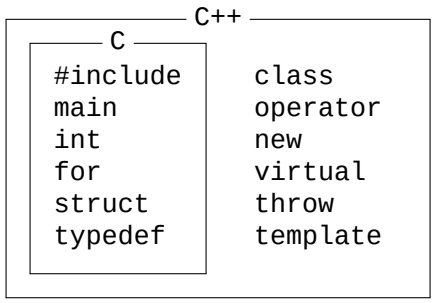
1

Before Objects

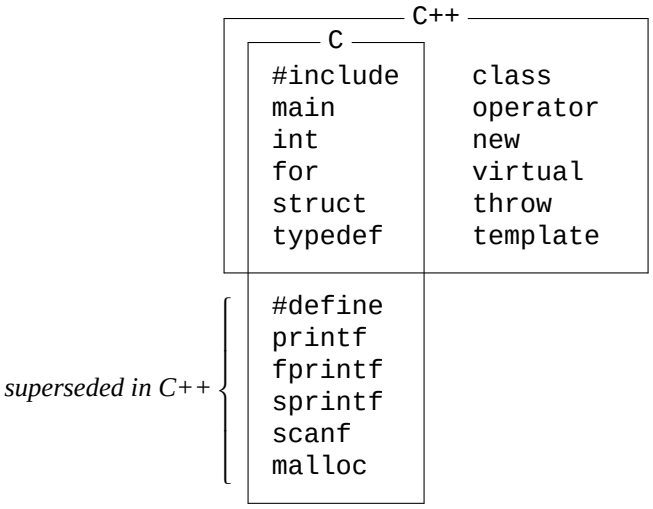
1.1 Introduction: C++ as a superset of C

> *Es ist das schönste Los einer physikalischen Theorie, wenn sie selbst zur Aufstellung einer umfassenden Theorie den Weg weist, in welcher sie als Grenzfall weiterlebt.*
>
> The fairest destiny of any physical theory is to point the way to the introduction of a more comprehensive theory, in which it lives on as a limiting case.
>
> —Albert Einstein, *Relativity: the Special and the General Theory* (1918), Chapter 22

C++ is a superset of C: it has everything that C has, plus more. Conversely, C is a subset of C++, in which it lives on as a limiting case. Here are a few representative words from each language, showing C nestled snugly within C++.



But it's not really this simple. C++ has better alternatives for many of the characteristic features of C, including `#define`, `malloc` and `free`, and the input and output functions in the C Standard Library. A better diagram would exclude these old war horses from C++.



I know that the reader is eager to learn about objects. But before we do, Chapter 1 will have to present the new machinery for input and output. After all, if we can't do output, we can't even tell if our program is running. This chapter will also round up all the other pre-object topics of C++. Chapter 2 will introduce objects and they will occupy the rest of the book.

In C, i/o is performed by passing arguments to a function.

```
1     printf("%d %d %d\n", i, j, k);                    /* C example */
```

In C++, i/o is performed by giving operands to an operator.

```
2     cout << i << " " << j << " " << k << "\n";        //C++ example
```

In C, the operators were used only for arithmetic. In C++ they will have many more applications, including i/o and formatting, dynamic memory allocation, and data structure access. In fact, all of the glamorous, high-profile features of C++ will ultimately be written in terms of operators. We therefore begin with a review of operators, operands, and expressions.

1.2 Expression Evaluation

The rules for evaluating an expression are the same for C and C++. The gaps where there are no rules are also the same.

Operators and their arity

An *operator* is a symbol that performs an operation.

`+`	*add*
`-`	*subtract*
`*`	*multiply*
`/`	*divide*

The values that are added, subtracted, or otherwise operated upon, are called *operands.* The number of operands is called the operator's *arity,* a term derived from the words unary, binary, ternary. Here are a few examples.

The addition operator takes two operands; we therefore say it is a *binary* operator.

```
a + b
```

The ‘‘negation operator’’ (negative sign) takes one operand; it is a *unary* operator.

```
-a
```

The ‘‘conditional operator’’ takes three operands; it is the only *ternary* operator.

```
a ? b : c
```

The value of the above expression is `b` if `a` is true, `c` if `a` is false.

The ‘‘throw’’ operator can take zero arguments or one argument.

```
1     throw               //no arguments
2     throw x             //one argument
```

The pair of parentheses in the expression `f()` are also an operator, like the pair of plusses in the expression `a++`. This ‘‘function call’’ operator can enclose any number of arguments, so it can have any arity. It can equally well be unary, binary, ternary, or worse.

`f()`	*unary*
`f(a)`	*binary*
`f(a, b)`	*ternary*
`f(a, b, c)`	*quaternary*
`f(a, b, c, d)`	*quinary*

The increment and decrement operators can be written before their operand or after it. They are the only operators that can be either *prefix* or *postfix.*

`++a`	*increment can be prefix or postfix*
`a++`	
`--a`	*decrement can be prefix or postfix*
`a--`	

Expressions and subexpressions

A *literal* is a number, character, or string. Examples are

`10`	*a literal of type* `int`
`3.14`	*a literal of type* `double`
`'A'`	*a literal of type* `char`
`"hello"`	*a literal of type* `array of 6 const char` *(including the terminating* `'\0'`*)*

The smallest *expressions* are the individual literals and variables of the program. A larger expression, such as

```
a + b
```

is built by pasting together smaller ones with operators. We say that `a` and `b` are *subexpressions* of the `a + b`, since they are little expressions in their own right.

