forgot to fix two others. ("Did I make the same mistake somewhere else?") At that point, it became clear that the program needed to be rewritten so there was less duplicated code; that led to the `do-while`.

The main routine of `strings` calls the `strings` function for each of its argument files:

```
/* strings main: find printable strings in files */
int main(int argc, char *argv[])
{
    int     i;
    FILE    *fin;

    setprogname("strings");
    if (argc == 1)
        printf("usage: strings filenames");
    else {
        for (i = 1; i < argc; i++) {
            if ((fin = fopen(argv[i], "rb")) == NULL)
                printf("can't open %s: ", argv[i]);
            else {
                strings(argv[i], fin);
                fclose(fin);
            }
        }
    }
    return 0;
}
```

You might be surprised that `strings` doesn't read its standard input if no files are named. Originally it did. To explain why it doesn't now, we need to tell a debugging story.

The obvious test case for `strings` is to run the program on itself. This worked fine on Unix, but under Windows 95 the command

```
C:\> strings < strings.exe
```

produced exactly five lines of output[4]:

[4]The message maybe contains errors, because the source is blurry.

```
!This program cannot be run in DOS mode
.rdata
@.data
.idata
.reloc
```

The first line looks like an error message and we wasted some time before realizing it's actually a string in the program, and the output is correct, at least as far as it goes. It's not unknown to have a debugging session derailed (转移) by misunderstanding the source of a message.

But there should be more output. Where is it? Late one night, the light finally dawned. ("I've seen that before!") This is a portability problem that is described in more detail in Chapter 8. We had originally written the program to read only from its standard input using `getchar`. On Windows. however, `getchar` returns `EOF` when it encounters a particular byte (`Ox1A` or `Ctrl-Z`) in text mode input and this was causing the early termination.

This is absolutely legal behavior, but not what we were expecting given our Unix background. The solution is to open the file in binary mode using the mode `"rb"`. But `stdin` is already open and there is no standard way to change its mode. (Functions like `fdopen` or `setmode` could be used but they are not part of the C standard.) Ultimately we face a set of unpalatable (不可口的) alternatives: force the user to provide a file name so it works properly on Windows but is unconventional on Unix; silently produce wrong answers if a Windows user attempts to read from standard input; or use conditional compilation to make the behavior adapt to different systems, at the price of reduced portability. We chose the first option so the same program works the same way everywhere.

Exercise 5-2 . The `strings` program prints strings with `MINLEN` or more characters, which sometimes produces more output than is useful. Provide `strings` with an optional argument to define the minimum string length. □

Exercise 5-3 . Write `vis`, which copies input to output, except that it displays non-printable bytes like backspace, control characters, and non-ASCII characters as `\Xhh` where `hh` is the hexadecimal representation of the non-printable byte. By contrast with `strings`, `vis` is most useful for examing input that contain only a few non-printable characters. □

Exercise 5-4 . What does `vis` produce if the input is `\X0A`? How could you make the output `vis` unambiguous? □

Exercise 5-5 . Extend `vis` to process a sequence of files, fold long lines at any desired column, and remove non-printable characters entirely. What other features might be consistent with the role of the program? □

5.7 Other People's Bugs

Realistically, most programmers do not have the fun of developing a brand new system from the ground up. Instead, they spend much of their time using, maintaining. modifying and thus, inevitably, debugging code written by other people.

When debugging others' code, everything that we have said about how to debug your own code applies. Before starting, though, you must first acquire some understanding of how the program is organized and

how the original programmers thought and wrote. The term used in one very large software project is "discovery," which is not a bad metaphor (隐喻). The task is discovering what on earth is going on in something that you didn't write.

This is a place where tools can help significantly. Text-search programs like grep can find all the occurrences of names. Cross-reference give some idea of the program's structure. A display of the graph of function calls is valuable if it isn't too big. Stepping through a program a function call at a time with a debugger can reveal the sequence of events. A revision history of the program may give some clues by showing what has been done to the program over time. Frequent changes are often a sign of code that is poorly understood or subject (屈服于) to changing requirements. and thus potentially buggy.

Sometimes you need to track down errors in software you are not responsible for and d o not have the source code for. In that case, the task is to identify and characterize the bug sufficiently well that you can report it accurately. and at the same time perhaps find a "work-around" (迂回路线) that avoids the problem.

If you think that you have found a bug in someone else's program, the first step is to make absolutely sure it is a genuine (眞正的) bug, so you don't waste the author's time and lose your own credibility.

When you find a compiler bug, make sure that the error is really in the compiler and not in your own code. For example, whether a right shift operation fills with zero bits (logical shift) or propagates (蔓延) the sign bit (arithmetic shift) is unspecified in C and C++, so novices (新手) sometimes think it's an error if a construct like

```
?   i = -1;
?   printf("%d\n", i >> 1);
```

yields an unexpected answer. But this is a portability issue, because this statement can legitimately (合法地) behave differently on different systems. Try your test on multiple systems and be sure you understand what happens; check the language definition to be sure.

Make sure the bug is new. Do you have the latest version of the program? Is there a list of bug fixes? Most software goes through multiple releases; if you find a bug in version 4.0b1, it might well be fixed or replaced by a new one in version 4.04b2. In any case, few programmers have much enthusiasm for fixing bugs in any- thing but the current version of a program.

Finally, put yourself in the shoes of the person who receives your report. You want to provide the owner with as good a test case as you can manage. It's not very helpful if the bug can be demonstrated (示范) only with large inputs, or an elaborate (复杂的) environment, or multiple supporting files. Strip the test down to a minimal and self-contained case. Include other information that could possibly be relevant, like the version of the program itself, and of the compiler, operating system, and hardware. For the buggy version of `isprint` mentioned in Section 5.4, we could provide this as a test program:

```
/* test program for isprint bug */
int main(void)
{
    int c;

    while (isprint(c = getchar()) || c != EOF)
        print("%c", c);
    return 0;
}
```

Any line of printable text will serve as a test case, since the output will contain only half the input:

```
% echo 1234567890 | isprint_test
24680
%
```

The best bug reports are the ones that need only a line or two of input on a plain vanilla system to demonstrate the fault, and that include a fix. Send the kind of bug report you'd like to receive yourself.

5.8 Summary

With the right attitude debugging can be fun, like solving a puzzle, but whether we enjoy it or not, debugging is an art that we will practice regularly. Still, it would be nice if bugs didn't happen, so we try to avoid them by writing code well in the first place. Well-written code has fewer bugs to begin with and those that remain are easier to find. Once a bug has been seen, the first thing to do is to think hard about the clues it presents. How could it have come about? Is it something familiar? Was something just changed in the program? Is there something special about the input data that provoked (触发) it? A few well-chosen test cases and a few print statements in the code may be enough.

If there aren't good clues, hard thinking is still the best first step, to be followed by systematic attempts to narrow down the location of the problem. One step is cutting down the input data to make a small input that fails; another is cutting out code to eliminate regions that can't be related. It's possible to insert checking code that gets turned on only after the program has executed some number of steps, again to try to localize the problem. All of these are instances of a general strategy, divide and conquer, which is as effective in debugging as it is in politics and war.

