

5. The best times for sorting n elements are $O(n \log n)$
6. DP algorithms which involves filling in a matrix usually in $O(n^3)$
7. In contest, most of the time $O(n \log n)$ algorithms will be sufficient.

The art of testing your code

You've done it. Identifying the problem, designing the best possible algorithm, you have calculate using time/space complexity, that it will be within time and memory limit given., and you have code it so well. But, is it 100% correct?

Depends on the programming contest's type, you may or may not get credit by solving the problem partially. In ACM ICPC, you will only get credit if your team's code solve all the test cases, that's it, you'll get either Accepted or Not Accepted (Wrong Answer, Time Limit Exceeded, etc). In IOI, there exist partial credit system, in which you will get score: number of correct/total number of test cases for each code that you submit.

In either case, you will need to be able to design a good, educated, tricky test cases. Sample input-output given in problem description is by default too trivial and therefore not a good way to measure your code's correctness for all input instances.

Rather than wasting your submission (and gaining time or points penalty) by getting wrong answer, you may want to design some tricky test cases first, test it in your own machine, and ensure your code is able to solve it correctly (otherwise, there is no point submitting your solution right?).

Some team coaches sometime ask their students to compete with each other by designing test cases. If student A's test cases can break other student's code, then A will get bonus point. You may want to try this in your school team training too.

Here is some guidelines in designing good test cases:

1. Must include sample input, the most trivial one, you even have the answer given.
2. Must include boundary cases, what is the maximum n, x, y , or other input variables, try varying their values to test for out of bound errors.
3. For multiple input test case, try using two identical test case consecutively. Both must output the same result. This is to check whether you forgot to initialize some variables, which will be easily identified if the first instance produce correct output but the second one doesn't.
4. Increase the size of input. Sometimes your program works for small input size, but behave wrongly when input size increases.
5. Tricky test cases, analyze the problem description and identify parts that are tricky, test them to your code.
6. Don't assume input will always nicely formatted if the problem description didn't say

so. Try inserting white spaces (space, tabs) in your input, check whether your code is able to read in the values correctly

7. Finally, do random test cases, try random input and check your code's correctness.

Producing Winning Solution

A good way to get a competitive edge is to write down a game plan for what you're going to do in a contest round. This will help you script out your actions, in terms of what to do both when things go right and when things go wrong. This way you can spend your thinking time in the round figuring out programming problems and not trying to figure out what the heck you should do next... it's sort of like Pre-Computing your reactions to most situations.

Read through all the problems first, don't directly attempt one problem since you may missed easier problem.

1. Order the problems: shortest job first, in terms of your effort (shortest to longest: done it before, easy, unfamiliar, hard).
2. Sketch the algorithms, complexity, the numbers, data structures, tricky details.
3. Brainstorm other possible algorithms (if any) - then pick the stupidest that works!
4. Do the Math! (space & time complexity & plug-in actual expected & worst case numbers).
5. Code it of course, as fast as possible, and it must be correct.
6. Try to break the algorithm - use special (degenerate?) test cases.

Coding a problem

1. Only coding after you finalize your algorithm.
2. Create test data for tricky cases.
3. Code the input routine and test it (write extra output routines to show data).
4. Code the output routine and test it.
5. Write data structures needed.
6. Stepwise refinement: write comments outlining the program logic.
7. Fill in code and debug one section at a time.
8. Get it working & verify correctness (use trivial test cases).
9. Try to break the code - use special cases for code correctness.

Time management strategy and "damage control" scenarios

Have a plan for what to do when various (foreseeable!) things go wrong; imagine problems you might have and figure out how you want to react. The central question is:

"When do you spend more time debugging a program, and when do you cut your losses and move on?". Consider these issues:

1. How long have you spent debugging it already?
2. What type of bug do you seem to have?
3. Is your algorithm wrong?
4. Do your data structures need to be changed?
5. Do you have any clue about what's going wrong?
6. A short amount (20 mins) of debugging is better than switching to anything else; but you might be able to solve another from scratch in 45 mins.
7. When do you go back to a problem you've abandoned previously?
8. When do you spend more time optimizing a program, and when do you switch?
9. Consider from here out - forget prior effort, focus on the future: how can you get the most points in the next hour with what you have?

