C Traps and Pitfalls*

Andrew Koenig

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The C language is like a carving knife: simple, sharp, and extremely useful in skilled hands. Like any sharp tool, C can injure people who don't know how to handle it. This paper shows some of the ways C can injure the unwary, and how to avoid injury.

0. Introduction

The C language and its typical implementations are designed to be used easily by experts. The language is terse and expressive. There are few restrictions to keep the user from blundering. A user who has blundered is often rewarded by an effect that is not obviously related to the cause.

In this paper, we will look at some of these unexpected rewards. Because they are unexpected, it may well be impossible to classify them completely. Nevertheless, we have made a rough effort to do so by looking at what has to happen in order to run a C program. We assume the reader has at least a passing acquaintance with the C language.

Section 1 looks at problems that occur while the program is being broken into tokens. Section 2 follows the program as the compiler groups its tokens into declarations, expressions, and statements. Section 3 recognizes that a C program is often made out of several parts that are compiled separately and bound together. Section 4 deals with misconceptions of meaning: things that happen while the program is actually running. Section 5 examines the relationship between our programs and the library routines they use. In section 6 we note that the program we write is not really the program we run; the preprocessor has gotten at it first. Finally, section 7 discusses portability problems: reasons a program might run on one implementation and not another.

1. Lexical Pitfalls

The first part of a compiler is usually called a *lexical analyzer*. This looks at the sequence of characters that make up the program and breaks them up into *tokens*. A token is a sequence of one or more characters that have a (relatively) uniform meaning in the language being compiled. In C, for instance, the token `->` has a meaning that is quite distinct from that of either of the characters that make it up, and that is independent of the context in which the `->` appears.

For another example, consider the statement:

```
if (x > big) big = x;
```

Each non-blank character in this statement is a separate token, except for the keyword `if` and the two instances of the identifier `big`.

In fact, C programs are broken into tokens twice. First the preprocessor reads the program. It must tokenize the program so that it can find the identifiers, some of which may represent macros. It must then replace each macro invocation by the result of evaluating that macro. Finally, the result of the macro replacement is reassembled into a character stream which is given to the compiler proper. The compiler then breaks the stream into tokens a second time.

* This paper, greatly expanded, is the basis for the book *C Traps and Pitfalls* (Addison-Wesley, 1989, ISBN 0–201–17928–8); interested readers may wish to refer there as well.

In this section, we will explore some common misunderstandings about the meanings of tokens and the relationship between tokens and the characters that make them up. We will talk about the preprocessor later.

1.1. = is not ==

Programming languages derived from Algol, such as Pascal and Ada, use := for assignment and = for comparison. C, on the other hand, uses = for assignment and == for comparison. This is because assignment is more frequent than comparison, so the more common meaning is given to the shorter symbol.

Moreover, C treats assignment as an operator, so that multiple assignments (such as `a=b=c`) can be written easily and assignments can be embedded in larger expressions.

This convenience causes a potential problem: one can inadvertently write an assignment where one intended a comparison. Thus, this statement, which looks like it is checking whether `x` is equal to `y`:

```
if (x = y)
     foo();
```

actually sets `x` to the value of `y` and then checks whether that value is nonzero. Or consider the following loop that is intended to skip blanks, tabs, and newlines in a file:

```
while (c == ' ' || c = '\t' || c == '\n')
     c = getc (f);
```

The programmer mistakenly used = instead of == in the comparison with `'\t'`. This ‘‘comparison’’ actually assigns `'\t'` to `c` and compares the (new) value of `c` to zero. Since `'\t'` is not zero, the ‘‘comparison’’ will always be true, so the loop will eat the entire file. What it does after that depends on whether the particular implementation allows a program to keep reading after it has reached end of file. If it does, the loop will run forever.

Some C compilers try to help the user by giving a warning message for conditions of the form *e1* = *e2*. To avoid warning messages from such compilers, when you want to assign a value to a variable and then check whether the variable is zero, consider making the comparison explicit. In other words, instead of:

```
if (x = y)
     foo();
```

write:

```
if ((x = y) != 0)
     foo();
```

This will also help make your intentions plain.

1.2. & and | are not && or ||

It is easy to miss an inadvertent substitution of = for == because so many other languages use = for comparison. It is also easy to interchange `&` and `&&`, or `|` and `||`, especially because the `&` and `|` operators in C are different from their counterparts in some other languages. We will look at these operators more closely in section 4.

1.3. Multi-character Tokens

Some C tokens, such as `/`, `*`, and `=`, are only one character long. Other C tokens, such as `/*` and `==`, and identifiers, are several characters long. When the C compiler encounters a `/` followed by an `*`, it must be able to decide whether to treat these two characters as two separate tokens or as one single token. The C reference manual tells how to decide: ‘‘If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.’’ Thus, if a `/` is the first character of a token, and the `/` is immediately followed by a `*`, the two characters begin a comment, *regardless* of any other context.

The following statement looks like it sets `y` to the value of `x` divided by the value pointed to by `p`:

```
y = x/*p     /* p points at the divisor */;
```

In fact, `/*` begins a comment, so the compiler will simply gobble up the program text until the `*/` appears. In other words, the statement just sets `y` to the value of `x` and doesn't even look at `p`. Rewriting this statement as

```
y = x / *p     /* p points at the divisor */;
```

or even

```
y = x/(*p)     /* p points at the divisor */;
```

would cause it to do the division the comment suggests.

This sort of near-ambiguity can cause trouble in other contexts. For example, older versions of C use =+ to mean what present versions mean by +=. Such a compiler will treat

```
a=-1;
```

as meaning the same thing as

```
a =- 1;
```

or

```
a = a - 1;
```

This will surprise a programmer who intended

```
a = -1;
```

On the other hand, compilers for these older versions of C would interpret

```
a=/*b;
```

as

```
a =/ * b ;
```

even though the `/*` looks like a comment.

1.4. Exceptions

Compound assignment operators such as += are really multiple tokens. Thus,

```
a + /* strange */ = 1
```

means the same as

```
a += 1
```

These operators are the only cases in which things that look like single tokens are really multiple tokens. In particular,

```
p - > a
```

is illegal. It is *not* a synonym for

```
p -> a
```

As another example, the >> operator is a single token, so >>= is made up of two tokens, not three.

On the other hand, those older compilers that still accept =+ as a synonym for += treat =+ as a single token.

1.5. Strings and Characters

Single and double quotes mean very different things in C, and there are some contexts in which confusing them will result in surprises rather than error messages.

A character enclosed in single quotes is just another way of writing an integer. The integer is the one that corresponds to the given character in the implementation's collating sequence. Thus, in an ASCII implementation, `'a'` means exactly the same thing as 0141 or 97. A string enclosed in double quotes, on the other hand, is a short-hand way of writing a pointer to a nameless array that has been initialized with the characters between the quotes and an extra character whose binary value is zero.

The following two program fragments are equivalent:

```
printf ("Hello world\n");

char hello[] = {'H', 'e', 'l', 'l', 'o', ' ',
     'w', 'o', 'r', 'l', 'd', '\n', 0};
printf (hello);
```

Using a pointer instead of an integer (or vice versa) will often cause a warning message, so using double quotes instead of single quotes (or vice versa) is usually caught. The major exception is in function calls, where most compilers do not check argument types. Thus, saying

```
printf('\n');
```

instead of

```
printf ("\n");
```

will usually result in a surprise at run time.

Because an integer is usually large enough to hold several characters, some C compilers permit multiple characters in a character constant. This means that writing `'yes'` instead of `"yes"` may well go undetected. The latter means "the address of the first of four consecutive memory locations containing `y`, `e`, `s`, and a null character, respectively." The former means "an integer that is composed of the values of the characters `y`, `e`, and `s` in some implementation-defined manner." Any similarity between these two quantities is purely coincidental.

2. Syntactic Pitfalls

To understand a C program, it is not enough to understand the tokens that make it up. One must also understand how the tokens combine to form declarations, expressions, statements, and programs. While these combinations are usually well-defined, the definitions are sometimes counter-intuitive or confusing.

In this section, we look at some syntactic constructions that are less than obvious.