Use other aids as well. Explaining your code to someone else (even a teddy bear) is wonderfully effective. Use a debugger to get a stack trace. Use some of the commercial tools that check for memory leaks, array bounds violations, suspect (可疑的) code, and the like. Step through your program when it has become clear that you have the wrong mental picture of how the code works.

Know yourself, and the kinds of errors you make. Once you have found and fixed a bug, make sure that you eliminate other bugs that might be similar. Think about what happened so you can avoid making that kind of mistake again.

Supplementary Reading

Steve Maguire's *Writing Solid Code* (Microsoft Press, 1993) and Steve McConnell's *Code Complete* (Microsoft Press, 1993) both have much good advice on debugging.

Chapter 6

Testing

Chapter 7

Performance

His promises were, as he then was, mighty; But his performance, as he is now, nothing.

Shakespeare, *King Henry VIII*

Long ago, programmers went to great effort to make their programs efficient because computers were slow and expensive. Today, machines are much cheaper and faster, so the need for absolute efficiency is greatly reduced. Is it still worth worrying about performance?

Yes, but only if the problem is important, the program is genuinely too slow, and there is some expectation that it can be made faster while maintaining correctness, robustness, and clarity. A fast program that gets the wrong answer doesn't save any time.

Thus the first principle of optimization is don't. Is the program good enough already? Knowing how a program will be used and the environment it runs in, is there any benefit to making it faster? Programs written for assignments in a college class are never used again; speed rarely matters. Nor will speed matter for most personal programs, occasional tools, test frameworks, experiments, and prototypes. The run-time of a commercial product or a central component such as a graphics library can be critically important, however, so we need to understand how to think about performance issues.

When should we try to speed up a program? How can we do so? What can we expect to gain? This chapter discusses how to make programs run faster or use less memory. Speed is usually the most important concern, so that is mostly what we'll talk about. Space (main memory, disk) is less frequently an issue but can be crucial, so we will spend some time and space on that too.

As we observed in Chapter 2, the best strategy is to use the simplest, cleanest algorithms and data structures appropriate for the task. Then measure performance to see if changes are needed; enable compiler options to generate the fastest possible code; assess what changes to the program itself will have the most effect; make changes one at a time and re-assess; and keep the simple versions for testing revisions against.

Measurement is a crucial component of performance improvement since reasoning and intuition are fallible (易错的) guides and must be supplemented with tools like timing commands and profilers (探查). Performance improvement has much in common with testing, including such techniques as automation, keeping careful records, and using regression (回归) tests to make sure that changes preserve correctness and do not undo previous improvements.

If you choose your algorithms wisely and write well originally you may find no need for further speedups. Often minor changes will fix any performance problems in well-designed code, while badly-designed code will require major rewriting.

7.1 A Bottleneck

Let us begin by describing how a bottleneck was removed from a critical program in our local environment.

Our incoming mail funnels (漏斗) through a machine, called a gateway, that connects our internal network with the external Internet. Electronic mail messages from outside -- tens of thousands a day for a community of a few thousand people -- arrive at the gateway and are transferred to the internal network; this separation isolates our private network from the public Internet and allows us to publish a single machine name (that of the gateway) for everyone in the community.

One of the services of the gateway is to filter out "spam (罐头猪肉)." unsolicited (未经同意的) mail that advertises services of dubious (可疑的) merit (值得). After successful early trials of the spam filter, the service was installed as a permanent feature for all users of the mail gateway, and a problem immediately became apparent. The gateway machine, antiquated (老旧的) and already very busy, was overwhelmed (倾覆) because the filtering program was taking so much time -- much more time than was required for all the other processing of each message -- that the mail queues filled and message delivery was delayed by hours while the system struggled to catch up.

This is an example of a true performance problem: the program was not fast enough to do its job, and people were inconvenienced by the delay. The program simply had to run much faster.

Simplifying quite a bit, the spam filter runs like this. Each incoming message is treated as a single string, and a textual pattern matcher examines that string to see if it contains any phrases from known spam, such as "Make millions in your spare time" or "XXX-rated." Messages tend to recur (重发), so this technique is remarkably effective, and if a spam message is not caught, a phrase is added to the list to catch it next time.

None of the existing string-matching tools, such as `grep`, had the right combination of performance and packaging (封装), so a special-purpose spam filter was written. The original code was very simple; it looked to see if each message contained any of the phrases (patterns):

```
/* isspam: test mesg for occurrence of any pat */
int isspam(char *mesg)
{
    int i;

    for (i = 0; i < npat; i++)
        if (strstr(mesg, pat[i]) != NULL) {
            printf("spam: match for '%s'\n", pat[i]);
            return 1;
        }

    return 0;
}
```

How could this be made faster? The string must be searched, and the `strstr` function from the C library is the best way to search: it's standard and efficient.

Using profiling (靠模切削***),*** a technique we'll talk about in the next section, it became clear that the implementation of `strstr` had unfortunate properties when used in a spam filter. By changing the way `strstr` worked, it could be made more efficient ***for this particular problem***.

The existing implementation of `strstr` looked something like this:

```
/* simple strstr: use strchr to look for first character */
char *strstr(const char *s1, const char *s2)
{
    int n;

    n = strlen(s2);
    for (;;) {
        s1 = strchr(s1, s2[0]);
        if (s1 == NULL)
            return NULL;
        if (strncmp(s1, s2, n) == 0)
            return s1;
        s1++;
    }
}
```

It had been written with efficiency in mind, and in fact for typical use it was fast because it used highly-optimized library routines to do the work. It called `strchr` to find the next occurrence of the first character of the pattern, and then called `strncmp` to see if the rest of the string matched the rest of the pattern. Thus it skipped quickly over most of the message looking for the first character of the pattern. and then did a fast scan to check the rest. Why would this perform badly?

There are several reasons. First, strncmp takes as an argument the length of the pattern. which must be computed with `strlen`. But the patterns are fixed, so it shouldn't be necessary to recompute their lengths for each message.

Second, `strncmp` has a complex inner loop. It must not only compare the bytes of the two strings, it must look for the terminating `\0` byte on both strings while also counting down the length parameter. Since the lengths of all the strings are known in advance (though not to strncmp), this complexity is unnecessary; we know the counts are right so checking for the `\n` wastes time.

Third, `strchr` is also complex, since it must look for the character and also watch for the `\0` byte that terminates the message. For a given call to `isspam`, the message is fixed, so time spent looking for the `\0` is wasted since we know where the message ends.

Finally, although `strncmp`, `strchr` , and `strlen` are all efficient in isolation, the overhead of calling these functions is comparable to the cost of the calculation they will perform. It's more efficient to do all the work in a special, carefully written version of `strstr` and avoid calling other functions altogether.

These sorts of problems are a common source of performance trouble -- a routine or interface works well for the typical case, but performs poorly in an unusual case that happens to be central to the program at issue. The existing `strstr` was fine when both the pattern and the string were short and changed each call, but when the string is long and fixed, the overhead is prohibitive (可抑制的).

With this in mind, `strstr` was rewritten to walk the pattern and message strings together looking for matches, without calling subroutines. The resulting implementation has predictable behavior: it is slightly slower in some cases, but much faster in the spam filter and, most important, is never terrible. To verify the new implementation's correctness and performance, a performance test suite was built. This suite included not only simple examples like searching for a word in a sentence, but also pathological (病态

的) cases such as looking for a pattern of a single x in a string of a thousand e's and a pattern of a thousand x's in a string of a single e, both of which can be handled badly by naive implementations. Such extreme cases are a key part of performance evaluation.