Tips & tricks for contests

1. Brute force when you can, Brute force algorithm tends to be the easiest to implement.
2. KISS: Simple is smart! (Keep It Simple, Stupid !!! / Keep It Short & Simple).
3. Hint: focus on limits (specified in problem statement).
4. Waste memory when it makes your life easier (trade memory space for speed).
5. Don't delete your extra debugging output, comment it out.
6. Optimize progressively, and only as much as needed.
7. Keep all working versions!
8. Code to debug:
 - a. white space is good,
 - b. use meaningful variable names,
 - c. don't reuse variables, (we are not doing software engineering here)
 - d. stepwise refinement,
 - e. Comment before code.
9. Avoid pointers if you can.
10. Avoid dynamic memory like the plague: statically allocate everything. (yeah yeah)
11. Try not to use floating point; if you have to, put tolerances in everywhere (never test equality)
12. Comments on comments:
 - a. Not long prose, just brief notes.
 - b. Explain high-level functionality: `++i; /* increase the value of i by */` is worse than useless.
 - c. Explain code trickery.
 - d. Delimit & document functional sections.

Keep a log of your performance in each contest: successes, mistakes, and what you could have done better; use this to rewrite and improve your game plan!

The Judges Are Evil and Out to Get You

Judges don't want to put easy problems on the contest, because they have thought up too many difficult problems. So what we do is hide the easy problems, hoping that you will be tricked into working on the harder ones. If we want you to add two non-negative numbers together, there will be pages of text on the addition of '0' to the number system and 3D-pictures to explain the process of addition as it was imagined on some island that nobody has ever heard of.

Once we've scared you away from the easy problems, we make the hard ones look easy. 'Given two polygons, find the area of their intersection.' Easy, right?

It isn't always obvious that a problem is easy, so teams ignore the problems or start on overly complex approaches to them. Remember, there are dozens of other teams working on the same problems, and they will help you find the easy problems. **If everyone is solving problem G, maybe you should take another look at it.**^[6]

CHAPTER 3 PROGRAMMING IN C: A TUTORIAL^[11]

C was created by **Dennis Ritchie** at the **Bell Telephone Laboratories** in 1972. Because C is such a powerful and flexible language, its use quickly spread beyond Bell Labs. Programmers everywhere began using it to write all sorts of programs. Soon, however, different organizations began utilizing their own versions of C, and subtle differences between implementations started to cause programmers headaches. In response to this problem, the **American National Standards Institute (ANSI)** formed a committee in 1983 to establish a standard definition of C, which became known as ANSI Standard C. With few exceptions, every modern C compiler has the ability to adhere this standard [11].

A Simple C Program

A C program consists of one or more functions, which are similar to the functions and subroutines of a Fortran program or the procedures of PL/I, and perhaps some external data definitions. `main` is such a function, and in fact all C programs must have a `main`. Execution of the program begins at the first statement of `main`. `main` will usually invoke other functions to perform its job, some coming from the same program, and others from libraries.

```
main( ) {  
    printf("hello, world");  
}
```

`printf` is a library function which will format and print output on the terminal (unless some other destination is specified). In this case it prints

```
hello, world
```

A Working C Program; Variables; Types and Type Declarations

Here's a bigger program that adds three integers and prints their sum.

```
main( ) {  
    int a, b, c, sum;  
    a = 1;  b = 2;  c = 3;  
    sum = a + b + c;  
    printf("sum is %d", sum);  
}
```

Arithmetic and the assignment statements are much the same as in Fortran (except for the semicolons) or PL/I. The format of C programs is quite free. We can put several

statements on a line if we want, or we can split a statement among several lines if it seems desirable. The split may be between any of the operators or variables, but *not* in the middle of a name or operator. As a matter of style, spaces, tabs, and newlines should be used freely to enhance readability.