2.1. Understanding Declarations

I once talked to someone who was writing a C program that was going to run stand-alone in a small microprocessor. When this machine was switched on, the hardware would call the subroutine whose address was stored in location 0.

In order to simulate turning power on, we had to devise a C statement that would call this subroutine explicitly. After some thought, we came up with the following:

```
(*(void(*)())0)();
```

Expressions like these strike terror into the hearts of C programmers. They needn't, though, because they can usually be constructed quite easily with the help of a single, simple rule: *declare it the way you use it.*

Every C variable declaration has two parts: a type and a list of stylized expressions that are expected to evaluate to that type. The simplest such expression is a variable:

```
float f, g;
```

indicates that the expressions `f` and `g`, when evaluated, will be of type `float`. Because the thing declared is an expression, parentheses may be used freely:

```
float ((f));
```

means that `((f))` evaluates to a `float` and therefore, by inference, that `f` is also a `float`.

Similar logic applies to function and pointer types. For example,

```
float ff();
```

means that the expression `ff()` is a `float`, and therefore that `ff` is a function that returns a `float`. Analogously,

```
float *pf;
```

means that `*pf` is a `float` and therefore that `pf` is a pointer to a `float`.

These forms combine in declarations the same way they do in expressions. Thus

```
float *g(), (*h)();
```

says that `*g()` and `(*h)()` are `float` expressions. Since `()` binds more tightly than `*`, `*g()` means the same thing as `*(g())`: `g` is a function that returns a pointer to a `float`, and `h` is a pointer to a function that returns a `float`.

Once we know how to declare a variable of a given type, it is easy to write a cast for that type: just remove the variable name and the semicolon from the declaration and enclose the whole thing in parentheses. Thus, since

```
float *g();
```

declares `g` to be a function returning a pointer to a `float`, `(float *())` is a cast to that type.

Armed with this knowledge, we are now prepared to tackle `(*(void(*)())0)()`. We can analyze this statement in two parts. First, suppose that we have a variable `fp` that contains a function pointer and we want to call the function to which `fp` points. That is done this way:

```
(*fp)();
```

If `fp` is a pointer to a function, `*fp` is the function itself, so `(*fp)()` is the way to invoke it. The parentheses in `(*fp)` are essential because the expression would otherwise be interpreted as `*(fp())`. We have now reduced the problem to that of finding an appropriate expression to replace `fp`.

This problem is the second part of our analysis. If C could read our mind about types, we could write:

```
(*0)();
```

This doesn't work because the `*` operator insists on having a pointer as its operand. Furthermore, the operand must be a pointer to a function so that the result of `*` can be called. Thus, we need to cast 0 into a type loosely described as ''pointer to function returning void.''

If `fp` is a pointer to a function returning void, then `(*fp)()` is a void value, and its declaration would look like this:

```
void (*fp)();
```

Thus, we could write:

```
void (*fp)();
(*fp)();
```

at the cost of declaring a dummy variable. But once we know how to declare the variable, we know how to cast a constant to that type: just drop the name from the variable declaration. Thus, we cast 0 to a ''pointer to function returning void'' by saying:

```
        (void(*)())0
```

and we can now replace `fp` by `(void(*)())0`:

```
        (*(void(*)())0)();
```

The semicolon on the end turns the expression into a statement.

At the time we tackled this problem, there was no such thing as a `typedef` declaration. Using it, we could have solved the problem more clearly:

```
        typedef void (*funcptr)();
        (* (funcptr) 0)();
```

2.2. Operators Don't Always Have the Precedence You Want

Suppose that the manifest constant `FLAG` is an integer with exactly one bit turned on in its binary representation (in other words, a power of two), and you want to test whether the integer variable `flags` has that bit turned on. The usual way to write this is:

```
        if (flags & FLAG) ...
```

The meaning of this is plain to most C programmers: an `if` statement tests whether the expression in the parentheses evaluates to 0 or not. It might be nice to make this test more explicit for documentation purposes:

```
        if (flags & FLAG != 0) ...
```

The statement is now easier to understand. It is also wrong, because `!=` binds more tightly than `&`, so the interpretation is now:

```
        if (flags & (FLAG != 0)) ...
```

This will work (by coincidence) if `FLAG` is 1 or 0 (!), but not for any other power of two.*

Suppose you have two integer variables, `h` and `l`, whose values are between 0 and 15 inclusive, and you want to set `r` to an 8-bit value whose low-order bits are those of `l` and whose high-order bits are those of `h`. The natural way to do this is to write:

```
        r = h<<4 + l;
```

Unfortunately, this is wrong. Addition binds more tightly than shifting, so this example is equivalent to

```
        r = h << (4 + l);
```

Here are two ways to get it right:

```
        r = (h << 4) + l;
        r = h << 4 | l;
```

One way to avoid these problems is to parenthesize everything, but expressions with too many parentheses are hard to understand, so it is probably useful to try to remember the precedence levels in C.

Unfortunately, there are fifteen of them, so this is not always easy to do. It can be made easier, though, by classifying them into groups.

The operators that bind the most tightly are the ones that aren't really operators: subscripting, function calls, and structure selection. These all associate to the left.

Next come the unary operators. These have the highest precedence of any of the true operators. Because function calls bind more tightly than unary operators, you must write `(*p)()` to call a function pointed to by `p`; `*p()` implies that `p` is a function that returns a pointer. Casts are unary operators and have the same precedence as any other unary operator. Unary operators are right-associative, so `*p++` is

* Recall that the result of `!=` is always either 1 or 0.

interpreted as `*(p++)` and not as `(*p)++`.

Next come the true binary operators. The arithmetic operators have the highest precedence, then the shift operators, the relational operators, the logical operators, the assignment operators, and finally the conditional operator. The two most important things to keep in mind are:

1. Every logical operator has lower precedence than every relational operator.
2. The shift operators bind more tightly than the relational operators but less tightly than the arithmetic operators.

Within the various operator classes, there are few surprises. Multiplication, division, and remainder have the same precedence, addition and subtraction have the same precedence, and the two shift operators have the same precedence.

One small surprise is that the six relational operators do not all have the same precedence: == and != bind less tightly than the other relational operators. This allows us, for instance, to see if a and b are in the same relative order as c and d by the expression

```
a < b == c < d
```

Within the logical operators, no two have the same precedence. The bitwise operators all bind more tightly than the sequential operators, each *and* operator binds more tightly than the corresponding *or* operator, and the bitwise *exclusive or* operator (^) falls between bitwise *and* and bitwise *or.*

The ternary conditional operator has lower precedence than any we have mentioned so far. This permits the selection expression to contain logical combinations of relational operators, as in

```
z = a < b && b < c ? d : e
```

This example also shows that it makes sense for assignment to have a lower precedence than the conditional operator. Moreover, all the compound assignment operators have the same precedence and they all group right to left, so that

```
a = b = c
```

means the same as

```
b = c; a = b;
```

Lowest of all is the comma operator. This is easy to remember because the comma is often used as a substitute for the semicolon when an expression is required instead of a statement.

Assignment is another operator often involved in precedence mixups. Consider, for example, the following loop intended to copy one file to another:

```
while (c=getc(in) != EOF)
     putc(c,out);
```

The way the expression in the `while` statement is written makes it look like c should be assigned the value of `getc(in)` and then compared with EOF to terminate the loop. Unhappily, assignment has lower precedence than any comparison operator, so the value of c will be the result of comparing `getc(in)`, the value of which is then discarded, and EOF. Thus, the ''copy'' of the file will consist of a stream of bytes whose value is 1.