Every expression in C and C++ has a value, except for those of data type `void`. When computing this value, or *evaluating* the expression, the operators are executed one at a time. A problem will therefore arise when an expression has more than operator. Which one goes first?

In some cases the answer is simple. In the expression

```
a * -b
```

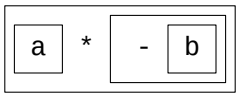
the multiplication and the negation are competing for the operand `b`. The negation wins, because it has elbowed its way closer to the `b`. The `b` is therefore an operand of the negation, the `-b` is a subexpression of the `a * -b`, and the negation is executed first.

Let’s annotate our expression `a * -b` to show how it is *parsed* into subexpressions. We will draw a box around each expression and subexpression.

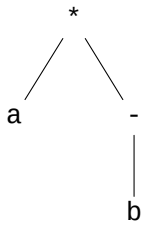
When one box contains another, the expression in the inner box will be evaluated before the one in the outer box. The innermost boxes hold the individual variables, so each variable will be evaluated before any other expression that contains it. In our example, the `b` is evaluated before the `-b`, which is evaluated before the `a * -b`.

When neither of two boxes contains the other, the expressions in the boxes can be evaluated in either order (exceptions on pp. 13−14). For example, neither of the boxes for `a` and `b` contains the other, so we cannot predict which will be evaluated first. But these expressions are merely variables. Their evaluation consists only of fetching their values from memory, so it does not matter which one is first.

Of the four boxes in the diagram, the one around the `-b` shows that the `-` executes before the `*`. The other four boxes are redundant, drawn only for completeness. We say that the `*` is the *outermost* operator of the expression, since it is enclosed by only the outermost box. The outermost operator is always executed last.



Incidentally, other textbooks use a tree diagram to show how an expression is parsed. In that notation, an operator at a lower node (branching point) is executed before its parent (the node immediately above it).



Some operators surround one of their operands, preventing any other operator from being adjacent to the operand. The subscripting operator, for example, consists of two separate ‘‘tokens’’ (p. 100). It is a binary operator that surrounds its second operand.

```
a[b]
```

Another operator that engulfs an operand is `static_cast<>()`. In C, we wrote the old-fashioned cast operator to form an expression whose value is that of an operand converted to another data type.

```
(int)a
```

In C++, the new notation for this is

```
static_cast<int>(a)
```

The parentheses are part of the `static_cast` operator, so it surrounds its operand.

▼ Homework 1.2a: draw the boxes

Each of the following expressions has two operators competing for a disputed operand. In each case, one of the operators has elbowed its way closer to the operand. Draw the boxes showing the articulation of each expression into subexpressions. Don’t worry yet about what the operators mean.

```
a++ * b
a * ++b
a :: ::b
a[b] + c
a[b]++
a() + b
a()++
*++a
-static_cast<int>(a)
a + static_cast<int>(b)
static_cast<int>(a) + b
static_cast<int>(static_cast<double>(a))
```

▲

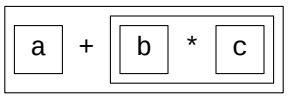
Operator precedence

Here are addition and multiplication competing for the `b`. This time, both operators are adjacent to the disputed operand. To evaluate the expression correctly, we must know which one will get to sink its teeth into the `b`.

```
a + b * c
```

Since they are both adjacent to the same disputed operand, we can use *operator precedence* to determine the outcome. Each operator has a level of precedence, listed in the following table. The `*` has a higher level of precedence than the `+` (level 13 vs. level 12), so the `b` is an operand of the `*`. Think of the `*` as having a greater gravitational pull or chemical valence than the `+`. The `*` is executed first.

The pictures show that the `b * c` is a subexpression of `a + b * c`. The hapless `a + b` is not a subexpression at all, just as the Delmarva Peninsula is not a state and Russia is not a continent. Of the five boxes in the diagram, the one around the `b * c` shows us that the `*` is executed before the `+`. The other four are drawn only for completeness.



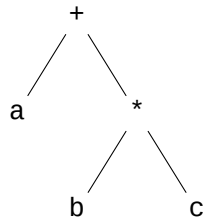


Table of operators

The operators are listed with the highest level of precedence at the top. The operators within each level share the same precedence. For example, the `[]` and `++` in level 16 have equal precedence. For ‘‘lvalues’’ and ‘‘rvalues’’, see pp. 12−13.

The `T` stands for the name of a data type, e.g., `int`.