C has fundamental types of variables:

Variable Type	Keyword	Bytes Required	Range
Character	char	1	-128 to 127
Integer	int	2	-32768 to 32767
Short integer	short	2	-32768 to 32767
Long integer	long	4	-2,147,483,648 to 2,147,438,647
Unsigned character	unsigned char	1	0 to 255
Unsigned integer	unsigned int	2	0 to 65535
Unsigned short integer	unsigned short	2	0 to 65535
Unsigned long integer	unsigned long	4	0 to 4,294,967,295
Single-precision	float	4	1.2E-38 to
floating-point			3.4E38
Double-precision	double	8	2.2E-308 to
floating-point			1.8E308 ²

There are also arrays and structures of these basic types, pointers to them and functions that return them, all of which we will meet shortly.

All variables in a C program must be declared, although this can sometimes be done implicitly by context. Declarations must precede executable statements. The declaration

```
int a, b, c, sum;
```

Variable names have one to eight characters, chosen from A-Z, a-z, 0-9, and `_`, and start with a non-digit.

Constants

We have already seen decimal integer constants in the previous example-- 1, 2, and 3. Since C is often used for system programming and bit-manipulation, octal numbers are an important part of the language. In C, any number that begins with 0 (zero!) is an octal integer (and hence can't have any 8's or 9's in it). Thus `0777` is an octal constant, with decimal value 511.

A `''character''` is one byte (an inherently machine-dependent concept). Most often this is expressed as a character constant, which is one character enclosed in single quotes. However, it may be any quantity that fits in a byte, as in flags below:

```
char quest, newline, flags;  
quest = '?';  
newline = '\n';  
flags = 077;
```

The sequence `''\n''` is C notation for `''newline character''`, which, when printed, skips the terminal to the beginning of the next line. Notice that `''\n''` represents only a single character. There are several other `''escapes''` like `''\n''` for representing hard-to-get or invisible characters, such as `''\t''` for tab, `''\b''` for backspace, `''\0''` for end of file, and `''\\''` for the backslash itself.

Simple I/O – `getchar()`, `putchar()`, `printf ()`

`getchar` and **`putchar`** are the basic I/O library functions in C. `getchar` fetches one character from the standard input (usually the terminal) each time it is called, and returns that character as the value of the function. When it reaches the end of whatever file it is reading, thereafter it returns the character represented by `''\0''` (ascii NUL, which has value zero). We will see how to use this very shortly.

```
main( ) {  
    char c;  
    c = getchar( );  
    putchar(c);  
}
```

`putchar` puts one character out on the standard output (usually the terminal) each time it is called. So the program above reads one character and writes it back out. By itself, this isn't very interesting, but observe that if we put a loop around this, and add a test for end of file, we have a complete program for copying one file to another.

`printf` is a more complicated function for producing formatted output.

```
printf ("hello, world\n");
```

is the simplest use. The string `''hello, world\n''` is printed out.

More complicated, if `sum` is 6,

```
printf ("sum is %d\n", sum);
```

prints

```
sum is 6
```

Within the first argument of `printf`, the characters ```%d``` signify that the next argument in the argument list is to be printed as a base 10 number.

Other useful formatting commands are ```%c``` to print out a single character, ```%s``` to print out an entire string, and ```%o``` to print a number as octal instead of decimal (no leading zero). For example,

```
n = 511;
printf ("What is the value of %d in octal?", n);
printf ("%s! %d decimal is %o octal\n", "Right", n, n);
```

prints

```
What is the value of 511 in octal? Right! 511 decimal
is 777 octal
```

If - relational operators and compound statements

The basic conditional-testing statement in C is the `if` statement:

```
c = getchar( );
if( c == '?' )
    printf("why did you type a question mark?\n");
```

The simplest form of **`if`** is

```
if (expression) statement
```

The condition to be tested is any expression enclosed in parentheses. It is followed by a statement. The expression is evaluated, and if its value is non-zero, the statement is executed. There's an optional **`else`** clause, to be described soon.