It is not too hard to see that the example above should be written:

```
while ((c=getc(in)) != EOF)
     putc(c,out);
```

However, errors of this sort can be hard to spot in more complicated expressions. For example, several versions of the *lint* program distributed with the UNIX® system have the following erroneous line:

```
     if( (t=BTYPE(pt1->aty)==STRTY) || t==UNIONTY ){
```

This was intended to assign a value to t and then see if t is equal to STRTY or UNIONTY. The actual effect is quite different.*

The precedence of the C logical operators comes about for historical reasons. B, the predecessor of C, had logical operators that corresponded rougly to C's & and | operators. Although they were defined to act on bits, the compiler would treat them as && and || if they were in a conditional context. When the two usages were split apart in C, it was deemed too dangerous to change the precedence much.**

2.3. Watch Those Semicolons!

An extra semicolon in a C program usually makes little difference: either it is a null statement, which has no effect, or it elicits a diagnostic message from the compiler, which makes it easy to remove. One important exception is after an if or while clause, which must be followed by exactly one statement. Consider this example:

```
if (x[i] > big);
     big = x[i];
```

The semicolon on the first line will not upset the compiler, but this program fragment means something quite different from:

```
if (x[i] > big)
     big = x[i];
```

The first one is equivalent to:

```
if (x[i] > big)   { }
big = x[i];
```

which is, of course, equivalent to:

```
big = x[i];
```

(unless x, i, or big is a macro with side effects).

Another place that a semicolon can make a big difference is at the end of a declaration just before a function definition. Consider the following fragment:

```
struct foo {
     int x;
}

f()
{
     . . .
}
```

There is a semicolon missing between the first } and the f that immediately follows it. The effect of this is to declare that the function f returns a struct foo, which is defined as part of this declaration. If the semicolon were present, f would be defined by default as returning an integer.†

2.4. The Switch Statement

C is unusual in that the cases in its switch statement can flow into each other. Consider, for example, the following program fragments in C and Pascal:

* Thanks to Guy Harris for pointing this out to me.

** Dennis Ritchie and Steve Johnson both pointed this out to me.

† Thanks to an anonymous benefactor for this one.

```
switch (color) {
case 1: printf ("red");
        break;
case 2: printf ("yellow");
        break;
case 3: printf ("blue");
        break;
}

case color of
1:   write ('red');
2:   write ('yellow');
3:   write ('blue')
end
```

Both these program fragments do the same thing: print `red`, `yellow`, or `blue` (without starting a new line), depending on whether the variable `color` is 1, 2, or 3. The program fragments are exactly analogous, with one exception: the Pascal program does not have any part that corresponds to the C `break` statement. The reason for that is that case labels in C behave as true labels: control can flow unimpeded right through a case label.

Looking at it another way, suppose the C fragment looked more like the Pascal fragment:

```
switch (color) {
case 1: printf ("red");
case 2: printf ("yellow");
case 3: printf ("blue");
}
```

and suppose further that `color` were equal to 2. Then, the program would print `yellowblue`, because control would pass naturally from the second `printf` call to the statement after it.

This is both a strength and a weakness of C `switch` statements. It is a weakness because leaving out a `break` statement is easy to do, and often gives rise to obscure program misbehavior. It is a strength because by leaving out a `break` statement deliberately, one can readily express a control structure that is inconvenient to implement otherwise. Specifically, in large `switch` statements, one often finds that the processing for one of the cases reduces to some other case after a relatively small amount of special handling.

For example, consider a program that is an interpreter for some kind of imaginary machine. Such a program might contain a `switch` statement to handle each of the various operation codes. On such a machine, it is often true that a subtract operation is identical to an add operation after the sign of the second operand has been inverted. Thus, it is nice to be able to write something like this:

```
case SUBTRACT:
     opnd2 = -opnd2;
     /* no break */
case ADD:
     . . .
```

As another example, consider the part of a compiler that skips white space while looking for a token. Here, one would want to treat spaces, tabs, and newlines identically except that a newline should cause a line counter to be incremented:

```
        case '\n':
            linecount++;
            /* no break */
        case '\t':
        case ' ':
            . . .
```

2.5. Calling Functions

Unlike some other programming languages, C requires a function call to have an argument list, even if there are no arguments. Thus, if `f` is a function,

```
    f();
```

is a statement that calls the function, but

```
    f;
```

does nothing at all. More precisely, it evaluates the address of the function, but does not call it.*

2.6. The Dangling `else` Problem

We would be remiss in leaving any discussion of syntactic pitfalls without mentioning this one. Although it is not unique to C, it has bitten C programmers with many years of experience.

Consider the following program fragment:

```
    if (x == 0)
        if (y == 0) error();
    else {
        z = x + y;
        f (&z);
    }
```

The programmer's intention for this fragment is that there should be two main cases: $x=0$ and $x \neq 0$. In the first case, the fragment should do nothing at all unless $y=0$, in which case it should call `error`. In the second case, the program should set $z=x+y$ and then call f with the address of z as its argument.

However, the program fragment actually does something quite different. The reason is the rule that an `else` is always associated with the closest unmatched `if`. If we were to indent this fragment the way it is actually executed, it would look like this:

```
    if (x == 0) {
        if (y == 0)
            error();
        else {
            z = x + y;
            f (&z);
        }
    }
```

In other words, nothing at all will happen if $x \neq 0$. To get the effect implied by the indentation of the original example, write:

* Thanks to Richard Stevens for pointing this out.

```
if (x == 0) {
    if (y == 0)
        error();
} else {
    z = x + y;
    f (&z);
}
```

3. Linkage

A C program may consist of several parts that are compiled separately and then bound together by a program usually called a *linker*, *linkage editor*, or *loader*. Because the compiler normally sees only one file at a time, it cannot detect errors whose recognition would require knowledge of several source program files at once.

In this section, we look at some errors of that type. Some C implementations, but not all, have a program called *lint* that catches many of these errors. It is impossible to overemphasize the importance of using such a program if it is available.

3.1. You Must Check External Types Yourself

Suppose you have a C program divided into two files. One file contains the declaration:

```
int n;
```

and the other contains the declaration:

```
long n;
```

This is not a valid C program, because the same external name is declared with two different types in the two files. However, many implementations will not detect this error, because the compiler does not know about the contents of either of the two files while it is compiling the other. Thus, the job of checking type consistency can only be done by the linker (or some utility program like *lint*); if the operating system has a linker that doesn't know about data types, there is little the C compiler can do to force it.

What actually happens when this program is run? There are many possibilities:

1. The implementation is clever enough to detect the type clash. One would then expect to see a diagnostic message explaining that the type of `n` was given differently in two different files.
2. You are using an implementation in which `int` and `long` are really the same type. This is typically true of machines in which 32-bit arithmetic comes most naturally. In this case, your program will probably work as if you had said `long` (or `int`) in both declarations. This would be a good example of a program that works only by coincidence.
3. The two instances of `n` require different amounts of storage, but they happen to share storage in such a way that the values assigned to one are valid for the other. This might happen, for example, if the linker arranged for the `int` to share storage with the low-order part of the `long`. Whether or not this happens is obviously machine- and system-dependent. This is an even better example of a program that works only by coincidence.
4. The two instances of `n` share storage in such a way that assigning a value to one has the effect of apparently assigning a different value to the other. In this case, the program will probably fail.

Another example of this sort of thing happens surprisingly often. One file of a program will contain a declaration like:

```
char filename[] = "/etc/passwd";
```

and another will contain this declaration:

```
char *filename;
```

Although arrays and pointers behave very similarly in some contexts, *they are not the same.* In the first declaration, `filename` is the name of an array of characters. Although using the name will generate a pointer to the first element of that array, that pointer is generated as needed and not actually kept around. In the second declaration, `filename` is the name of a pointer. That pointer points wherever the programmer makes it point. If the programmer doesn't give it a value, it will have a zero (null) value by default.

The two declarations of `filename` use storage in different ways; they cannot coexist.

One way to avoid type clashes of this sort is to use a tool like *lint* if it is available. In order to be able to check for type clashes between separately compiled parts of a program, some program must be able to see all the parts at once. The typical compiler does not do this, but *lint* does.

Another way to avoid these problems is to put external declarations into `include` files. That way, the type of an external object only appears once.*

4. Semantic Pitfalls

A sentence can be perfectly spelled and written with impeccable grammar and still be meaningless. In this section, we will look at ways of writing programs that look like they mean one thing but actually mean something quite different.

We will also discuss contexts in which things that look reasonable on the surface actually give undefined results. We will limit ourselves here to things that are not guaranteed to work on any C implementation. We will leave those that might work on some implementations but not others until section 7, which looks at portability problems.