The library was updated with the new `strstr` and the spam filter ran about 30% faster, a good payoff for rewriting a single routine.

Unfortunately, it was still too slow.

When solving problems, it's important to ask right question. Up to now, we've been asking for the fastest way to search for a textual pattern in a string. But the real problem is to search for a large, fixed set of textual patterns in a long, variable string. Put that way, `strstr` is not so obviously the right solution.

The most effective way to make a program faster is to use a better algorithm. With a clearer idea of the problem, it's time to think about what algorithm would work best.

The basic loop,

```
for (i = 0; i < npat; i++)
    if (strstr(mesg, pat[i]) != NULL)
        return 1;
```

scans down the message `npat` independent times; assuming it doesn't find any matches, it examines each byte of the message `npat` times, for a total of `strlen(mesg)*npat` comparisons.

A better approach is to invert the loops, scanning the message once in the outer loop while searching for all the patterns in parallel in the inner loop:

```
for (j = 0; mesg[j] != '\0'; j++)
    if (some pattern matches starting at mesg[j])
        return 1;
```

The performance improvement stems from (主要来源于) a simple observation. To see if any pattern matches the message at position `j`, we don't need to look at all patterns, only those that begin with the same character as `mesg[j]`. Roughly, with 52 upper and lower-case letters we might expect to do only `strlen(mesg)*npat/52` comparisons. Since the letters are not evenly distributed -- words with `s` more often than `x` -- we won't see a factor of 52 improvement, but we should see some. In effect, we construct a hash table using the first character of the pattern as the key.

Given some precomputation to construct a table of which patterns begin with each character, `isspam` is still short:

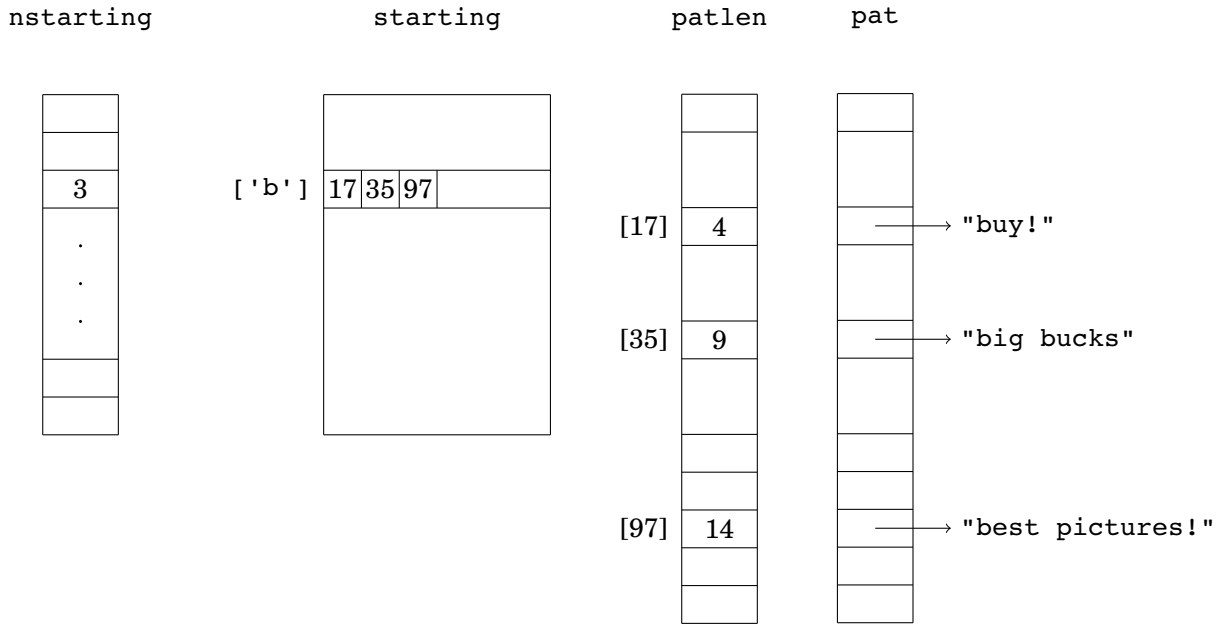
```
int patlen[NPAT];                   /* length of pattern */
int starting[UCHAR_MAX+1][NSTART];  /* pats starting with char */
int nstarting[UCHAR_MAX+1];         /* number of such patterns */
...
/* isspam: test mesg for occurrence of any pat */
int isspam(char *mesg)
{
    int i, j, k;
    unsigned char c;

    for (j = 0; (c = mesg[j]) != '\0'; j++) {
        for (i = 0; i < nstarting[c]; i++) {
            k = starting[c][i];
            if (memcmp(mesg+j, pat[k], patlen[k]) == 0) {
                printf("spam: match for '%s'\n", pat[k]);
```

```
                return 1;
            }
        }
    }
    return 0;
}
```

The two-dimensional array `starting[c][]` stores, for each character `c`, the indices of those patterns that begin with that character. Its companion `nstarting[c]` records how many patterns begin with `c`. Without these tables, the inner loop would run from 0 to `npat`, about a thousand; instead it runs from 0 something like 20. Finally, the array element `patlen[k]` stores the precomputed result of `strlen(pat[k])`.

The following figure sketches (概括) these data structures for a set of three pattern that begin with the letter `b`:



The code to build these tables is easy:

```
int i;
unsigned char c;

for (i = 0; i < npat; i++) {
    c = pat[i][0];
    if (nstarting[c] >= NSTART)
        printf("too many patterns (>=%d) begin '%c'",
                NSTART, c);
    starting[c][nstarting[c]++] = i;
    patlen[i] = strlen(pat[i]);
}
```

Depending on the input, the spam filter is now five to ten times faster than it was using the improved `strstr`, and seven to fifteen times faster than the original implementation. We didn't get a factor of 52, partly because of the non-uniform distribution of letters, partly because the loop is more complicated in the new program, and partly because there are still many failing string comparisons to execute, but the spam filter is no longer the bottleneck for mail delivery. Performance problem solved.

The rest of this chapter will explore the techniques used to discover performance problems, isolate the slow code. and speed it up. Before moving on, though, it's worth looking back at the spam filter to see what lessons it teaches. Most important, make sure performance matters. It wouldn't have been worth all the effort if spam filtering wasn't a bottleneck. Once we knew it was a problem, we used profiling and other techniques to study the behavior and learn where the problem really lay. Then we made sure we were solving the right problem, examining the overall program rather than just focusing on `strstr`, the obvious but incorrect suspect. Finally, we solved the correct problem using a better algorithm, and checked that it really was faster. Once it was fast enough, we stopped; why over-engineer?