The character sequence ```==``` is one of the relational operators in C; here is the complete set:

```
==      equal to (.EQ. to Fortraners)
!=      not equal to
>       greater than
```



```
<      less than
>=     greater than or equal to
<=     less than or equal to
```

The value of "`expression relation expression`" is 1 if the relation is true, and 0 if false. Don't forget that the equality test is `==`; a single `=` causes an assignment, not a test, and invariably leads to disaster.

Tests can be combined with the operators `&&` (AND), `||` (OR), and `!` (NOT). For example, we can test whether a character is blank or tab or newline with

```
if( c==' ' || c=='\t' || c=='\n' ) ...
```

C guarantees that `&&` and `||` are evaluated left to right -- we shall soon see cases where this matters.

As a simple example, suppose we want to ensure that `a` is bigger than `b`, as part of a sort routine. The interchange of `a` and `b` takes three statements in C, grouped together by `{}`:

```
if (a < b) {
    t = a;
    a = b;
    b = t;
}
```

As a general rule in C, anywhere you can use a simple statement, you can use any compound statement, which is just a number of simple or compound ones enclosed in `{}`. There is no semicolon after the `}` of a compound statement, but there *is* a semicolon after the last non-compound statement inside the `{}`.

While Statement; Assignment within an Expression; Null Statement

The basic looping mechanism in C is the while statement. Here's a program that copies its input to its output a character at a time. Remember that `'\0'` marks the end of file.

```
main( ) {
    char c;
    while( (c=getchar( )) != '\0' )
        putchar(c);
}
```

The while statement is a loop, whose general form is

```
while (expression) statement
```

Its meaning is

- (a) evaluate the expression
- (b) if its value is true (i.e., not zero) do the statement, and go back to (a)

Because the expression is tested before the statement is executed, the statement part can be executed zero times, which is often desirable. As in the if statement, the expression and the statement can both be arbitrarily complicated, although we haven't seen that yet. Our example gets the character, assigns it to `c`, and then tests if it's a `'\0'`. If it is not a `'\0'`, the statement part of the while is executed, printing the character. The while then repeats. When the input character is finally a `'\0'`, the while terminates, and so does main.

Notice that we used an assignment statement

```
c = getchar( )
```

within an expression. This is a handy notational shortcut which often produces clearer code. (In fact it is often the only way to write the code cleanly. As an exercise, rewrite the file-copy without using an assignment inside an expression.) It works because an assignment statement has a value, just as any other expression does. Its value is the value of the right hand side. This also implies that we can use multiple assignments like

```
x = y = z = 0;
```

Evaluation goes from right to left.

By the way, the extra parentheses in the assignment statement within the conditional were really necessary: if we had said

```
c = getchar( ) != '\0'
```

`c` would be set to 0 or 1 depending on whether the character fetched was an end of file or not. This is because in the absence of parentheses the assignment operator `'='` is evaluated after the relational operator `'!='`. When in doubt, or even if not, parenthesize.

```
main( ) {  
    while( putchar(getchar( )) != '\0' ) ;  
}
```

What statement is being repeated? None, or technically, the null statement, because all the work is really done within the test part of the while. This version is slightly different from the previous one, because the final `\0` is copied to the output before we decide to stop.

Arithmetic

The arithmetic operators are the usual `+`, `-`, `*`, and `/` (truncating integer division if the operands are both int), and the remainder or mod operator `%`:

```
x = a % b;
```

sets `x` to the remainder after `a` is divided by `b` (i.e., `a mod b`). The results are machine dependent unless `a` and `b` are both positive.

In arithmetic, char variables can usually be treated like int variables. Arithmetic on characters is quite legal, and often makes sense:

```
c = c + 'A' - 'a';
```

converts a single lower case ascii character stored in `c` to upper case, making use of the fact that corresponding ascii letters are a fixed distance apart. The rule governing this arithmetic is that all chars are converted to int before the arithmetic is done. Beware that conversion may involve sign-extension if the leftmost bit of a character is 1, the resulting integer might be negative. (This doesn't happen with genuine characters on any current machine.)