4.1. Expression Evaluation Sequence

Some C operators always evaluate their operands in a known, specified order. Others don't. Consider, for instance, the following expression:

```
a < b && c < d
```

The language definition states that `a<b` will be evaluated first. If `a` is indeed less than `b`, `c<d` must then be evaluated to determine the value of the whole expression. On the other hand, if `a` is greater than or equal to `b`, then `c<d` is not evaluated at all.

To evaluate `a<b`, on the other hand, the compiler may evaluate either `a` or `b` first. On some machines, it may even evaluate them in parallel.

Only the four C operators `&&`, `||`, `?:`, and `,` specify an order of evaluation. `&&` and `||` evaluate the left operand first, and the right operand only if necessary. The `?:` operator takes three operands: `a?b:c` evaluates `a` first, and then evaluates either `b` or `c`, depending on the value of `a`. The `,` operator evaluates its left operand and discards its value, then evaluates its right operand.†

All other C operators evaluate their operands in undefined order. In particular, the assignment operators do not make any guarantees about evaluation order.

For this reason, the following way of copying the first `n` elements of array `x` to array `y` doesn't work:

```
i = 0;
while (i < n)
     y[i] = x[i++];
```

The trouble is that there is no guarantee that the address of `y[i]` will be evaluated before `i` is incremented.

* Some C compilers insist that there must be exactly one definition of an external object, although there may be many declarations. When using such a compiler, it may be easiest to put a declaration in an `include` file and a definition in some other place. This means that the type of each external object appears twice, but that is better than having it appear more than two times.

† Commas that separate function arguments are not comma operators. For example, `x` and `y` are fetched in undefined order in `f(x,y)`, but not in `g((x,y))`. In the latter example, `g` has one argument. The value of that argument is determined by evaluating `x`, discarding its value, and then evaluating `y`.

On some implementations, it will; on others, it won't. This similar version fails for the same reason:

```
        i = 0;
        while (i < n)
            y[i++] = x[i];
```

On the other hand, this one will work fine:

```
        i = 0;
        while (i < n) {
            y[i] = x[i];
            i++;
        }
```

This can, of course, be abbreviated:

```
        for (i = 0; i < n; i++)
            y[i] = x[i];
```

4.2. The `&&`, `||`, and `!` Operators

C has two classes of logical operators that are occasionally interchangeable: the bitwise operators `&`, `|`, and `~`, and the logical operators `&&`, `||`, and `!`. A programmer who substitutes one of these operators for the corresponding operator from the other class may be in for a surprise: the program may appear to work correctly after such an interchange but may actually be working only by coincidence.

The `&`, `|`, and `~` operators treat their operands as a sequence of bits and work on each bit separately. For example, `10&12` is 8 (`1000`), because `&` looks at the binary representations of 10 (`1010`) and 12 (`1100`) and produces a result that has a bit turned on for each bit that is on in the same position in both operands. Similarly, `10|12` is 14 (`1110`) and `~10` is –11 (`11...110101`), at least on a 2's complement machine.

The `&&`, `||`, and `!` operators, on the other hand, treat their arguments as if they are either "true" or "false," with the convention that 0 represents "false" and any other value represents "true." These operators return 1 for "true" and 0 for "false," and the `&&` and `||` operators do not even evaluate their right-hand operands if their results can be determined from their left-hand operands.

Thus `!10` is zero, because 10 is nonzero, `10&&12` is 1, because both 10 and 12 are nonzero, and `10||12` is also 1, because 10 is nonzero. Moreover, 12 is not even evaluated in the latter expression, nor is `f()` in `10||f()`.

Consider the following program fragment to look for a particular element in a table:

```
        i = 0;
        while (i < tabsize && tab[i] != x)
            i++;
```

The idea behind this loop is that if `i` is equal to `tabsize` when the loop terminates, then the element sought was not found. Otherwise, `i` contains the element's index.

Suppose that the `&&` were inadvertently replaced by `&` in this example. Then the loop would probably still appear to work, but would do so only because of two lucky breaks.

The first is that both comparisons in this example are of a sort that yield 0 if the condition is false and 1 if the condition is true. As long as `x` and `y` are both 1 or 0, `x&y` and `x&&y` will always have the same value. However, if one of the comparisons were to be replaced by one that uses some non-zero value other than 1 to represent "true," then the loop would stop working.

The second lucky break is that looking just one element off the end of an array is usually harmless, provided that the program doesn't change that element. The modified program looks past the end of the array because `&`, unlike `&&`, must always evaluate both of its operands. Thus in the last iteration of the loop, the value of `tab[i]` will be fetched even though `i` is equal to `tabsize`. If `tabsize` is the number of elements in `tab`, this will fetch a non-existent element of `tab`.

4.3. Subscripts Start from Zero

In most languages, an array with n elements normally has those elements numbered with subscripts ranging from 1 to n inclusive. Not so in C.

A C array with n elements does not have an element with a subscript of n, as the elements are numbered from 0 through n-1. Because of this, programmers coming from other languages must be especially careful when using arrays:

```
int i, a[10];
for (i=1; i<=10; i++)
     a[i] = 0;
```

This example, intended to set the elements of a to zero, had an unexpected side effect. Because the comparison in the for statement was i<=10 instead of i<10, the non-existent element number 10 of a was set to zero, thus clobbering the word that followed a in memory. The compiler on which this program was run allocates memory for users' variables in decreasing memory locations, so the word after a turned out to be i. Setting i to zero made the loop into an infinite loop.

4.4. C Doesn't Always Cast Actual Parameters

The following simple program fragment fails for two reasons:

```
double s;
s = sqrt (2);
printf ("%g\n", s);
```

The first reason is that sqrt expects a double value as its argument and it isn't getting one. The second is that it returns a double result but isn't declared as such. One way to correct it is:

```
double s, sqrt();
s = sqrt (2.0);
printf ("%g\n", s);
```

C has two simple rules that control conversion of function arguments: (1) integer values shorter than an int are converted to int; (2) floating-point values shorter than a double are converted to double. All other values are left unconverted. *It is the programmer's responsibility to ensure that the arguments to a function are of the right type.*

Therefore, a programmer who uses a function like sqrt, whose parameter is a double, must be careful to pass arguments that are of float or double type only. The constant 2 is an int and is therefore of the wrong type.

When the value of a function is used in an expression, that value is automatically cast to an appropriate type. However, the compiler must know the actual type returned by the function in order to be able to do this. Functions used without further declaration are assumed to return an int, so declarations for such functions are unnecessary. However, sqrt returns a double, so it must be declared as such before it can be used successfully.

In practice, C implementations generally provide a file that can be brought in with an include statement that contains declarations for library functions like sqrt, but writing declarations is still necessary for programmers who write their own functions – in other words, for anyone who writes non-trivial C programs.

Here is a more spectacular example:

```
main()
{
     int i;
     char c;
     for (i=0; i<5; i++) {
          scanf ("%d", &c);
          printf ("%d ", i);
     }
     printf ("\n");
}
```

Ostensibly, this program reads five numbers from its standard input and writes `0 1 2 3 4` on its standard output. In fact, it doesn't always do that. On one compiler, for example, its output is `0 0 0 0 0 1 2 3 4`.

Why? The key is the declaration of `c` as a `char` rather than as an `int`. When you ask *scanf* to read an integer, it expects a pointer to an integer. What it gets in this case is a pointer to a character. *Scanf* has no way to tell that it didn't get what it expected: it treats its input as an integer pointer and stores an integer there. Since an integer takes up more memory than a character, this steps on some of the memory near `c`.

Exactly what is near `c` is the compiler's business; in this case it turned out to be the low-order part of `i`. Therefore, each time a value was read for `c`, it reset `i` to zero. When the program finally reached end of file, *scanf* stopped trying to put new values into `c`, so `i` could be incremented normally to end the loop.