The 60 Operators of C++

prec.	*associativity*	*arity*	*value*	*operator*	*description*
18	none	unary	lvalue	`::`	global scope operator
17	left-to-right	binary	lvalue	`::`	class or namespace scope
16	left-to-right	binary	lvalue	`[]`	subscripting
		binary	lvalue	`. ->`	access a field of a structure
		any	rvalue	`()`	function call operator
		unary	rvalue	`++ --`	postfix increment, decrement
		unary	rvalue	`static_cast<T>()`	safe cast
		unary	rvalue	`reinterpret_cast<T>()`	unsafe or unportable cast
		unary	rvalue	`const_cast<T *>()`	remove the read-only'ness
		unary	rvalue	`dynamic_cast<T *>()`	runtime type identification
		unary	rvalue	`typeid(T` *or expression*`)`	runtime type identification
15	right-to-left	unary	lvalue	`++ --`	prefix increment, decrement
			lvalue	`*`	dereference a pointer
			rvalue	`&`	address of
			rvalue	`+ -`	positive, negative (2's comp.)
			rvalue	`~`	bitwise not (1's complement)
			rvalue	`!`	not
			rvalue	`sizeof`	size in bytes
			rvalue	`new`	dynamic memory allocation
			none	`delete`	and deallocation (scalar)
			none	`delete[]`	and deallocation (array)
			rvalue	`(T)`	old-style cast (type conversion)
14	left-to-right	binary	lvalue	`.* ->*`	dereference pointer to member
13	left-to-right	binary	rvalue	`* / %`	multiply, divide, remainder
12	left-to-right	binary	rvalue	`+ -`	add, subtract
11	left-to-right	binary	rvalue	`<< >>`	left and right shift
10	left-to-right	binary	rvalue	`< <= > >=`	compare
9	left-to-right	binary	rvalue	`== !=`	equality, inequality compare
8	left-to-right	binary	rvalue	`&`	bitwise and
7	left-to-right	binary	rvalue	`^`	bitwise exclusive or
6	left-to-right	binary	rvalue	`|`	bitwise or
5	left-to-right	binary	rvalue	`&&`	and
4	left-to-right	binary	rvalue	`||`	or
3	right-to-left	ternary	lvalue	`?:`	conditional operator
		binary	lvalue	`=`	assignment
		binary	lvalue	`*= /= %= += -=`	} assign back to same variable
		binary	lvalue	`<<= >>= &= ^= |=`	
2	none	unary	none	`throw`	throw an exception
1	left-to-right	binary	lvalue	`,`	comma operator (sequencing)

As in C, the same symbol can represents two different operators. There are four groups of examples.

(1) The `++` and `--` in level 15 are the prefix increment and decrement; the ones in level 16 are the postfix ones.

(2) The `-` in level 15 is the negation operator (the negative sign) because it is unary; the one in level 12 is the subtraction operator (the minus sign) because it is binary. Similarly, the `+` in level 15 is the ''positive sign'' because it is unary; the `+` in level 12 is the ''plus sign'' (the addition operator) because it is binary. The positive sign does nothing, merely yielding the value of its operand. `+10` is the same as a plain old `10`.

(3) The unary `*` in level 15 is the dereferencing operator; the binary `*` in level 13 is the multiplication operator. Similarly, the unary `&` in level 15 is the ‘‘address of’’ operator; the binary `&` in level 8 is the ‘‘bitwise and’’ operator.

(4) The `::`’s in levels 18 and 17 are two different ‘‘scope’’ operators, unary and binary. These are new in C++.

A digression on unfamiliar operators

The ‘‘non-bitwise’’ operators `!`, `&&`, and `||` know only the two values of data type `bool`. They treat any non-zero operand as `true`, and any zero operand as `false`. For example, `1 || 2` yields the value `true`.

The ‘‘bitwise’’ operators `~`, `&`, `^`, and `|` know about multi-bit operands and results. For example, `1 | 2` yields the value `3`. To see this, write the operands one above the other in binary, draw a horizontal line, and write the answer underneath. For ‘‘bitwise or’’, each bit of the answer will be `0` if all the bits above it were `0`, and `1` otherwise.

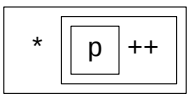
```
  00000000000000000000000000000001    one
| 00000000000000000000000000000010    two
  00000000000000000000000000000011    three
```

Precedence of unary operators

Returning to operator precedence, the same rules apply when two unary operators are adjacent to a disputed operand. The expression

```
*p++
```

has two operators competing for the `p`: the postfix increment operator `++` at level 16, and the dereferencing operator `*` at level 15. Due to its higher precedence, the `++` wins. The `p` is an operand of the `++`, the `p++` is a subexpression of the `*p++`, and the `++` is executed first.



So because the `++` has higher precedence, the computer will execute the `++` before the `*`. But because the `++` is postfix, the computer should perform the dereference before the increment. To reconcile these requirements, we must analyze the evaluation of the subexpression `p++` into three steps.

(1) The postfix `++` creates an invisible, nameless variable called an *anonymous temporary,* and copies the value of `p` into it.

(2) The postfix `++` adds 1 to `p`.

(3) The anonymous temporary is used as the value of the expression `p++`. (This value becomes the operand of the next operator, the `*`, which dereferences the value.)

The `*p++` therefore does behave as if it dereferenced the original value of `p` and then incremented `p`. What it actually does, however, is to increment `p` and then dereference a copy of the original value of `p`. The net effect is the same. (The three steps will become explicit in lines 50−54 of `date.h` on p. 274.)

Establishing that the `++` executes before the `*`, even though the `++` is postfix, is no empty exercise in metaphysics. When we do ‘‘operator overloading’’, we will see that each operator in an expression may actually call a function (sneak preview, p. 18; full-blown example, pp. 291−292). Our `*p++` might call two functions, with the bizarre names `operator++` and `operator*`. `operator++` will be called first because the `++` operator is executed first. Thus, the rules of precedence can dictate the order in which our functions are called. Without knowing this order, any attempt at debugging would be hopeless.