So to convert a file into lower case:

```
main( ) {
    char c;
    while( (c=getchar( )) != '\0' )
        if( 'A' <= c && c <= 'Z' )
            putchar(c + 'a' - 'A');
        else
            putchar(c); }
```

Else Clause; Conditional Expressions

We just used an `else` after an `if`. The most general form of `if` is

```
if (expression) statement1 else statement2
```

the `else` part is optional, but often useful. The canonical example sets `x` to the minimum of `a` and `b`:

```
if (a < b)
    x = a;
else
    x = b;
```

C provides an alternate form of conditional which is often more concise. It is called the "conditional expression" because it is a conditional which actually has a value and can be used anywhere an expression can. The value of

```
a < b ? a : b;
```

is `a` if `a` is less than `b`; it is `b` otherwise. In general, the form

```
expr1 ? expr2 : expr3
```

To set `x` to the minimum of `a` and `b`, then:

```
x = (a < b ? a : b);
```

The parentheses aren't necessary because `'?:'` is evaluated before `'='`, but safety first.

Going a step further, we could write the loop in the lower-case program as

```
while( (c=getchar( )) != '\0' )
    putchar( ('A'<=c && c<='Z') ? c-'A'+'a' : c );
```

If's and else's can be used to construct logic that branches one of several ways and then rejoins, a common programming structure, in this way:

```
if(...)
    {...}
else if(...)
    {...}
else if(...)
    {...}
else
    {...}
```

The conditions are tested in order, and exactly one block is executed; either the first one whose if is satisfied, or the one for the last `else`. When this block is finished, the next statement executed is the one after the last `else`. If no action is to be taken for the "default" case, omit the last `else`.

For example, to count letters, digits and others in a file, we could write

```
main( ) {
    int let, dig, other, c;
    let = dig = other = 0;
    while( (c=getchar( )) != '\0' )
        if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )
            ++let;
        else if( '0'<=c && c<='9' ) ++dig;
        else ++other;
    printf("%d letters, %d digits, %d others\n", let, dig, other);
}
```

This code letters, digits and others in a file.

Increment and Decrement Operators

In addition to the usual '-', C also has two other interesting unary operators, '++' (increment) and '--' (decrement). Suppose we want to count the lines in a file.

```
main( ) {
    int c,n;
    n = 0;
    while( (c=getchar( )) != '\0' )
        if( c == '\n' )
            ++n;
    printf("%d lines\n", n);
}
```

++n is equivalent to n=n+1 but clearer, particularly when n is a complicated expression. '++' and '--' can be applied only to int's and char's (and pointers which we haven't got to yet).

The unusual feature of '++' and '--' is that they can be used either before or after a variable. The value of ++k is the value of k *after* it has been incremented. The value of k++ is k *before* it is incremented. Suppose k is 5. Then

```
x = ++k;
```

increments k to 6 and then sets x to the resulting value, i.e., to 6. But

```
x = k++;
```

first sets *x* to 5, and *then* increments *k* to 6. The incrementing effect of *++k* and *k++* is the same, but their values are respectively 5 and 6. We shall soon see examples where both of these uses are important.

Arrays

In C, as in Fortran or PL/I, it is possible to make arrays whose elements are basic types. Thus we can make an array of 10 integers with the declaration

```
int x[10];
```

The square brackets mean subscripting; parentheses are used only for function references. Array indexes begin at zero, so the elements of *x* are

```
x[0], x[1], x[2], ..., x[9]
```

If an array has *n* elements, the largest subscript is *n*-1.

Multiple-dimension arrays are provided, though not much used above two dimensions. The declaration and use look like

```
int name[10] [20];  
n = name[i+j] [1] + name[k] [2];
```

Subscripts can be arbitrary integer expressions. Multi-dimension arrays are stored by row (opposite to Fortran), so the rightmost subscript varies fastest; *name* has 10 rows and 20 columns.