Exercise 7-1 . A table that maps a single character to the set of patterns that begin with that character gives an order of magnitude (巨大的) improvement. Implement a version of `isspam` that uses two characters as the index. How much improvement does that lead to? These are simple special cases of a data structure called a trie. Most such data structures are based on trading (交易) space for time. □

7.2 Timing and Profiling

Automate timing measurements. Most systems have a command to measure how long a program takes. On Unix, the command is called `time`:

```
% time slowprogram
real    7.0
user    6.2
sys     0.1
%
```

This runs the command and reports three numbers, all in seconds: "real" time, the elapsed time for the program to complete; "user" CPU time, time spent executing the user's program; and "system" CPU time, time spent within the operating system on the program's behalf. If your system has a similar command, use it; the numbers will be more informative, reliable, and easier to track than time measured with a stopwatch. And keep good notes. As you work on the program, making modifications and measurements, you will accumulate a lot of data that can become confusing a day or two later. (Which version was it that ran 20% faster?) Many of the techniques we discussed in the chapter on testing can be adapted for measuring and improving performance. Use the machine to run and measure your test suites and, most important, use regression (回归) testing to make sure your modifications don't break the program.

If your system doesn't have a `time` command, or if you're timing a function in isolation, it's easy to construct a timing scaffold (脚手架) analogous to a testing scaffold. C and C++ provide a standard routine, clock, that reports how much CPU time the program has consumed so far. It can be called before and after a function to measure CPU usage:

```
#include <time.h>
#include <stdio.h>
    ...
    clock_t before;
    double  elapsed;

    before = clock();
    long_running_function();
    elapsed = clock() - before;
```

```
printf("function used %.3f seconds\n",
        elapsed / CLOCKS_PER_SEC);
```

The scaling (定比) term, `CLOCKS_PER_SEC`, records the resolutions of the timer as reported by clock. If the function takes only a small fraction of a second, run it in a loop, but be sure to compensate (补偿) for loop overhead if that is significant:

```
before = clock();
for (i = 0; i < 1000; i++)
    short_running_function();
elapsed = (clock(0 - before) / (double)i;
```

In Java, functions in the `Date` class give wall clock time, which is an approximation to CPU time:

```
Date before = new Date();
long_running_function();
Date after = new Date();
long elapsed = after.getTime() - before.getTime();
```

The return value of `getTime` is in milliseconds.

Use a profiler (刻画器). Besides a reliable timing method, the most important tool for performance analysis is a system for generating profiles. A profile is a measurement of where a program spends its time. Some profiles list each function, the number of times it is called, and the fraction of execution time it consumes. Others show counts of how many times each statement was executed. Statements that are executed frequently contribute more to run-time, while statements that are never executed may indicate useless code or code that is not being tested adequately.

Profiling is an effective tool for finding *hot spots* in a program, the functions or sections of code that consume most of the computing time. Profiles should be interpreted with care, however. Given the sophistication of compilers and the complexity of caching and memory effects, as well as the fact that profiling a program affects its performance, the statistics in a profile can be only approximate.

In the 1971 paper that introduced the term profiling, Don Knuth wrote that "less than 4 percent of a program generally accounts for more than half of its running time." This indicates that the way to use profiling is to identify the critical time-consuming parts of the program, improve them to the degree possible, and then measure again to see if a new hot spot has surfaced. Eventually, often after only one or two iterations, there is no obvious hot spot left.

Profiling is usually enabled with a special compiler flat or option. The program is run, and then an analysis tool shows the results. On Unix, the flag is usually `-p` and the tool is called `prof`[1]:

```
% cc -p spamtest.c -o spamtest
% spamtest
% prof spamtest
```

The following table shows the profile generated by a special version of the spam filter we built to understand its behavior. It uses a fixed message and a fixed set of 217 phrases, which it matches against the message 10000 times. This run on a 250 MHz MIPS R 10000 used the original implementation of `strstr` that calls other standard functions. The output has been edited reformatted so it fits the page. Notice how sizes of input (217 phrases) and the number of runs (10000) show up as consistency checks in the "calls" column, which counts the number of calls of each function.

[1] `gprof` in Linux

```
12234768552: Total number of instructions executed
13961810001: Total computed cycles
55.847: Total computed execution time (secs)
1.141: Average cycles per instruction
```

secs	%	cum%	cycles	instructions	calls	function
45.260	81.0%	81.0%	11314990000	9440110000	48350000	strchr
6.081	10.9%	91.9%	1520280000	1566460000	46180000	strncmp
2.592	4.6%	94.6%	648080000	854500000	2170000	strstr
1.825	3.3%	99.8%	456225559	344882213	21704435	strlen
0.088	0.2%	100.0%	21950000	28510000	10000	isspam
0.000	0.0%	100.0%	100025	100028	1	main
0.000	0.0%	100.0%	53677	70268	219	_memcopy
0.000	0.0%	100.0%	48888	46403	217	strcpy
0.000	0.0%	100.0%	17989	19894	219	fgets
0.000	0.0%	100.0%	16798	17547	230	__malloc
0.000	0.0%	100.0%	10305	10900	204	realfree
0.000	0.0%	100.0%	6293	7161	217	estrdup
0.000	0.0%	100.0%	6032	8575	231	cleanfree
0.000	0.0%	100.0%	5932	5729	1	readpat
0.000	0.0%	100.0%	5899	6339	219	getline
0.000	0.0%	100.0%	5500	5720	220	_malloc

It's obvious that `strchr` and `strncmp`, both called by `strstr`, completely dominate the performance. Knuth's guideline is right: a small part of the program consumes most of the run-time. When a program is first profiled, it's common to see the top-running function at 50 percent or more, as it is here, making it easy to decide where to focus attention.

Concentrate on the hot spots. After rewriting `strstr`, we profiled `spamtest` again and found that 99.8% of the time was now spent in `strstr` alone, even though the whole program was considerable faster. When a single function is so overwhelmingly (压倒性的) the bottleneck, there are only two ways to go: improve the function to use a better algorithm, or eliminate the function altogether by rewriting the surrounding program.

In this case, we rewrote the program. Here are the first few lines of the profile for `spamtest` using the final, fast implementation of `isspam`. Notice that the overall time is much less, that `memcmp` is now the hot spot, and that `isspam` now consumes a significant fraction of the computation. It is more complex than the version that called `strstr`, but its cost is more than compensated for by eliminating `strlen` and `strchr` from `isspam` and by replacing `strncmp` with `memcmp`, which does less work per byte.

secs	%	cum%	cycles	instructions	calls	function
3.524	56.9%	56.9%	880890000	1027590000	46180000	memcmp
2.662	43.04%	100.0%	665550000	902920000	10000	isspam
0.001	0.0%	100.0%	140304	106043	652	strlen
0.000	0.0%	100.0%	100025	100028	1	main

It's instructive (有益的) to spend some time comparing the cycle counts and number of calls in the two profiles. Notice that `strlen` went from a couple of million calls to 652, and that strncmp and memcmp are called the same number of times. Also notice that `isspam`. which now incorporates (吸收) the function

of `strchr`, still manages to use far fewer cycles than `strchr` did before because it examines only the relevant patterns at each step. Many more details of the execution can be discovered by examining the numbers.