▼ **Homework 1.2b: operator precedence**

The operators in the following expression are binary. Draw a box around each subexpression, but don't bother with the box around each individual variable. If there is no room to draw the boxes, just number the operators in order of execution. Don't worry about the meaning of the operators.

```
1     a = b && c ^ d == e << f * g -> h :: i ->* j + k < l & m | n || o , p
```

The operators in the following expression are unary. Draw the boxes.

```
2     -- :: a ()
```

The following expressions have unary and binary operators. Draw the boxes.

```
3     -a - b
4     -a -> b
5     ::a -> b
6     ::a :: b
7     ::a :: b++
8     cout << a + b
9     cin >> a[i]
```

▲

Parentheses

Consider the expression

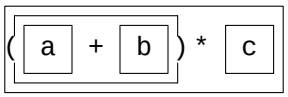
```
a + b * c
```

The `b` is an operand of the `*`, the `b * c` is a subexpression, and the multiplication executes before the addition. The hapless `a + b` is not an expression at all: it is bits and pieces of several expressions.

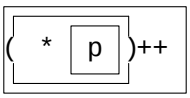
To cause the `b` to be an operand of the `+`, we surround the `a + b` with parentheses. The `a + b` is now a subexpression of `(a + b) * c`. As a consequence of the fact that the `b` is an operand of the `+`, the addition now executes before the multiplication.

```
(a + b) * c
```

These parentheses are not operators; they merely change the way the expression is parsed into subexpressions. We therefore embed them in the walls of the box they create.



Similarly, in the expression `*p++` we can force the `*` to execute before the `++`.



The `p` is now an operand of the `*`, and the `*p` is a subexpression of `(*p)++`. We dereference `p` and then increment the resulting value.

Let's use the term *binding parentheses* for these parentheses that override which operator a given expression is an operand of. We can easily distinguish between the binding parentheses and the function call operator. The function call operator always is immediately preceded by an expression; the pair of binding parentheses never is. In the last example, the left pair of parentheses are the function call operator because `a` is an expression; the right pair of parentheses are also function call operator because `a()` is an expression.

1 `*a(b)` *function call operator because* `a` *is an expression*
2 `a*(b)` *binding parentheses because neither* `*` *nor* `a*` *is an expression*
3 `a++(b)` *function call operator because* `a++` *is an expression*
4 `a+(b)` *binding parentheses because neither* `+` *nor* `a+` *is an expression*
5 `a()()`

Note that the binding parentheses have no effect other than to override which operator an exression is an operand of. For example, the following parentheses have no effect at all, since even without them the `a` and `b` would still be operands of the left `*`. In particular, they have no effect on which multiplication is executed first. See pp. 14−16.

```
(a * b) + c * d
```

▼ Homework 1.2c: operator precedence and parentheses

Kernighan and Ritchie once remarked that in C, ‘‘[s]ome of the operators have the wrong precedence’’.* Which ones did they have in mind? What would go wrong here without the order-of-opertions parentheses? Note that the outermost parentheses belong to the `if` statement, not the expression.

```
1     if ((a & b) == c) {
```

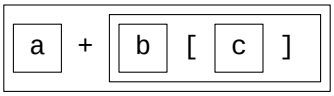
The following examples call functions in the C Standard Library (except `term_key`). In what order do the three operators of each expression execute? What would go wrong without the order-of-operation parentheses?

```
2     while ((c = getchar()) != EOF) {
3     if ((p = malloc(n)) == NULL) {
4     if ((fp = fopen("filename", "w")) == NULL) {
5     while ((c = term_key()) != '\0') {
```

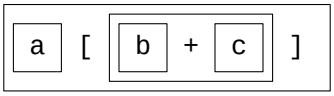
▲

Operators that act as parentheses

Here are two operators adjacent to, and competing for, the `b`. The `[]` has higher precedence, so it wins. The `b` is an operand of the `[]`, and the `b[c]` is a subexpression.



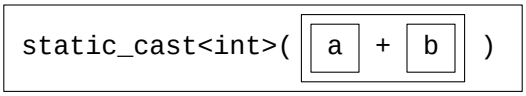
Here are the same two operators adjacent to, and competing for, the same `b`. We would expect the `[]` to win again. But an operator that surrounds an operand acts as a pair of order-of-operations parentheses, forcing the enclosed material to be a subexpression. This time, the `b` is an operand of the `+`, and the `b+c` is a subexpression. This is true even though the `[]` and `+` are both adjacent to the `b` and the `[]` has higher precedence.



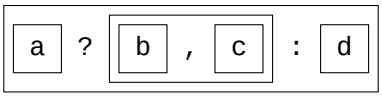
Another operator that acts like parentheses is the `static_cast<>()` operator. It surrounds its operand, since the parentheses are part of the operator. The `a+b` is a subexpression even though the `static_cast<>()` and the `+` are both adjacent to the `a` and the `static_cast<>()` has higher precedence.

---

* *The C Programming Language, 2nd ed.,* p. 3.



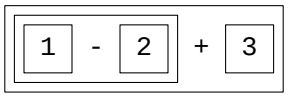
Another example is the conditional operator `?:`. Like the `[]`, it surrounds its second operand. The `b , c` is a subexpression true even though the `?:` and the `,` are both adjacent to the `b` and the `?:` has higher precedence.



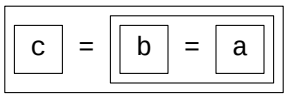
Operator associativity

Operator precedence cannot help us when the two adjacent operators have equal precedence. In this case, we resort to *operator associativity.* Like precedence, it can be used only when the two operators are adjacent to a disputed operand. Here are four examples.