Here is a program which reads a line, stores it in a buffer, and prints its length (excluding the newline at the end).

```
main( ) {  
    int n, c;  
    char line[100];  
    n = 0;  
    while( (c=getchar( )) != '\n' ) {  
        if( n < 100 )  
            line[n] = c;  
        n++;  
    }  
    printf("length = %d\n", n);  
}
```

As a more complicated problem, suppose we want to print the count for each line in the input, still storing the first 100 characters of each line. Try it as an exercise before looking at the solution:

```
main( ) {
    int n, c; char line[100];
    n = 0;
    while( (c=getchar( )) != '\0' )
        if( c == '\n' ) {
            printf("%d", n);
            n = 0;
        }
        else {
            if( n < 100 ) line[n] = c;
            n++;
        }
}
```

Above code stores first 100 characters of each line!

Character Arrays; Strings

Text is usually kept as an array of characters, as we did with `line[]` in the example above. By convention in C, the last character in a character array should be a `'\0'` because most programs that manipulate character arrays expect it. For example, `printf` uses the `'\0'` to detect the end of a character array when printing it out with a `'%s'`.

We can copy a character array `s` into another `t` like this:

```
i = 0;
while( (t[i]=s[i]) != '\0' )
    i++;
```

Most of the time we have to put in our own `'\0'` at the end of a string; if we want to print the line with `printf`, it's necessary. This code prints the character count before the line:

```
main( ) {
    int n;
    char line[100];
    n = 0;
    while( (line[n++]=getchar( )) != '\n' );
    line[n] = '\0';
    printf("%d:\t%s", n, line);
}
```

Here we increment *n* in the subscript itself, but only after the previous value has been used. The character is read, placed in *line[n]*, and only then *n* is incremented.

There is one place and one place only where C puts in the `'\0'` at the end of a character array for you, and that is in the construction

```
"stuff between double quotes"
```

The compiler puts a `'\0'` at the end automatically. Text enclosed in double quotes is called a *string*; its properties are precisely those of an (initialized) array of characters.

for Statement

The for statement is a somewhat generalized while that lets us put the initialization and increment parts of a loop into a single statement along with the test. The general form of the for is

```
for( initialization; expression; increment )
    statement
```

The meaning is exactly

```
initialization;
while( expression ) {
    statement
    increment;
}
```

This slightly more ornate example adds up the elements of an array:

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

In the for statement, the initialization can be left out if you want, but the semicolon has to be there. The increment is also optional. It is *not* followed by a semicolon. The second clause, the test, works the same way as in the `while`: if the expression is true (not zero) do another loop, otherwise get on with the next statement. As with the `while`, the for loop may be done zero times. If the expression is left out, it is taken to be always true, so

```
for( ; ; ) ...
while( 1 ) ...
```

are both infinite loops.

You might ask why we use a `for` since it's so much like a `while`. (You might also ask why we use a `while` because...) The `for` is usually preferable because it keeps the code where it's used and sometimes eliminates the need for compound statements, as in this code that zeros a two-dimensional array:

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

Functions; Comments

Suppose we want, as part of a larger program, to count the occurrences of the ascii characters in some input text. Let us also map illegal characters (those with value >127 or <0) into one pile. Since this is presumably an isolated part of the program, good practice dictates making it a separate function. Here is one way:

```
main( ) {
    int hist[129];      /* 128 legal chars + 1 illegal group */
    ...
    count(hist, 128);   /* count the letters into hist */
    printf( ... );      /* comments look like this; use them */
    ...
    /* anywhere blanks, tabs or newlines could appear */
}

count(buf, size)
    int size, buf[ ]; {
    int i, c;
    for( i=0; i<=size; i++ )
        buf[i] = 0;      /* set buf to zero */
    while( (c=getchar( )) != '\0' ) { /* read til eof */
        if( c > size || c < 0 )
            c = size;     /* fix illegal input */
        buf[c]++;
    }
    return;
}
```

We have already seen many examples of calling a function, so let us concentrate on how to define one. Since `count` has two arguments, we need to declare them, as shown, giving their types, and in the case of `buf`, the fact that it is an array. The declarations of arguments go between the argument list and the opening `{`. There is no need to specify the size of the array `buf`, for it is defined outside of `count`.

The return statement simply says to go back to the calling routine. In fact, we could have omitted it, since a return is implied at the end of a function.