4.5. Pointers are not Arrays

C programs often adopt the convention that a character string is stored as an array of characters, followed by a null character. Suppose we have two such strings `s` and `t`, and we want to concatenate them into a single string `r`. To do this, we have the usual library functions `strcpy` and `strcat`. The following obvious method doesn't work:

```
char *r;
strcpy (r, s);
strcat (r, t);
```

The reason it doesn't work is that `r` is not initialized to point anywhere. Although `r` is potentially capable of identifying an area of memory, *that area doesn't exist until you allocate it.*

Let's try again, allocating some memory for `r`:

```
char r[100];
strcpy (r, s);
strcat (r, t);
```

This now works as long as the strings pointed to by `s` and `t` aren't too big. Unfortunately, C requires us to state the size of an array as a constant, so there is no way to be certain that `r` will be big enough. However, most C implementations have a library function called `malloc` that takes a number and allocates enough memory for that many characters. There is also usually a function called `strlen` that tells how many characters are in a string. It might seem, therefore, that we could write:

```
char *r, *malloc();
r = malloc (strlen(s) + strlen(t));
strcpy (r, s);
strcat (r, t);
```

This example, however, fails for two reasons. First, `malloc` might run out of memory, an event that it generally signals by quietly returning a null pointer.

Second, and much more important, is that the call to `malloc` doesn't allocate quite enough memory. Recall the convention that a string is terminated by a null character. The `strlen` function returns the number of characters in the argument string, *excluding* the null character at the end. Therefore, if `strlen(s)` is *n*, `s` really requires *n+1* characters to contain it. We must therefore allocate one extra character for `r`. After doing this and checking that `malloc` worked, we get:

```
char *r, *malloc();
r = malloc (strlen(s) + strlen(t) + 1);
if (!r) {
     complain();
     exit (1);
}
strcpy (r, s);
strcat (r, t);
```

4.6. Eschew Synecdoche

A synecdoche (sin-ECK-duh-key) is a literary device, somewhat like a simile or a metaphor, in which, according to the Oxford English Dictionary, ''a more comprehensive term is used for a less comprehensive or vice versa; as whole for part or part for whole, genus for species or species for genus, etc.''

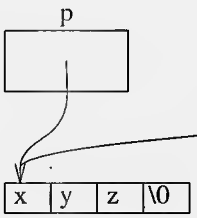
This exactly describes the common C pitfall of confusing a pointer with the data to which it points. This is most common for character strings. For instance:

```
char *p, *q;
p = "xyz";
```

It is important to understand that while it is sometimes useful to think of the value of `p` as the string `xyz` after the assignment, this is not really true. Instead, the value of `p` is a *pointer* to the 0th element of an array of four characters, whose values are `'x'`, `'y'`, `'z'`, and `'\0'`. Thus, if we now execute

```
q = p;
```

`p` and `q` are now two pointers to the same part of memory. The characters in that memory did not get copied by the assignment. The situation now looks like this:



The thing to remember is that *copying a pointer does not copy the thing it points to.*

Thus, if after this we were to execute

```
q[1] = 'Y';
```

`q` would point to memory containing the string `xYz`. So would `p`, because `p` and `q` point to the same memory.

4.7. The Null Pointer is Not the Null String

The result of converting an integer to a pointer is implementation-dependent, with one important exception. That exception is the constant `0`, which is guaranteed to be converted to a pointer that is unequal to any valid pointer. For documentation, this value is often given symbolically:

```
#define NULL 0
```

but the effect is the same. The important thing to remember about `0` when used as a pointer is that *it must never be dereferenced.* In other words, when you have assigned `0` to a pointer variable, you must not ask what is in the memory it points to. It is valid to write:

```
        if (p == (char *) 0) ...
```

but it is not valid to write:

```
        if (strcmp (p, (char *) 0) == 0) ...
```

because strcmp always looks at the memory addressed by its arguments.

If p is a null pointer, it is not even valid to say:

```
        printf (p);
```

or

```
        printf ("%s", p);
```

4.8. Integer Overflow

The C language definition is very specific about what happens when an integer operation overflows or underflows.

If either operand is unsigned, the result is unsigned, and is defined to be modulo 2^n, where n is the word size. If both operands are signed, the result is *undefined*.

Suppose, for example, that a and b are two integer variables, known to be non-negative, and you want to test whether a+b might overflow. One obvious way to do it looks something like this:

```
        if (a + b < 0)
            complain();
```

In general, this does not work.

The point is that once a+b has overflowed, all bets are off as to what the result will be. For example, on some machines, an addition operation sets an internal register to one of four states: positive, negative, zero, or overflow. On such a machine, the compiler would have every right to implement the example given above by adding a and b and checking whether this internal register was in negative state afterwards. If the operation overflowed, the register would be in overflow state, and the test would fail.

One correct way of doing this particular test relies on the fact that unsigned arithmetic is well-defined for all values, as are the conversions between signed and unsigned values:

```
        if ((int) ((unsigned) a + (unsigned) b) < 0)
            complain();
```

4.9. Shift Operators

Two questions seem to cause trouble for people who use shift operators:

1. In a right shift, are vacated bits filled with zeroes or copies of the sign bit?
2. What values are permitted for the shift count?

The answer to the first question is simple but sometimes implementation-dependent. If the item being shifted is unsigned, zeroes are shifted in. If the item is signed, the implementation is permitted to fill vacated bit positions either with zeroes or with copies of the sign bit. If you care about vacated bits in a right shift, declare the variable in question as unsigned. You are then entitled to assume that vacated bits will be set to zero.

The answer to the second question is also simple: if the item being shifted is n bits long, then the shift count must be greater than or equal to zero and *strictly* less than n. Thus, it is not possible to shift all the bits out of a value in a single operation.

For example, if an int is 32 bits, and n is an int, it is legal to write n<<31 and n<<0 but not n<<32 or n<<-1.

Note that a right shift of a signed integer is generally not equivalent to division by a power of two, even if the implementation copies the sign into vacated bits. To prove this, consider that the value of (-1)>>1 cannot possibly be zero.

5. Library Functions

Every useful C program must use library functions, because there is no way of doing input or output built into the language. In this section, we look at some cases where some widely-available library functions behave in ways that the programmer might not expect.

5.1. Getc Returns an Integer

Consider the following program:

```
#include <stdio.h>

main()
{
        char c;

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

This program looks like it should copy its standard input to its standard output. In fact, it doesn't quite do this.

The reason is that c is declared as a character rather than as an integer. This means that it is impossible for c to hold every possible character as well as EOF.

Thus there are two possibilities. Either some legitimate input character will cause c to take on the same value as EOF, or it will be impossible for c to have the value EOF at all. In the former case, the program will stop copying in the middle of certain files. In the latter case, the program will go into an infinite loop.

Actually, there is a third case: the program may work by coincidence. The C Reference Manual defines the result of the expression

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

quite rigorously. Section 6.1 states:

> When a longer integer is converted to a shorter or to a char, it is truncated on the left; excess bits are simply discarded.

Section 7.14 states:

> There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.

The combined effect of these two sections is to require that the result of *getchar* be truncated to a character value by discarding the high-order bits, and that this truncated value then be compared with EOF. As part of this comparison, the value of c must be extended to an integer, either by padding on the left with zero bits or by sign extension, as appropriate.

However, some compilers do not implement this expression correctly. They properly assign the low-order bits of the value of *getchar* to c. However, instead of then comparing c to EOF, they compare the entire value of *getchar*! A compiler that does this will make the sample program shown above appear to work ''correctly.''

5.2. Buffered Output and Memory Allocation

When a program produces output, how important is it that a human be able to see that output immediately? It depends on the program.

For example, if the output is going to a terminal and is asking the person sitting at that terminal to answer a question, it is crucial that the person see the output in order to be able to know what to type. On the other hand, if the output is going to a file, from where it will eventually be sent to a line printer, it is only important that all the output get there eventually.

It is often more expensive to arrange for output to appear immediately than it is to save it up for a while and write it later on in a large chunk. For this reason, C implementations typically afford the programmer some control over how much output is to be produced before it is actually written.

That control is often vested in a library function called *setbuf.* If `buf` is a character array of appropriate size, then

```
setbuf (stdout, buf);
```

tells the I/O library that all output written to *stdout* should henceforth use *buf* as an output buffer, and that output directed to *stdout* should not actually be written until *buf* becomes full or until the programmer directs it to be written by calling *fflush.* The appropriate size for such a buffer is defined as `BUFSIZ` in <*stdio.h*>.

Thus, the following program illustrates the obvious way to use *setbuf* in a program that copies its standard input to its standard output:

```
#include <stdio.h>

main()
{
     int c;

     char buf[BUFSIZ];
     setbuf (stdout, buf);

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

Unfortunately, this program is wrong, for a subtle reason.