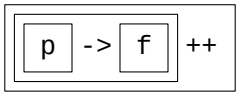
(1) The expression `1 - 2 + 3` has two operators of equal precedence competing for the `2`. Since their associativity is left-to-right, the `2` is an operand of the `-`, the `1 - 2` is a subexpression, and the `-` is executed before the `+`. The value of the whole expression is 2, not −4.



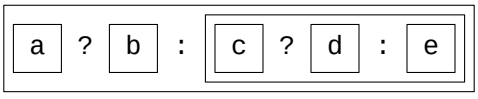
(2) The expression `c = b = a` has two assignment operators competing for the `b`. They have equal precedence since they are the same operator. This time the associativity is right-to-left, so the `=` on the right wins. We assign the `a` to `b` and then the `b` to `c`.



(3) The expression `p->f++` has a unary and a binary operator competing for the `f`. They have equal precedence since the `++` is postfix. Their associativity is left-to-right, so the `->` wins.



(4) The expression `a ? b : c ? d : e` has two ternary operators competing for the `c`. They have equal precedence since they are the same operator. Their associativity is right-to-left, so the one on the right wins.



Here are examples of adjacent ternaries. The pointer `p` is initialized to the address of one of three possible strings.

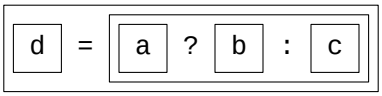
```
1     const char *p =
2         i < j ? "less than" :
3         i > j ? "greater than" :
```

```
 4                "equal";

 5     //ordinal suffix for number in range 1 to 10 inclusive
 6     const char *suffix =
 7         n == 1 ? "st" :
 8         n == 2 ? "nd" :
 9         n == 3 ? "rd" :
10                  "th";
```

See p. 776 for another example.

Connoisseurs of grammar will notice that one of the C operators has a new precedence in C++. The conditional operator `?:` used to have higher precedence than the assignment operator `=`, but they are now at the same level. We hasten to assure the reader that every legal expression in C will still be parsed (boxed) the same way in C++, although the reasons may sometimes be different. The simplest example is the following, which executes the `?:` before the `=` in both languages. In C, this was because the `?:` had higher precedence. In C++, it is because they have the same precedence and right-to-left associativity. In both cases, we get the same parse.



▼ Homework 1.2d: operator associativity

Draw the boxes for the following expressions. Don't worry about their meanings.

The first example requires both precedence and associativity. Precedence tells us that the `a` is an operand of the left `+`, not of the `=`. Associativity tells us that the `b` is an operand of the left `+`, not of the right `+`.

```
 1 d = a + b + c
 2 cout << a << b << c
 3 cout << a << b << c + d
 4 cout << a << b << (c & d)
 5 cin  >> a >> b >> c

 6 a[i][j]
 7 a[i].f

 8     c.d(e)
 9   b(c).d(e)
10 a.b(c).d(e)
```

▲

▼ Homework 1.2e: operator associativity

Integer division yields an integer result. So won't the expression `5 / 9` be zero, giving us a Celsius temperature of zero? Is this a bug or "merely" bad style?

```
 1     double fahrenheit = 72;
 2     double celsius = (fahrenheit - 32) * 5 / 9;
```

▲

Other uses of precedence and associativity

The multiplication operator * needs a level of precedence and a direction of associativity because it can compete with other operators for an adjacent operand.

```
a + b * c
```

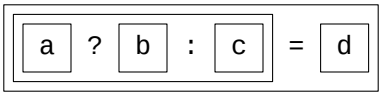
But why did they bother to assign a precedence and associativity to `static_cast<>()`? The cast surrounds its operand, so no other operator can compete with it.

```
a + static_cast<int>(b) * c
```

Well, precedence and associativity do more than just determine how an expression is parsed. They also can be used to disallow certain illegal combinations of operators. A C example would be the following.

```
a ? b : c = d
```

In that language, the `?:` had a higher precedence than `=`, leading us to expect the following parse.

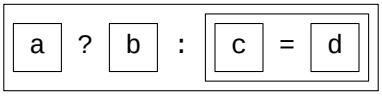


does not compile in C

But the grammar of C stipulates that the left operand of = cannot be an expression whose outermost operator (p. 4) has a precedence lower than that of prefix ++. This disqualifies the expression `a?b:c` from being the left operand of =, so the ‘‘expression’’ `a?b:c=d` will not compile in C.

Lvalues and rvalues

The expression does compile in C++, with the following parse. The operators = and `?:` have equal precedence and right-to-left associativity.



The expression

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

also compiles in C++. Let’s introduce the terminology for talking about why it compiles in C++ but not in C.

An *lvalue* is an expression that can be used as the left operand of the assignment operator =. An *rvalue* is an expression that can be used as the right operand of the assignment operator. A variable for example, can be an lvalue or an rvalue; a literal can be only an rvalue.

```
1     a = 10              //variable used as lvalue, literal used as rvalue
2     a = b               //variables used as lvalue and rvalue
```

More precisely, an lvalue is an expression whose address can be taken and whose value can be changed by = or ++ and -- (prefix or postfix), or whose value *could* be changed were it not of a `const` data type.