A hot spot can often be eliminated, or at least cooled, by much simpler engineering than we undertook for the spam filter. Long ago, a profile of Awk indicated that one function was begin called about a million times over the course of a regression (回归) test, in this loop:

```
?   for (j = i; j < MAXFLD; j++)
?       clear(j);
```

The loop, which clears fields before each new input line is read, was taking as much as 50 percent of the run-time. The constant `MAXFLD`. the maximum number of fields permitted in an input line, was 200. But in most uses of Awk, the actual number of fields was only two or three. Thus an enormous amount of time was being wasted clearing fields that had never been set. Replacing the constant by the previous value of the maximum number of fields gave a 25 percent overall speedup. The fix was to change the upper limit of the loop:

```
for (j = i; j < maxfld; j++)
    clear(j);
maxfld = i;
```

Draw a picture. Pictures are especially good for presentating performance measurements. They can convey information about the effects of parameter changes, compare algorithms and data structures, and sometimes point to unexpected behavior. The graphs of chain length counts for several hash multipliers in Chapter 5 showed clearly that some multipliers were better than others.

The following graph[2] shows the effect of the size of the hash table array on run-time for the C version of `markov` with Psalms as input (42685 words, 22482 prefixes). We did two experiments. One set of runs used array sizes that are powers of two from 2 to 16384; the other used sizes that are the largest prime less than each power of two. We wanted to see if a prime array size made any measurable difference to the performance.

The graph shows that run-time for this input is not sensitive to the table size once the size is above 1000 elements, nor is there a discernible (可辨别的) difference between prime and power-of-two table sizes.

Exercise 7-2 . Whether or not your system has a `time` command, use `clock` or `getTime` to write a timing facility for your own use. Compare its times t a wall clock. How does other activity on the machine affect the timing? □

Exercise 7-3 . In the first profile, `strchr` was called 48,350,000 times and `strncmp` only 46,180,000. Explain the difference. □

7.3 Strategies for Speed

Before changing a program to make it faster, be certain that it really is too slow, and use timing tools and profilers to discover where the time is going. Once you know what's happening, there are a number of strategies to follow. We list a few here in decreasing order of profitability (收益性).

[2] Lack of graphs, because the graph is too blurry and ugly to represent.

Use a better algorithm or data structure. The most important factor in making a program faster is the choice of algorithm and data structure; there can be a huge difference between an algorithm that is efficient and one that is not. Our spam filter saw a change in data structure that was worth a factor of ten; even greater improvement is possible if the new algorithm reduces the order (阶数) of computation, say from $O(n^2)$ to $O(n \log n)$. We covered this topic in Chapter 2, so we won't dwell on (详述) it here.

Be sure that the complexity is really what you expect; if not, there might be a hidden performance bug. This apparently linear algorithm for scanning a string,

```
?   for (i = 0; i < strlen(s); i++)
?       if (s[i] == c)
?           ...
```

is in fact quadratic (平方的): if `s` has `n` characters, each call to `strlen` walks down the `n` characters of the string and the loop is performed `n` times.

Enable compiler optimizations. One zero-cost change that usually produces a reasonable improvement is to turn on whatever optimization the compiler provides. Modern compilers do sufficiently well that they obviate (排除) much of the need for small-scale changes by programmers.

By default, most C and C++ compilers do not attempt much optimization. A compiler option enables the optimizer ("improver" would be a more accurate term). It should probably be the default except that the optimizations tend to confuse source-level debuggers, so programmers must enable the optimizer explicitly once they believe the program has been debugged.

Compiler optimization usually improves run-time anywhere from a few percent to a factor of two. Sometimes, though, it slows the program down, so measure the improvement before shipping (装运) your product. We compared unoptimized and optimized compilation on a couple of versions of the spam filter. For the test suite using the final version of the matching algorithm, the original run-time was 8.1 seconds, which dropped to 5.9 seconds when optimization was enabled, an improvement of over 25%. On the other hand, the version that used the fixed-up `strstr` showed no improvement under optimization, because `strstr` had already been optimized when it was installed in the library; the optimizer applies only to the source code being compiled now and not to the system libraries. However, some compilers have global optimizer -- which analyze the entire program for potential improvements. If such a compiler is available on your system, try it; it might squeeze out a few more cycles.

One thing to be aware of is that the more aggressively the compiler optimizes, the more likely it is to introduce bugs into the compiled program. After enabling the optimizer, re-run your regression test suite, as you should for any other modification.

Tune the code. The right choice of algorithm matters if data sizes are big enough. Furthermore, algorithmic improvements work across different machines, compilers and languages. But once the right algorithm is in place, if speed is still an issue the next thing to try is tuning the code: adjusting the details of loops and expressions to make things go faster.

The version of `isspam` we showed at the end of Section 7.1 hadn't been tuned. Here, we'll show what further improvements can be achieved by tweaking (扭) the loop. As a reminder, this is how we left it:

```
for (j = 0; (c = mesg[j]) != '\0'; j++) {
    for (i = 0; i < nstarting[c]; i++) {
        k = starting[c][i];
        if (memcmp(mesg+j, pat[k], patlen[k]) == 0) {
            printf("spam: match for '%s'\n", pat[k]);
            return 1;
```

```
            }
        }
    }
```

This initial version takes 6.6 seconds in our test suite when compiled using the optimizer. The inner loop has an array index (`nstarting[c]`) in its loop condition whose value is fixed for each iteration of the outer loop. We can avoid recalculating it by saving the value in a local variable:

```
for (j = 0; (c = mesg[j]) != '\0'; j++) {
    n = nstarting[c];
    for (i = 0; i < n; i++) {
        k = starting[c][i];
        ...
```

This drops the time to 5.9 seconds, about 10% faster, a speedup typical of what tuning can achieve. There's another variable we can pull out: `starting[c]` is also fixed. It seems like pulling that computation out of the loop would also help, but in our tests it made no measurable difference. This, too, is typical of tuning: some things help, some things don't, and one must measure to find out which. And results will vary with different machines or compilers.

There is another change we could make to the spam filter. The inner loop compares the entire pattern against the string, but the algorithm ensures that the first character already matches. We can therefore tune the code to start `memcmp` one byte further along. We tried this and found it gave about 3% improvement, which is slight but it requires modifying only three lines of the program, one of them in precomputation.

Don't optimize what doesn't matter. Sometimes tuning achieves nothing because it is applied where it makes no difference. Make sure the code you're optimizing is where time is really spent. The following story might be apocryphal (不可信的), but we'll tell it anyway. An early machine from a now-defunct (现已倒闭的) company was analyzed with a hardware performance monitor and discovered to be spending 50 percent of its time executing the same sequence of several instructions. The engineers built a special instruction to encapsulate the function of the sequence, rebuilt the system, and found it made no difference at all; they had optimized the idle loop of the operating system.

How much effort should you spend making a program run faster? The main criterion (标准) is whether the changes will yield enough to be worthwhile. As a guideline, the personal time spent making a program faster should not be more than the time the speedup will recover during the lifetime of the program. By this rule, the algorithmic improvement to `isspam` was worthwhile: it took a day of work but saved (and continues to save) hours every day. Removing the array index from the inner loop was less dramatic (戏剧性的) but still worth doing, since the program provides a service to a large community. Optimizing public services like the spam filter or a library is almost always worthwhile; speeding up test programs is almost never worthwhile. And for a program that runs for a year, squeeze out everything you can. It may be worth restarting if you find a way to make a ten percent improvement even after the program has been running for a month.

Competitive programs -- games, compilers. word processors, spreadsheets, database systems -- fall into this category as well, since commercial success is often to the swiftest, at least in published benchmark results.