To see where the trouble lies, ask when the buffer is flushed for the last time. Answer: after the main program has finished, as part of the cleaning up that the library does before handing control back to the operating system. But by that time, the buffer has already been freed!

There are two ways to prevent this sort of trouble.

First, make the buffer static, either by declaring it explicitly as static:

```
static char buf[BUFSIZ];
```

or by moving the declaration outside the main program entirely.

Another possibility is to allocate the buffer dynamically and never free it:

```
char *malloc();
setbuf (stdout, malloc (BUFSIZ));
```

Note that in this latter case, it is unnecessary to check if *malloc* was successful, because if it fails it will return a null pointer. A null pointer is an acceptable second argument to *setbuf*; it requests that *stdout* be unbuffered. This will work slowly, but it will work.

6. The Preprocessor

The programs we run are not the programs we write: they are first transformed by the C preprocessor. The preprocessor gives us a way of abbreviating things that is important for two major reasons (and several minor ones).

First, we may want to be able to change all instances of a particular quantity, such as the size of a table, by changing one number and recompiling the program.*

Second, we may want to define things that appear to be functions but do not have the execution overhead normally associated with a function call. For example, *getchar* and *putchar* are usually implemented as macros to avoid having to call a function for each character of input or output.

6.1. Macros are not Functions

Because macros can be made to appear almost as if they were functions, programmers are sometimes tempted to regard them as truly equivalent. Thus, one sees things like this:

```
#define max(a,b)  ((a)>(b)?(a):(b))
```

Notice all the parentheses in the macro body. They defend against the possibility that a or b might be expressions that contain operators of lower precedence than >.

The main problem, though, with defining things like max as macros is that an operand that is used twice may be evaluated twice. Thus, in this example, if a is greater than b, a will be evaluated twice: once during the comparison, and again to calculate the value yielded by max.

Not only can this be inefficient, it can also be wrong:

```
biggest = x[0];
i = 1;
while (i < n)
     biggest = max (biggest, x[i++]);
```

This would work fine if max were a true function, but fails with max a macro. Suppose, for example, that x[0] is 2, x[1] is 3, and x[2] is 1. Look at what happens during the first iteration of the loop. The assignment statement expands into:

```
biggest = ((biggest)>(x[i++])?(biggest):(x[i++]));
```

First, biggest is compared to x[i++]. Since i is 1 and x[1] is 3, the relation is false. As a side effect, i is incremented to 2.

Because the relation is false, the value of x[i++] is now assigned to biggest. However, i is now 2, so the value assigned to biggest is the value of x[2], which is 1.

One way around these worries is to ensure that the arguments to the max macro do not have any side effects:

```
biggest = x[0];
for (i = 1; i < n; i++)
     biggest = max (biggest, x[i]);
```

Here is another example of the hazards of mixing side effects and macros. This is the definition of the putc macro from <stdio.h> in the Eighth Edition of the Unix system:

```
#define putc(x,p)  (--(p)->_cnt>=0?(*(p)->_ptr++=(x)):_flsbuf(x,p))
```

The first argument to putc is a character to be written to a file; the second argument is a pointer to an internal data structure that describes the file. Notice that the first argument, which could easily be something like *z++, is carefully evaluated only once, even though it appears in two separate places in the macro body, while the second argument is evaluated twice (in the macro body, x appears twice, but since

* The preprocessor also makes it easy to group such *manifest constants* together to make them easier to find.

the two occurrences are on opposite sides of a `:` operator, exactly one of them will be evaluated in any single instance of `putc`). Since it is unusual for the file argument to *putc* to have side effects, this rarely causes trouble. Nevertheless, it is documented in the user's manual: ''Because it is implemented as a macro, *putc* treats a *stream* argument with side effects improperly. In particular, `putc(c,*f++)` doesn't work sensibly.'' Notice that `putc(*c++,f)` works fine in this implementation.

Some C implementations are less careful. For instance, not everyone handles `putc(*c++,f)` correctly. As another example, consider the *toupper* function that appears in many C libraries. It translates a lower-case letter to the corresponding upper-case letter while leaving other characters unchanged. If we assume that all the lower-case letters and all the upper-case letters are contiguous (with a possible gap between the cases), we get the following function:

```
toupper(c)
{
     if (c >= 'a' && c <= 'z')
          c += 'A' - 'a';
     return c;
}
```

In most C implementations, the subroutine call overhead is much longer than the actual calculations, so the implementor is tempted to make it a macro:

```
#define toupper(c) ((c)>='a' && (c)<='z'? (c)+('A'-'a'): (c))
```

This is indeed faster than the function in many cases. However, it will cause a surprise for anyone who tries to use `toupper(*p++)`.

Another thing to watch out for when using macros is that they may generate very large expressions indeed. For example, look again at the definition of `max`:

```
#define max(a,b) ((a)>(b)?(a):(b))
```

Suppose we want to use this definition to find the largest of `a`, `b`, `c`, and `d`. If we write the obvious:

```
max(a,max(b,max(c,d)))
```

this expands to:

```
((a)>(((b)>(((c)>(d)?(c):(d)))?(b):(((c)>(d)?(c):(d)))))?
 (a):(((b)>(((c)>(d)?(c):(d)))?(b):(((c)>(d)?(c):(d)))))
```

which is surprisingly large. We can make it a little less large by balancing the operands:

```
max(max(a,b),max(c,d))
```

which gives:

```
((((a)>(b)?(a):(b)))>(((c)>(d)?(c):(d)))?
 (((a)>(b)?(a):(b))):(((c)>(d)?(c):(d))))
```

Somehow, though, it seems easier to write:

```
biggest = a;
if (biggest < b) biggest = b;
if (biggest < c) biggest = c;
if (biggest < d) biggest = d;
```

6.2. Macros are not Type Definitions

One common use of macros is to permit several things in diverse places to be the same type:

```
#define FOOTYPE struct foo
FOOTYPE a;
FOOTYPE b, c;
```

This lets the programmer change the types of a, b, and c just by changing one line of the program, even if a, b, and c are declared in widely different places.

Using a macro definition for this has the advantage of portability – any C compiler supports it. Most C compilers also support another way of doing this:

```
typedef struct foo FOOTYPE;
```

This defines FOOTYPE as a new type that is equivalent to struct foo.

These two ways of naming a type may appear to be equivalent, but the typedef is more flexible. Consider, for example, the following:

```
#define T1 struct foo *
typedef struct foo *T2;
```

These definitions make T1 and T2 conceptually equivalent to a pointer to a struct foo. But look what happens when we try to use them with more than one variable:

```
T1 a, b;
T2 c, d;
```

The first declaration gets expanded to

```
struct foo * a, b;
```

This defines a to be a pointer to a structure, but defines b to be a structure (not a pointer). The second declaration, in contrast, defines both c and d as pointers to structures, because T2 behaves as a true type.

7. Portability Pitfalls

C has been implemented by many people to run on many machines. Indeed, one of the reasons to write programs in C in the first place is that it is easy to move them from one programming environment to another.

However, because there are so many implementors, they do not all talk to each other. Moreover, different systems have different requirements, so it is reasonable to expect C implementations to differ slightly between one machine and another.

Because so many of the early C implementations were associated with the UNIX operating system, the nature of many of these functions was shaped by that system. When people started implementing C under other systems, they tried to make the library behave in ways that would be familiar to programmers used to the UNIX system.

They did not always succeed. What is more, as more people in different parts of the world started working on different versions of the UNIX system, the exact nature of some of the library functions inevitably diverged. Today, a C programmer who wishes to write programs useful in someone else's environment must know about many of these subtle differences.

7.1. What's in a Name?

Some C compilers treat all the characters of an identifier as being significant. Others ignore characters past some limit when storing identifiers. C compilers usually produce object programs that must then be processed by loaders in order to be able to access library subroutines. Loaders, in turn, often impose their own restrictions on the kinds of names they can handle.