A literal is not an lvalue. None of the following will compile.

```
3     10 = a
4     ++20
5     --30
6     &40
```

Several operators build an expression that can be either an lvalue or an rvalue. The operator `?:` always yields an rvalue in C, but can yield an lvalue or an rvalue in C++.

```
 7     a[i] = 10                    x = a[i]
 8     s.f = 20                     x = s.f
 9     p->f = 30                    x = p->f
10     *q = 40                      x = *q
11     (a ? b : c) = x              x = a ? b : c
```

But most operators yield only an rvalue. The parentheses are needed to attempt to apply the `++`, `--`, and `&` to the entire expression `a + b`.

```
12     c = a + b           //a + b can be an rvalue

13     a + b = 10          //a + b cannot be an lvalue: won't compile
14     ++(a + b)
15     --(a + b)
16     &(a + b)
```

With heroic effort, involving ‘‘operator overloading’’ and ‘‘references’’, almost any operator in C++ can be forced to yield an lvalue. The table on p. 5 identifies the ones that can do so without any extraordinary machinery. The other operators yield only rvalues.

Lvalues and rvalues will reappear when we do operator overloading on p. 284.

Exceptional operators that evaluate their left operand first

Four operators always evaluate their left operand before the other(s).

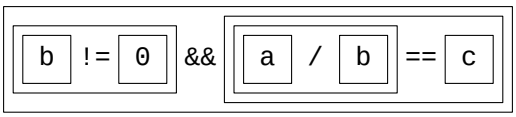
```
a && b
a || b
a ? b : c
a , b
```

The `&&` evaluates its left operand first, and then evaluates its right one only if the left one was true. The `||` also evaluates its left operand first, but then evaluates its right one only if the left one was *false.* The `?:` evaluates its left operand first, and then evaluates one of its other two operands: the second if the first is true, the third if the first is false. The comma operator evaluates its left operand first, and then evaluates its right operand. Our only example will be on pp. 263−264.

The rules for `&&` and `||` gives us a shortcut or checking if it safe to perform a dangerous operation. A division by zero, for example, will result in *Qundefined* behavior, a polite way of saying that the program may crash. The following expression will perform the division only if the divisor `b` is non-zero. If `b` is zero, the division will be skipped and the `if` will be false.

```
1     if (b != 0 && a / b == c) {
2         //arrive here if a / b == c
```

Unfortunately, our box notation has no way to show that the left operand of `&&` is evaluated first, and the right operand possibly not evaluated at all. We will just have to remember it.



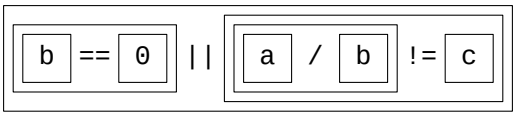
In a language offering no guarantee that the left operand of `&&` is evaluated first, we would have to evaluate the `b != 0` and the `a / b == c` in two separate statements.

```
3     if (b != 0) {
4         if (a / b == c) {
5             //arrive here if a / b == c
```

See p. 64 for another example.

Using `||`, we can perform the opposite test. Once again, we perform the division only if the divisor `b` is non-zero. If `b` is zero, the division will be skipped and the `if` will be true.

```
6       if (b == 0 || a / b != c) {
7           //arrive here if a / b != c
```



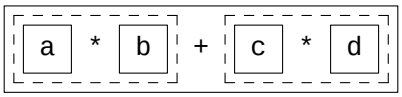
In a language offering no guarantee that the left operand of `||` is evaluated first, we would have to evaluate the `b==0` and the `a/b!=c` in two separate statements. We would need a `bool` variable as well as another `if`.

```
8       bool x = b == 0;                //true if b == 0
9
10      if (!x) {
11          //arrive here if b != 0
12          x = a / b != c;             //x stays true if a / b != c
13      }
14
15      if (x) {
16          //arrive here if a / b != c
```

Ambiguity

Armed with precedence and associativity, any expression can be parsed (boxed) in only one way. This would lead us to believe that there is only one order in which the subexpressions will be evaluated, i.e., only one order in which the operators will be executed. Surprisingly, this is not the case. The following examples have pairs of boxes, drawn with dashes, that do not contain each other. We mentioned on p. 4 that when this happens, the subexpressions in the boxes can be evaluated in either order (except for the four operators in the previous section, `&& || ?: ,`).

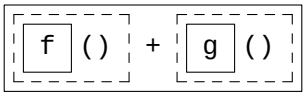
(1) The following addition definitely goes last because it has lower precedence than the adjacent multiplications. But there is no way to tell which multiplication goes first. We cannot use operator precedence or associativity, because the two multiplications are not adjacent: they are not competing for the same operand. Their order of execution could be different on each platform.



C and C++ permit this freedom (anarchy) because of their lust for speed. Some machines are faster when they perform the left multiplication first, others the right one first. C and C++ permit each machine to pick the order that is best for it. Java, on the other hand, is executed on only one machine, the ‘‘Java Virtual Machine’’. It always follows the same order, the left dashed box first.

Admittedly, it made no difference which of the above multiplications was evaluated first. That is because the multiplications had no *side effects:* they did not change the value of a variable or perform I/O. But they might have side effects in the future. When we do operator overloading, the `*` and `+` operators might call functions named `operator*` and `operator+`. If these functions have side effects, the order in which they are called could make a big difference. If they produce output, for example, the order in which they are called will be visible.

(2) In this example the dashed boxes contain expressions that definitely call functions. Once again, the addition goes last because it has the lowest precedence. But precedence or associativity cannot tell us which function call goes first. The order could be different on each platform.