It's important to time programs as changes are being made, to make sure that things are improving. Sometimes two changes that each improve a program will interact, negating (否定) their individual effects.

It's also the case that timing mechanisms can be so erratic (不稳定的) that it's hard to draw firm conclusions about the effect of changes. Even on single-user systems, times can fluctuate (波动) unpredictably. If the variability of the internal timer (or at least what is reported back to you) is ten percent, changes that yield improvements of only ten percent are hard to distinguish from noise.

7.4 Tuning the Code

There are many techniques to reduce run-time when a hot spot is found. Here are some suggestions, which should be applied with care, and with regression testing after each to be sure that the code still works. Bear in mind that good compilers will do some of these for you, and in fact you may impede (阻止) their efforts by complicating the program. Whatever you try, measure its effect to make sure it helps.

Collect common subexpressions. If an expensive computation appears multiple time, do it in only one place and remember the result. For example, in Chapter 1 we showed a macro that computed a distance by calling `sqrt` twice in a row with the same values; in effect the computation was

```
?   sqrt(dx*dx + dy*dy) + ((sqrt(dx*dx + dy*dy) > 0) ? ...)
```

Compute the square root once and use its value in two places.

If a computation is done within a loop but does not depend on anything that changes within the loop, move the computation outside, as when we replaced

```
    for (i = 0; i < nstarting[c]; i++) {
```

by

```
    n = nstarting[c];
    for (i = 0; i < n; i++) {
```

Replace expensive operations by cheap ones. The term *reduction in strength* refers to optimizations that replace an expensive operation by a cheaper one. In olden times, this used to mean replacing multiplications by additions or shifts, but that rarely buys much now. Division and remainder are much slower than multiplication, however, so there may be improvement if a division can be replaced with multiplication by the inverse, or a remainder by a masking operation if the divisor (除数) is a power of two. Replacing array indexing by pointers in C or C++ might speed things up, although most compilers do this automatically. Replacing a function call by a simpler calculation can still be worthwhile. Distance in the plane (平面) is determined by the formula `sqrt(dx*dx+dy*dy)`, so to decide which of two points is further away would normally involve calculating two square roots. But the same decision can be made by comparing the squares of the distances:

```
    if (dx1*dx1 + dy1*dy1 < dx2*dx2 + dy2*dy2)
        ...
```

gives the same result as comparing the square roots of the expressions.

Another instance occurs in textual pattern matchers such as our spam filter or `grep`. If the pattern begins with a literal character, a quick search is made down the input text for that character; if no match is found, the more expensive search machinery (结构)is not invoked at all.

Unroll (展开) or eliminate loops. There is a certain overhead in setting up and running a loop. If the body of the loop isn't too long and doesn't iterate too many times, it can be more efficient to write out each iteration in sequence. Thus, for example:

```
for (i = 0; i < 3; i++)
    a[i] = b[i] + c[i];
```

becomes

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
```

This eliminates loop overhead, particularly branching, which can slow modern processors by interrupting the flow of execution.

If the loop is longer, the same kind of transformation can be used to amortize (分摊) the overhead over fewer iterations:

```
for (i = 0; i < 3*n; i++)
    a[i] = b[i] + c[i];
```

becomes

```
for (i = 0; i < 3*n; i += 3) {
    a[i+0] = b[i+0] + c[i+0];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
}
```

Note that this works only if the length is a multiple of the step size; otherwise additional code is needed to fix up the ends, which is a place for mistakes to creep in (潜入) and for some of the efficiency to be lost again.

Cache frequently-used values. Cached values don't have to be recomputed. Caching takes advantage of *locality*, the tendency for programs (and people) to re-use recently accessed or nearby items in preference to (优先于) older or distant data. Computing hardware makes extensive use of caches; indeed. adding cache memory to a computer can make great improvements in how fast a machine appears. The same is true of software. Web browsers, for instance, cache pages and images to avoid the slow transfer of data over the Internet. In a print preview program we wrote years ago, non-alphabetic special characters like ½ had to be looked up in a table. Measurement showed that much of the use of special characters involved drawing lines with long sequences of the same single character. Caching just the single most recently used character made the program significantly faster on typical inputs.

It's best if the caching operation is invisible from outside, so that it doesn't affect the rest of the program except for making it run faster. Thus in the case of the print previewer, the interface to the character drawing function didn't change; it was always

```
drawchar(c);
```

The original version of `drawchar` called `show(lookup(c))`. The cache implementation used internal static variables to remember the previous character and its code:

```
if (c != lastc) { /* update the cache */
    lastc = c;
    lastc = lookup(c);
}
show(lastc);
```

Write a special-purpose allocator. Often the single hot spot in a program is memory allocation, which manifests (表明) itself as lots of calls on `malloc` or new. When most requests are for blocks of the same size, substantial (实质的) speedups are possible by replacing calls to the general-purpose allocator by calls to a special-purpose one. The special-purpose allocator makes one call to `malloc` to fetch a big array of items, then hands them out one at a time as needed, a cheaper operation. Freed items are placed back in a free list so they can be reused quickly.

If the requested sizes are similar, you can trade space for time by always allocating enough for the largest request. This can be effective for managing short strings if you use the same size for all strings up to a specified length.

Some algorithms can use stack-based allocation, where a whole sequence of allocations is done, and then the entire set is freed at once. The allocator obtains one big chunk for itself and treats it as a stack, pushing allocated items on as needed and popping them all off in a single operation at the end. Some C libraries offer a function `alloca` for this kind of allocation, though it is not standard. It uses the local call stack as the source of memory, and frees all the items when the function that calls `alloca` returns.

Buffer input and output. Buffering batches transactions so that frequent operations are done with as little overhead as possible, and the high-overhead operations are done only when necessary. The cost of an operation is thereby spread over multiple data values. When a C program calls `printf`, for example, the characters are stored in a buffer but not passed to the operating system until the buffer is full or flushed explicitly. The operating system itself may in turn (轮流) delay writing the data to disk. The drawback is the need to flush output buffers to make data visible; in the worst case, information still in a buffer will be lost if a program crashes.

Handle special cases separately. By handling same-sized objects in separate code, special-purpose allocators reduce time and space overhead in the general allocator and incidentally (顺便) reduce fragmentation. In the graphics library for the Inferno system, the basic draw function was written to be as simple and straightforward as possible. With that working, optimizations for a variety of cases (chosen by profiling) were added one at a time; it was always possible to test the optimized version against the simple one. In the end, only a handful of cases were optimized because the dynamic distribution of calls to the drawing function was heavily skewed towards (为了) displaying characters; it wasn't worth writing clever code for all the cases.

Precompute results. Sometimes it is possible to make a program run faster by precomputing values so they are ready when they are needed. We saw this in the spam filter, which precomputed `strlen(pat[i])` and stored it in the array at `patlen[i]`. If a graphics system needs to repeatedly compute a mathematical function like sine but only for a discrete (离散的) set of values, such as integer degrees, it will be faster to precompute a table with 360 entries (or provide it as data) and index into it as needed. This is an example of trading space for time. There are many opportunities to replace code by data or to do computation during compilation, to save time and sometimes space as well. For example, the `ctype` functions like `isdigit` are almost always implemented by indexing into a table of bit flags rather than by evaluating a sequence of tests.

Use approximate values. If accuracy isn't an issue, use lower-precision data types. On older or smaller machines, or machines that simulate floating point in software, single-precision floating-point arithmetic is often faster than double-precision, so use `float` instead of `double` to save time. Some modern graphics processors use a related trick. The IEEE floating-point standard requires "graceful underflow" as calculations approach the low end of representable values, but this is expensive to compute.

For images, the feature is unnecessary, and it is faster and perfectly acceptable to truncate to zero. This not only saves time when the numbers underflow, it can simplify the hardware for all arithmetic. The use of integer `sin` and `cos` routines is another example of using approximate values.

Rewrite in a lower-level language. Lower-level languages tend to be more efficient, although at a cost in programmer time. Thus rewriting some critical part of a C++ or Java program in C or replacing an interpreted script by a program in a compiled language may make it run much faster.

Occasionally, one can get significant speedups with machine-dependent code. This is a last resort (手段), not a step to be taken lightly, because it destroys portability and makes future maintenance and modifications much harder. Almost always, operations to be expressed in assembly language are relatively small functions that should be embedded in a library; `memset` and `memmove`, or graphics operations, are typical examples. The approach is to write the code as cleanly as possible in a high-level language and make sure it's correct by testing it as we described for `memset` in Chapter 6. This is your portable version, which will work everywhere, albeit (虽然) slowly. When you move to a new environment, you can start with a version that is known to work. Now when you write an assembly-language version, test it exhaustively (彻底地) against (针对) the portable one. When bugs occur, non-portable code is always suspect: it's comforting (令人鼓舞的) to have a comparison implementation.

Exercise 7-4 . One way to make a function like `memset` run faster is to have it write in word-sized chunks instead of byte-sized; this is likely to match the hardware better and might reduce the loop overhead by a factor of four or eight. The downside is that there are now a variety of end effects to deal with if the target is not aligned on a word boundary and if the length is not a multiple of the word size. Write a version of `memset` that does this optimization. Compare its performance to the existing library version and to a straightforward byte-at-a-time loop. □

Exercise 7-5 . Write a memory allocator `smalloc` for C strings that uses a special-purpose allocator for small strings but calls `malloc` directly for large ones. You will need to define a `struct` to represent the strings in either case. How do you decide where to switch from calling `smalloc` to `malloc`? □

7.5 Space Efficiency

Memory used to be the most precious computing resource, always in short supply, and much bad programming was done in an attempt to squeeze the most out of what little there was. The infamous (声名狼藉的) "Year 2000 Problem" is frequently cited as an example of this; when memory was truly scarce (稀有), even the two bytes needed to store 19 were deemed (认为) too expensive. Whether or not space is the true reason for the problem -- such code may simply reflect the way people use dates in everyday life, where the century is commonly omitted -- it demonstrates the danger inherent (固有的) in short-sighted (短浅的) optimization.

In any case, times have changed, and both main memory and secondary storage are amazingly cheap. Thus the first approach to optimizing space should be the same as to improving speed: don't bother.

There are still situations, however, where space efficiency matters. If a program doesn't fit into the available main memory, parts of it will be paged out, and that will make its performance unacceptable. We see this when new versions of software squander (使分散) memory; it is a sad reality that software upgrades are often followed by the purchase of more memory.

Save space by using the smallest-possible data type. One step to space efficiency is to make minor changes to use existing memory better, for example by using the smallest data type that will work. This might mean replacing `int` with `short` if the data will fit; this is a common technique for coordinates (协调) in 2-D graphics systems, since 16 bits are likely to handle any expected range of screen coordinates. Or it might mean replacing `double` with `float`; the potential problem is loss of precision, since `floats` usually hold only 6 or 7 decimal digits.

In these cases and analogous ones, other changes may be required as well, notably (特别是) format specifications in `printf` and especially `scanf` statements.

The logical extension of this approach is to encode information in a byte or even fewer bits, say a single bit where possible. Don't use C or C++ bit-fields; they are highly non-portable and tend to generate voluminous (长篇的) and inefficient code. Instead, encapsulate the operations you want in functions that fetch and set individual bits within words or an array of words with shift and mask operations. This function returns a group of contiguous bits from the middle of a word:

```
/* getbits: get n bits from position p */
/* bits are numbered from 0 (least significant) up */
unsigned int getbits(unsigned int x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

If such functions turn out to be too slow, they can be improved with the techniques described earlier in this chapter. In C++, operator overloading can be used to make bit accesses look like regular subscripting.

Don't store what you can easily recompute. Changes like these are minor, however; they are analogous to code tuning. Major improvements are more likely to come from better data structures, perhaps coupled with algorithm changes. Here's an example. Many years ago, one of us was approached (商量) by a colleague who was trying to do a computation on a matrix so large that it was necessary to shut down the machine and reload a stripped-down (精简的) operating system so the matrix would fit. He wanted to know if there was an alternative, since this was an operational nightmare. We asked what the matrix was like, and learned that it contained integer values, *most of which were zero.* In fact, fewer than five percent of the matrix elements were non-zero. This immediately suggested a representation in which only the non-zero elements of the matrix were stored, and each matrix access like `m[i][j]` would be replaced by a function call `m(i,j)`. There are several ways to store the data; the easiest is probably an array of pointers, one for each row, each of which points to a compact array of column numbers and corresponding values. This has higher space overhead per non-zero item but requires much less space overall, and although individual accesses will be slower, they will be noticeably faster than reloading the operating system. To complete the story: the colleague applied the suggestion and went away completely satisfied.

We used a similar approach to solve a modern version of the same problem. A radio design system needed to represent terrain (地形) and radio signal strengths over a very large geographical area (100 to 200 kilometers on a side) to a resolution of 100 meters. Storing this as a large rectangular array exceeded the memory available on the target machine and would have caused unacceptable paging behavior. But over large regions, the terrain and signal strength values are likely to be the same, so a hierarchical representation that coalesces (联合) regions of the same value into a single cell makes the problem manageable.

Variations on this theme are frequent, and so are specific representations, but all share the same basic idea: store the common value or values implicitly or in a compact form, and spend more time and space on the remaining values. If the most common values are really common, this is a win.

The program should be organized so that the specific data representation of complex types is hidden in a class or set of functions operating on a private data type. This precaution (预防措施) ensures that the rest of the program will not be affected if the representation changes.

Space efficiency concerns sometimes manifest (表现) themselves in the external representation of information as well, both conversion (转换) and storage. In general, it is best to store information as text wherever feasible rather than in some binary representation. Text is portable, easy to read, and amenable (服从的) to processing by all kinds of tools; binary representations have none of these advantages. The argument in favor of binary is usually based on "speed," but this should be treated with some skepticism (批判思想), since the disparity (差异) between text and binary forms may not be all that great.

Space efficiency often comes with a cost in run-time. One application had to transfer a big image from one program to another. Images in a simple format called PPM were typically a megabyte, so we thought it would be much faster to encode them for transfer in the compressed GIF format instead; those files were more like 50K bytes. But the encoding and decoding of GIF took as much time as was saved by transferring a shorter file, so nothing was gained. The code to handle the GIF format is about 500 lines long; the PPM source is about 10 lines. For ease of maintenance, therefore, the GIF encoding was dropped and the application continues to use PPM exclusively. Of course the tradeoff would be different if the file were to be sent across a slow network instead; then a GIF encoding would be much more cost-effective.