What if we wanted count to return a value, say the number of characters read? The return statement allows for this too:

```
int i, c, nchar;
nchar = 0;
...
while( (c=getchar( )) != '\0' ) {
    if( c > size || c < 0 )
        c = size;
    buf[c]++;
    nchar++;
}
return(nchar);
```

Any expression can appear within the parentheses. Here is a function to compute the minimum of two integers:

```
min(a, b)
int a, b; {
    return( a < b ? a : b );
}
```

To copy a character array, we could write the function

```
strcpy(s1, s2)          /* copies s1 to s2 */
char s1[ ], s2[ ]; {
    int i;
    for( i = 0; (s2[i] = s1[i]) != '\0'; i++ ); }
```

As is often the case, all the work is done by the assignment statement embedded in the test part of the for. Again, the declarations of the arguments s1 and s2 omit the sizes, because they don't matter to strcpy. (In the section on pointers, we will see a more efficient way to do a string copy.)

There is a subtlety in function usage which can trap the unsuspecting Fortran programmer. Simple variables (not arrays) are passed in C by "call by value", which means that the called function is given a copy of its arguments, and doesn't know their addresses. This makes it impossible to change the value of one of the actual input arguments.

There are two ways out of this dilemma. One is to make special arrangements to pass to the function the address of a variable instead of its value. The other is to make the variable

a global or external variable, which is known to each function by its name. We will discuss both possibilities in the next few sections.

Local and External Variables

If we say

```
f( ) {  
    int x;  
    ...  
}  
g( ) {  
    int x;  
    ...  
}
```

each *x* is *local* to its own routine -- the *x* in *f* is unrelated to the *x* in *g*. (Local variables are also called "automatic".) Furthermore each local variable in a routine appears only when the function is called, and *disappears* when the function is exited. Local variables have no memory from one call to the next and must be explicitly initialized upon each entry. (There is a static storage class for making local variables with memory; we won't discuss it.)

As opposed to local variables, external variables are defined external to all functions, and are (potentially) available to all functions. External storage always remains in existence. To make variables external we have to define them external to all functions, and, wherever we want to use them, make a declaration.

```
main( ) {  
    extern int nchar, hist[ ];  
    ...  
    count( );  
    ...  
}  
count( ) {  
    extern int nchar, hist[ ];  
    int i, c;  
    ...  
}  
  
int    hist[129];    /* space for histogram */  
int    nchar;        /* character count */
```

Roughly speaking, any function that wishes to access an external variable must contain an *extern* declaration for it. The declaration is the same as others, except for the added keyword *extern*. Furthermore, there must somewhere be a definition of the external variables external to all functions.

External variables can be initialized; they are set to zero if not explicitly initialized. In its simplest form, initialization is done by putting the value (which must be a constant) after the definition:

Pointers

A pointer in C is the address of something. It is a rare case indeed when we care what the specific address itself is, but pointers are a quite common way to get at the contents of something. The unary operator `&` is used to produce the address of an object, if it has one. Thus

```
int a, b;  
b = &a;
```

puts the address of `a` into `b`. We can't do much with it except print it or pass it to some other routine, because we haven't given `b` the right kind of declaration. But if we declare that `b` is indeed a pointer to an integer, we're in good shape:

```
int a, *b, c;  
b = &a;  
c = *b;
```

`b` contains the address of `a` and `*c = *b` means to use the value in `b` as an address, i.e., as a pointer. The effect is that we get back the contents of `a`, albeit rather indirectly. (It's always the case that `*&x` is the same as `x` if `x` has an address.)

The most frequent use of pointers in C is for walking efficiently along arrays. In fact, in the implementation of an array, the array name represents the address of the zeroth element of the array, so you can't use it on the left side of an expression. (You can't change the address of something by assigning to it.) If we say

```
char *y; char x[100];
```

`y` is of type pointer to character (although it doesn't yet point anywhere). We can make `y` point to an element of `x` by either of

```
y = &x[0];  
y = x;
```

Since `x` is the address of `x[0]` this is legal and consistent.

Now `*y` gives `x[0]`. More importantly,

```
*(y+1) gives x[1]  
*(y+i) gives x[i]
```