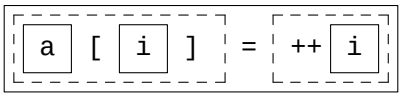
To ensure that `f` is called before `g` on all platforms, we would have to split the expression into two separate statements.

```
1     const int temp = f();
2     temp + g();
```

(3) Assume that we have declared an array with 12 elements. A variable that holds an array subscript should be of data type `size_t`; see p. 66.

```
3     int a[12];
4     size_t i = 10;
```

The `=` in the following expression will go last because it has the lowest precedence. But we cannot tell whether the `[]` or the `++` will go first. On platforms where the subexpression `a[i]` is evaluated before the `++i`, the assignment will put 11 into `a[10]`. On platforms where the `++i` is evaluated before the `a[i]`, the assignment will put 11 into `a[11]`. In either case, `i` would be left with the value 11.



Changing the `++` to postfix would not remove the ambiguity. On platforms where the `a[i]` is evaluated before the `i++`, we would just be putting 10 into `a[10]`. On platforms where the `i++` is evaluated before the `a[i]`, we would put 10 into `a[11]`. In either case, `i` would once again be left with the value 11.

Once again, the solution is to split the expression into two statements.

```
5     ++i;
6     a[i] = i;                       //Put 11 into a[11] on all platforms.
```

or

```
7     const int temp = i;
8     a[temp] = ++i;                  //Put 11 into a[10] on all platforms.
```

I'm not encouraging you to write ambiguous expressions. I want you to recognize them and stay away from them. For other examples, see pp. 393, 688.

▼ Homework 1.2f: unpredictable output

(1) Why can't we predict the following output? What are the two possibilities in each case? The moral is that we should never increment or decrement a variable whose value is used elsewhere in the same expression.

```
9     int i = 10;
10    cout << ++i << " " << i << "\n";
11
12    int j = 10;
13    cout << ++j << " " << ++j << "\n";
```

(2) Why can't we predict the following output? What are the two possibilities in each case?

```
1     int i = 10;
2     cout << ++i + i << "\n";
3
```

```
4     int j = 10;
5     cout << j + ++j << "\n";
```

6 (3) Why can't we predict which array element is the left operand of the assignment? What are the
7 two possibilities, and what is the new value assigned to the array element?

```
8     int i = 8;
9     int j = 9;
10    int a[10];
11
12    a[i = j] = i;
```

▲

1.3 Output and Input

> "[I]t is not enough to discharge a projectile and then take no further notice of it. We must follow it throughout its course, until the moment it hits its target."
>
> "What?" shouted the general and the major, a bit taken aback by this idea.
>
> "Absolutely," replied Barbicane with self-assurance. "Absolutely. Otherwise our experiment would produce no result."
>
> —Jules Verne, *From the Earth to the Moon* (1865), Chapter 7

A C program to be translated into C++

Every program in this book is on the web. See if you can download the one below. It accepts one command line argument and echoes it to the standard output.

Many platforms rely on the filename suffix to determine what language the program is written in. Our convention will be to end the name of a C program with `.c` (dot lowercase c) and a C++ program with `.C` (dot uppercase C). If your platform has different conventions, you will have to rename the downloaded files before you compile them.

The line numbers, and the blank after each line number, are not part of the source code.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/prog/prog.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char **argv)   /* char *argv[] would mean the same thing */
5 {
6     if (argc != 2) {
7         fprintf(stderr, "%s: requires 1 command line argument\n", argv[0]);
8         return EXIT_FAILURE;
9     }
10
11    printf("I received the argument %s.\n", argv[1]);
12    return EXIT_SUCCESS;
13 }
```

The variable `argc` in the above line 6 gives the number of words constituting the command line that launched the program. The first word is the name of the program itself; the remaining ones are the *command line arguments.* The value of `argc` is therefore one more than the number of command line arguments. If the program is supposed to receive one command line argument, `argc` should be 2.

The `printf` in the above line 11 is an abbreviation for

```
14      fprintf(stdout, "I received the argument %s.\n", argv[1]);
```

This `stdout` and the `stderr` in the above line 7 are two destinations for output, called *file pointers* in C. `stdout` stands for "standard output"; `stderr` stands for "standard error output".

The words `printf`, `fprintf`, and `stderr` are part of the C Standard Library, not the C language itself. We must therefore include the header file `stdio.h` in which they are declared. The GNU C compiler on our Unix machine i5.nyu.edu is named `gcc`. For this compiler, the `stdio.h` file is in the directory `/usr/include`. I discovered this by giving the `-H` option to `gcc` when compiling the program.

On some platforms (Windows), you might have to insert the following function call immediately before line 12 to prevent the program's output from disappearing before you can read it.

```
1       system("PAUSE");
```

If you have to make this call in many places, consider doing it automatically with the `atexit` function in the C Standard Library.

The integer returned by the `main` function in line 12 tells the operating system (Windows, Macintosh, Unix) whether the program succeeded or failed at its primary mission. The code number for success is represented by the macro `EXIT_SUCCESS`. This macro belongs to the C Standard Library, and is defined in the header file `stdlib.h`. If the program fails, we return the value `EXIT_FAILURE` in line 8.

To remove the leading whitespace from each line so you can insert your own, Unix people can use the following "global substitute" command in `vi`:

```
:g/^[   ]*/s///                    one blank and one tab inside the square brackets
```

To see the online manual for `gcc`, give the following command. The `1$` is the Unix shell prompt.