7.6 Estimation

It's hard to estimate ahead of time how fast a program will be, and it's doubly hard to estimate the cost of specific language statements or machine instructions. It's easy, though, to create a cost model for a language or a system, which will give you at least a rough idea of how long a important operations take.

One approach that is often used for conventional programming languages is a program that times representative code sequences. There are operational difficulties, like getting reproducible results and canceling out irrelevant overheads, but it is possible to get useful insights without much effort. For example, we have a C and C++ cost model program that estimates the costs of individual statements by enclosing them in a loop that runs them many millions of times, then computes an average time. On a 250 MHz MIPS R10000, it produces this data, with times in nanoseconds per operation.

```
int operations:
    i1++                    8
    i1 = i2 + i3           12
    i1 = i2 - i3           12
    i1 = i2 * i3           12
    i1 = i2 / i3          114
    i1 = i2 % i3          114

float operations
    f1 = f2                 8
    f1 = f2 + f3           12
    f1 = f2 - f3           12
    f1 = f2 * f3           11
    f1 = f2 / f3           28

double operations
    d1 = d2                 8
    d1 = d2 + d3           12
```

```
    d1 = d2 - d3           12
    d1 = d2 * d3           11
    d1 = d2 / d3           58

numeric conversions
    i1 = f1                 8
    f1 = i1                 8
```

Integer operations are fast, except for division and modulus (取模). Floating-point operations are as fast or faster, a surprise to people who grew up at a time when floating point operations were much more expensive than integer operations.

Other basic operations are also quite fast, including function calls, the last three lines of this group:

```
integer vector operations
    v[i] = i               49
    v[v[i]] = i            81
    v[v[v[i]]] = i        100

control structures
    if (i == 5) i1++        4
    if (i != 5) i1++       12
    while (i < 0) i1++      3
    i1 = sum1(i2)          57
    i1 = sum2(i2, i3)      58
    i1 = sum3(i2, i3, i4)  54
```

But input and output are not so cheap, nor are most other library function:

```
input/output
    fputs(s, fp)                        270
    fgets(s, 9, fp)                     222
    fprintf(fp, "%d\n", i)             1820
    fscanf(fp, "%d", &i1)              2070

malloc
    free(malloc(8))                     342

string functions
    strcpy(s, "0123456789")             157
    i1 = strcmp(s, s)                   176
    i1 = strcmp(s, "a123456789")         64

string/number conversions
    i1 = atoi("12345")                  402
    sscanf("12345", "%d", &i1)         2376
    sprintf(s, "%d", i)                1492
    f1 = atof("123.45")                4098
    sscanf("123.45", "%f", &f1)        6438
    sprintf(s, "%6.2f", 123.45)        3902
```

The time for `malloc` and `free` are probably not indicative of true performance, since freeing immediately after allocating is not a typical pattern.

Finally, math functions:

```
math functions
    i1 = rand()             135
    f1 = log(f2)            418
```

```
f1 = exp(f2)            462
f1 = sin(f2)            514
f1 = sqrt(f2)           112
```

These values would be different on different hardware, of course, but the trends can be used for back-of-the-envelope (粗略) estimates of how long something might take, or for comparing the relative costs of I/O versus basic operations, or for deciding whether to rewrite an expression or use an inline function.

There are many sources of variability. One is compiler optimization level. Modern compilers can find optimizations that elude most programmers. Furthermore, current CPUs are so complicated that only a good compiler can take advantage of their ability to issue multiple instructions concurrently, pipeline their execution, fetch instructions and data before they are needed, and the like.

Computer architecture itself is another major reason why performance numbers are hard to predict. Memory caches make a great difference in speed, and much cleverness in hardware design goes into hiding the fact that main memory is quite a bit slower than cache memory. Raw processor clock rates (频率) like "400 MHz" are suggestive (提示性的) but don't tell the whole story; one of our old 200 MHz Pentium is significantly slower than an even older 100 MHz Pentium because the latter has a big second-level cache and the former has none. And different generations of processor, even for the same instruction set, take different numbers of clock cycles to do a particular operation.

Exercise 7-6 . Create a set of tests for estimating the costs of basic operations for computers and compilers near you, and investigates and differences in performance. □

Exercise 7-7 . Create a cost model for higher-level operations in C++. Among the features that might be included are construction, copying, and deletion of class objects; member function calls; virtual functions; inline functions; the `iostream` library; the STL. This exercise is open-ended (开放式的), so Concentrate on a small set of representative operations. □

Exercise 7-8 . Repeat the previous exercise for Java. □

7.7 Summary

Once you have chosen the right algorithm, performance optimization is generally the last thing to worry about as you write a program. If you must undertake it, however, the basic cycle is to measure, focus on the few places where a change will make the most difference, verify the correctness of your changes, then measure again. Stop as soon as you can, and preserve the simplest version as a baseline (基准) for timing and correctness.

When you're trying to improve the speed or space consumption of a program, it's a good idea to make up some benchmark tests and problems so you can estimate and keep track of performance for yourself. If there are already standard benchmarks for your task, use them too. If the program is relatively self-contained, one approach is to find or create a collection of "typical" inputs; these might well be part of a test suite as well. This is the genesis (起源) of benchmark suites for commercial and academic systems like compilers, computers, and the like. For example, Awk comes with about 20 small programs that together cover most of the commonly-used language features; these programs are run over a very large input file to assure that the same results are computed and that no performance bug has been introduced. We also

have a collection of standard large data files that can be used for timing tests. In some cases it might help that such files have easily verified properties, for example a size that is a power of ten or of two.

Benchmarking can be managed with the same kind of scaffolding (脚手架) as we recommended for testing in Chapter 6. Timing tests are run automatically; outputs include enough identification that they can be understood and replicated (复制); records are kept so that trends and significant changes can be observed.

By the way, it's extremely difficult to do good benchmarking, and it is not unknown for companies to tune their products to show up well on benchmarks, so it is wise to take all benchmark results with a grain of salt.

Supplementary Reading

Our discussion of the spam filter is based on work by Bob Handrena and Ken Thompson. Their filter includes regular expressions for more sophisticated matching and automatically classifies messages (certainly spam, possibly spam, not spam) according to the strings they match.

Knuth's profiling paper, "An Empirical (经验主义的) Study of FORTRAN Programs," appeared in *Software -- Practice and Experience, 1, 2*, PP.105-133, 1971. The core of the paper is a statistical analysis of a set of programs found by rummaging (仔细搜索) in waste basket (篮子) and publicly-visible directories on the computer center's machines.

Jon Bentley's *Programming Pearls* and *More Programming Pearls* (Addison-Wesley, 1986 and 1988) have several fine example of algorithmic and code-tuning improvements; there are also good essays (短文) on scaffolds for performance improvements and the use of profiles.

Inner Loops, by Rick Booth (Addison-Wesley, 1997), is a good reference on tuning PC programs, although processors evolve so fast that specific details age quickly.

John Hennessy and David Patterson's family of books on computer architecture (for example, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufman, 1997) contain thorough discussions of performance considerations for modern computers.

Chapter 8

Portability

Chapter 9

Notation