One common loader restriction is that letters in external names must be in upper case only. When faced with such a restriction, it is reasonable for a C implementor to force all external names to upper case. Restrictions of this sort are blessed by section 2.1 the C reference manual:

An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as as a letter. Upper and lower case letters are different. No more than the first eight characters are significant, although more may be used. External identifiers, which are used by various assemblers and loaders, are more restricted:

Here, the reference manual goes on to give examples of various implementations that restrict external identifiers to a single case, or to fewer than eight characters, or both.

Because of all this, it is important to be careful when choosing identifiers in programs intended to be portable. Having two subroutines named, say `print_fields` and `print_float` would not be a very good idea.

As a striking example, consider the following function:

```
char *
Malloc (n)
     unsigned n;
{
     char *p, *malloc();
     p = malloc (n);
     if (p == NULL)
          panic ("out of memory");
     return p;
}
```

This function is a simple way of ensuring that running out of memory will not go undetected. The idea is for the program to allocate memory by calling `Malloc` instead of `malloc`. If `malloc` ever fails, the result will be to call `panic` which will presumably terminate the program with an appropriate error message.

Consider, however, what happens when this function is used on a system that ignores case distinctions in external identifiers. In effect, the names `malloc` and `Malloc` become equivalent. In other words, the library function `malloc` is effectively replaced by the `Malloc` function above, which when it calls `malloc` is really calling itself. The result, of course, is that the first attempt to allocate memory results in a recursion loop and consequent mayhem, even though the function will work on an implementation that preserves case distinctions.

7.2. How Big is an Integer?

C provides the programmer with three sizes of integers: ordinary, short, and long, and with characters, which behave as if they were small integers. The language definition does not guarantee much about the relative sizes of the various kinds of integer:

1. The four sizes of integers are non-decreasing.
2. An ordinary integer is large enough to contain any array subscript.
3. The size of a character is natural for the particular hardware.

Most modern machines have 8-bit characters, though a few have 7- or 9-bit characters, so characters are usually 7, 8, or 9 bits.

Long integers are usually at least 32 bits long, so that a long integer can be used to represent the size of a file.

Ordinary integers are usually at least 16 bits long, because shorter integers would impose too much of a restriction on the maximum size of an array.

Short integers are almost always exactly 16 bits long.

What does this all mean in practice? The most important thing is that one cannot count on having any particular precision available. Informally, one can probably expect 16 bits for a short or an ordinary integer, and 32 bits for a long integer, but not even those sizes are guaranteed. One can certainly use ordinary integers to express table sizes and subscripts, but what about a variable that must be able to hold values up to ten million?

The most portable way to do that is probably to define a ''new'' type:

```
typedef long tenmil;
```

Now one can use this type to declare a variable of that width and know that, at worst, one will have to change a single type definition to get all those variables to be the right type.

7.3. Are Characters Signed or Unsigned?

Most modern computers support 8-bit characters, so most modern C compilers implement characters as 8-bit integers. However, not all compilers interpret those 8-bit quantities the same way.

The issue becomes important only when converting a `char` quantity to a larger integer. Going the other way, the results are well-defined: excess bits are simply discarded. But a compiler converting a `char` to an `int` has a choice: should it treat the `char` as a signed or an unsigned quantity? If the former, it should expand the `char` to an `int` by replicating the sign bit; if the latter, it should fill the extra bit positions with zeroes.

The results of this decision are important to virtually anyone who deals with characters with their high-order bits turned on. It determines whether 8-bit characters are going to be considered to range from –128 through 127 or from 0 through 255. This, in turn, affects the way the programmer will design things like hash tables and translate tables.

If you care whether a character value with the high-order bit on is treated as a negative number, you should probably declare it as `unsigned char`. Such values are guaranteed to be zero-extended when converted to integer, whereas ordinary `char` variables may be signed in one implementation and unsigned in another.

Incidentally, it is a common misconception that if `c` is a character variable, one can obtain the unsigned integer equivalent of `c` by writing `(unsigned) c`. This fails because a `char` quantity is converted to `int` before any operator is applied to it, *even a cast.* Thus `c` is converted first to a signed integer and then to an unsigned integer, with possibly unexpected results.

The right way to do it is `(unsigned char) c`.

7.4. Are Right Shifts Signed or Unsigned?

This bears repeating: a program that cares how shifts are done had better declare the quantities being shifted as unsigned.

7.5. How Does Division Truncate?

Suppose we divide `a` by `b` to give a quotient `q` and remainder `r`:

```
q = a / b;
r = a % b;
```

For the moment, suppose also that `b>0`.

What relationships might we want to hold between `a`, `b`, `p`, and `q`?

1. Most important, we want `q*b + r == a`, because this is the relation that defines the remainder.
2. If we change the sign of `a`, we want that to change the sign of `q`, but not the absolute value.
3. We want to ensure that `r>=0` and `r<b`. For instance, if the remainder is being used as an index to a hash table, it is important to be able to know that it will always be a valid index.

These three properties are clearly desirable for integer division and remainder operations. Unfortunately, *they cannot all be true at once.*

Consider 3/2, giving a quotient of 1 and a remainder of 1. This satisfies property 1. What should be the value of $-3/2$? Property 2 suggests that it should be -1, but if that is so, the remainder must *also* be -1, which violates property 3. Alternatively, we can satisfy property 3 by making the remainder 1, in which case property 1 demands that the quotient be -2. This violates property 2.

Thus C, and any language that implements truncating integer division, must give up at least one of these three principles.

Most programming languages give up number 3, saying instead that the remainder has the same sign as the dividend. This makes it possible to preserve properties 1 and 2. Most C implementations do this in practice, also.

However, the C language definition only guarantees property 1, along with the property that $|r|<|b|$ and that $r \geq 0$ whenever $a \geq 0$ and $b > 0$. This property is less restrictive than either property 2 or property 3, and actually permits some rather strange implementations that would be unlikely to occur in practice (such as an implementation that always truncates the quotient *away* from zero).

Despite its sometimes unwanted flexibility, the C definition is enough that we can usually make integer division do what we want, provided that we know what we want. Suppose, for example, that we have a number n that represents some function of the characters in an identifier, and we want to use division to obtain a hash table entry h such that $0 \leq h < HASHSIZE$. If we know that n is never negative, we simply write

```
h = n % HASHSIZE;
```

However, if n might be negative, this is not good enough, because h might also be negative. However, we know that $h > -HASHSIZE$, so we can write:

```
h = n % HASHSIZE;
if (h < 0)
     h += HASHSIZE;
```

Better yet, declare n as unsigned.

7.6. How Big is a Random Number?

This size ambiguity has affected library design as well. When the only C implementation ran on the PDP-11‡ computer, there was a function called *rand* that returned a (pseudo-) random non-negative integer. PDP-11 integers were 16 bits long, including the sign, so *rand* would return an integer between 0 and $2^{15}-1$.

When C was implemented on the VAX-11, integers were 32 bits long. What was the range of the *rand* function on the VAX-11?

For their system, the people at the University of California took the view that *rand* should return a value that ranges over all possible non-negative integers, so their version of *rand* returns an integer between 0 and $2^{31}-1$.

The people at AT&T, on the other hand, decided that a PDP-11 program that expected the result of *rand* to be less than 2^{15} would be easier to transport to a VAX-11 if the *rand* function returned a value between 0 and 2^{15} there, too.

As a result, it is now difficult to write a program that uses *rand* without tailoring it to the implementation.

7.7. Case Conversion

The *toupper* and *tolower* functions have a similar history. They were originally written as macros:

```
#define toupper(c) ((c)+'A'-'a')
#define tolower(c) ((c)+'a'-'A')
```

When given a lower-case letter as input *toupper* yields the corresponding upper-case letter. *Tolower* does the opposite. Both these macros depend on the implementation's character set to the extent that they demand that the difference between an upper-case letter and the corresponding lower-case letter be the same constant for all letters. This assumption is valid for both the ASCII and EBCDIC character sets, and probably isn't too dangerous, because the non-portability of these macro definitions can be encapsulated in

‡ PDP-11 and VAX-11 are Trademarks of Digital Equipment Corporation.

the single file that contains them.

These macros do have one disadvantage, though: when given something that is not a letter of the appropriate case, they return garbage. Thus, the following innocent program fragment to convert a file to lower case doesn’t work with these macros:

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

Instead, one must write:

```
int c;
while ((c = getchar()) != EOF)
     putchar (isupper (c)? tolower (c): c);
```

At one point, some enterprising soul in the UNIX development organization at AT&T noticed that most uses of *toupper* and *tolower* were preceded by tests to ensure that their arguments were appropriate. He considered rewriting the macros this way:

```
#define toupper(c) ((c) >= ’a’ && (c) <= ’z’? (c) + ’A’ - ’a’: (c))
#define tolower(c) ((c) >= ’A’ && (c) <= ’Z’? (c) + ’a’ - ’A’: (c))
```

but realized that this would cause c to be evaluated anywhere between one and three times for each call, which would play havoc with expressions like toupper(*p++). Instead, he decided to rewrite *toupper* and *tolower* as functions. *Toupper* now looked something like this:

```
int toupper (c)
     int c;
{
     if (c >= ’a’ && c <= ’z’)
          return c + ’A’ - ’a’;
     return c;
}
```

and *tolower* looked similar.

This change had the advantage of robustness, at the cost of introducing function call overhead into each use of these functions. Our hero realized that some people might not be willing to pay the cost of this overhead, so he re-introduced the macros with new names:

```
#define _toupper(c) ((c)+’A’-’a’)
#define _tolower(c) ((c)+’a’-’A’)
```

This gave users a choice of convenience or speed.

There was just one problem in all this: the people at Berkeley never followed suit, nor did some other C implementors. This means that a program written on an AT&T system that uses *toupper* or *tolower*, and assumes that it will be able to pass an argument that is not a letter of the appropriate case, may stop working on some other C implementation.

This sort of failure is very hard to trace for someone who does not know this bit of history.

7.8. Free First, then Reallocate

Most C implementations provide users with three memory allocation functions called *malloc*, *realloc*, and *free*. Calling malloc(n) returns a pointer to *n* characters of newly-allocated memory that the programmer can use. Giving *free* a pointer to memory previously returned by *malloc* makes that memory available for re-use. Calling *realloc* with a pointer to an allocated area and a new size stretches or shrinks the memory to the new size, possibly copying it in the process.

Or so one might think. The truth is actually somewhat more subtle. Here is an excerpt from the description of *realloc* that appears in the System V Interface Definition:

> *Realloc* changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

The Seventh Edition of the reference manual for the UNIX system contains a copy of the same paragraph. In addition, it contains a second paragraph describing *realloc*:

> *Realloc* also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*; thus sequences of *free*, *malloc* and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

Thus, the following is legal under the Seventh Edition:

```
free (p);
p = realloc (p, newsize);
```

This idiosyncrasy remains in systems derived from the Seventh Edition: it is possible to free a storage area and then reallocate it. By implication, freeing memory on these systems is guaranteed not to change its contents until the next time memory is allocated. Thus, on these systems, one can free all the elements of a list by the following curious means:

```
for (p = head; p != NULL; p = p->next)
     free ((char *) p);
```

without worrying that the call to *free* might invalidate `p->next`.

Needless to say, this technique is not recommended, if only because not all C implementations preserve memory long enough after it has been freed. However, the Seventh Edition manual leaves one thing unstated: the original implementation of *realloc* actually required that the area given to it for reallocation be free first. For this reason, there are many C programs floating around that free memory first and then reallocate it, and this is something to watch out for when moving a C program to another implementation.

7.9. An Example of Portability Problems

Let's take a look at a problem that has been solved many times by many people. The following program takes two arguments: a long integer and a (pointer to a) function. It converts the integer to decimal and calls the given function with each character of the decimal representation.

```
void
printnum (n, p)
     long n;
     void (*p)();
{
     if (n < 0) {
          (*p) ('-');
          n = -n;
     }
     if (n >= 10)
          printnum (n/10, p);
     (*p) (n % 10 + '0');
}
```

This program is fairly straightforward. First we check if `n` is negative; if so, we print a sign and make `n` positive. Next, we test if $n \geq 10$. If so, its decimal representation has two or more digits, so we call *printnum* recursively to print all but the last digit. Finally, we print the last digit.

This program, for all its simplicity, has several portability problems. The first is the method it uses to convert the low-order decimal digit of `n` to character form. Using `n%10` to get the value of the low-order digit is fine, but adding `'0'` to it to get the corresponding character representation is not. This addition assumes that the machine collating sequence has all the digits in sequence with no gaps, so that `'0'+5` has the same value as `'5'`, and so on. This assumption, while true of the ASCII and EBCDIC character sets, might not be true for some machines. The way to avoid that problem is to use a table:

```
void
printnum (n, p)
        long n;
        void (*p)();
{
        if (n < 0) {
                (*p) ('-');
                n = -n;
        }
        if (n >= 10)
                printnum (n/10, p);
        (*p) ("0123456789"[n % 10]);
}
```

The next problem involves what happens if $n < 0$. The program prints a negative sign and sets `n` to `-n`. This assignment might overflow, because 2's complement machines generally allow more negative values than positive values to be represented. In particular, if a (long) integer is k bits plus one extra bit for the sign, -2^k can be represented but 2^k cannot.

There are several ways around this problem. The most obvious one is to assign `n` to an `unsigned long` value and be done with it. However, some C compilers do not implement `unsigned long`, so let us see how we can get along without it.

In both 1's complement and 2's complement machines, changing the sign of a *positive* integer is guaranteed not to overflow. The only trouble comes when changing the sign of a *negative* value. Therefore, we can avoid trouble by making sure we do not attempt to make `n` positive.

Of course, once we have printed the sign of a negative value, we would like to be able to treat negative and positive numbers the same way. The way to do that is to force `n` to be negative after printing the sign, and to do all our arithmetic with negative values. If we do this, we will have to ensure that the part of the program that prints the sign is executed only once; the easiest way to do that is to split the program into two functions:

```
void
printnum (n, p)
        long n;
        void (*p)();
{
        void printneg();
        if (n < 0) {
                (*p) ('-');
                printneg (n, p);
        } else
                printneg (-n, p);
}

void
printneg (n, p)
        long n;
        void (*p)();
{
        if (n <= -10)
                printneg (n/10, p);
        (*p) ("0123456789"[-(n % 10)]);
}
```

Printnum now just checks if the number being printed is negative; if so it prints a negative sign. In either case, it calls *printneg* with the negative absolute value of *n*. We have also modified the body of *printneg* to cater to the fact that *n* will always be a negative number or zero.

Or have we? We have used `n/10` and `n%10` to represent the leading digits and the trailing digit of *n* (with suitable sign changes). Recall that integer division behaves in a somewhat implementation-dependent way when one of the operands is negative. For that reason, it might actually be that `n%10` is positive! In that case, `-(n%10)` would be negative, and we would run off the end of our digit array.

We cater to this problem by creating two temporary variables to hold the quotient and remainder. After we do the division, we check that the remainder is in range and adjust both variables if not. *Printnum* has not changed, so we show only *printneg*:

```
void
printneg (n, p)
     long n;
     void (*p)();
{
     long q;
     int r;

     q = n / 10;
     r = n % 10;
     if (r > 0) {
          r -= 10;
          q++;
     }
     if (n <= -10)
          printneg (q, p);
     (*p) ("0123456789"[-r]);
}
```

8. This Space Available

There are many ways for C programmers to go astray that have not been mentioned in this paper. If you find one, please contact the author. It may well be included, with an acknowledging footnote, in a future revision.

References

The C Programming Language (Kernighan and Ritchie, Prentice-Hall 1978) is the definitive work on C. It contains both an excellent tutorial, aimed at people who are already familiar with other high-level languages, and a reference manual that describes the entire language succinctly. While the language has expanded slightly since 1978, this book is still the last word on most subjects. This book also contains the ''C Reference Manual'' we have mentioned several times in this paper.

The C Puzzle Book (Feuer, Prentice-Hall, 1982) is an unusual way to hone one's syntactic skills. The book is a collection of puzzles (and answers) whose solutions test the reader's knowledge of C's fine points.

C: A Reference Manual (Harbison and Steele, Prentice Hall 1984) is mostly intended as a reference source for implementors. Other users may also find it useful, particularly because of its meticulous cross references.