```
1$ man gcc
```

To see the manual on the web, visit `http://i5.nyu.edu/~mm64/man/` and type `gcc`.

To see the version number of `gcc`,

```
2$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/opt/gcc450/bin/../libexec/gcc/sparc-sun-solaris2.10/4.5.0/lto-
Target: sparc-sun-solaris2.10
Configured with: ../configure --prefix=/opt/gcc4 --build=sparc-sun-solaris2.10 --wi
Thread model: posix
gcc version 4.5.0 (GCC)
```

We will tell the compiler to place the executable file in the `bin` directory of your home directory. The following command will give an error message if you do not already have a `bin`.

```
3$ ls -ld ~/bin                    "list" with lowercase L's
```

If you do not already have it, create it by saying

```
4$ mkdir ~/bin                     "make directory"
5$ ls -ld ~/bin
```

Compile the program `prog.c` and place the executable file `prog` in the `bin` subdirectory of your home directory.

```
6$ gcc -o ~/bin/prog prog.c                              minus lowercase O
7$ ls -l ~/bin/prog        Verify that we created an executable file named ~/bin/prog.
8$ prog hello              Run the program, giving it the argument hello.
9$ echo $?                 See the program's exit status; should be zero for EXIT_SUCCESS.
```

Our convention will be to show output in a box. With the argument `hello`, the output will be

```
I received the argument hello.
```

Direct the output to a file in Windows

To store the standard output of a C or C++ program into a file on the disk in Windows, go to the command prompt and specify the filename after the > symbol.

```
Start → Programs → Accessories → Command Prompt
C:\> cd to the directory that contains prog.exe
C:\> prog.exe hello                    send output to screen
C:\> echo %errorlevel%                 See the program's exit status; should be zero for EXIT_SUCCESS.
C:\> prog.exe hello > prog.out         send output to the file prog.out
C:\> type prog.out
```

Keep the `printf` in the above line 11; do not change it to `fprintf`.

The same program, in C++

Since this is a C++ program, its name ends in uppercase `.C`. Rename it if your platform demands a different convention.

The keywords are the same in both languages: `main`, `if`, and `return`. So are the parentheses, curly braces, quotation marks, and exit status codes. But the output statements are completely different. The `std::cout` in lines 12–14 and the `std::cerr` in lines 7–8 are two destinations for output, called *output streams* in C++. They lead to the same destinations as the C file pointers `stdout` and `stderr`. The `c` in `cout` and `cerr` stands for ''character''; we also have `wcout` and `wcerr` for ''wide characters'' such as Chinese and Unicode. The annoying prefix `std::` will be removed on p. 20.

	C *file pointers*	*C++* *input and output streams*
standard input	`stdin`	`std::cin`
standard output	`stdout`	`std::cout`
standard error output	`stderr`	`std::cerr`

Like `stdout` and `stderr`, the words `cout` and `cerr` belong to the C++ Standard Library, not to the language itself. We must therefore include the header file `iostream` in which they are declared. The GNU C++ compiler on our Unix machine i5.nyu.edu is named `g++`. For this compiler, the `iostream` file is in the directory `/opt/gcc450/include/c++/4.5.0`. I discovered this by giving the `-H` option to `g++` when compiling the program. The file used to be named `iostream.h` or even `stream.h`, but the names of the C++ Standard Library header files no longer end with `.h`.

The operator `<<` is pronounced ''put to''. It represents output because it points toward the destination: the `cout` in line 12 or the `cerr` in line 7. Forthcoming will be a lengthy explanation. For now, recall that an operator may, under certain circumstances, call a function (p. 7). This `<<` operator calls a function named `operator<<`, which is (roughly) the C++ equivalent of the function `printf`.

A one-line comment may be delimited with the double slash in line 4. The comment starts at the double slash (no whitespace between them) and ends at the end of the line. No terminating delimiter is necessary. For multi-line comments, you can still use the old-fashioned `/*` and `*/`.

As in C, you might have to insert `system("PAUSE");` immediately before line 16 to prevent the program's output from disappearing before you can read it.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/prog/prog.C`

```
1 #include <iostream>
2 #include <cstdlib>
```

```
 3
 4 int main(int argc, char **argv)   //char *argv[] would mean the same thing
 5 {
 6     if (argc != 2) {
 7         std::cerr << argv[0];
 8         std::cerr << ": requires 1 command line argument\n";
 9         return EXIT_FAILURE;
10     }
11
12     std::cout << "I received the argument ";
13     std::cout << argv[1];
14     std::cout << ".\n";
15
16     return EXIT_SUCCESS;
17 }
```

To see the online manual for `g++`, give the following command.

```
1$ man g++
```

To see the manual on the web, visit `http://i5.nyu.edu/~mm64/man/` and type `g++`.

To see the version number of `g++`,

```
2$ g++ -v
Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=/opt/gcc450/bin/../libexec/gcc/sparc-sun-solaris2.10/4.5.0/lto-
Target: sparc-sun-solaris2.10
Configured with: ../configure --prefix=/opt/gcc4 --build=sparc-sun-solaris2.10 --wi
Thread model: posix
gcc version 4.5.0 (GCC)
```

We will tell the compiler to place the executable file in the `bin` directory of your home directory. The following command will give an error message if you do not already have a `bin`.

```
3$ ls -ld ~/bin                          "list" with lowercase L's
```

If you do not already have it, create it by saying

```
4$ mkdir ~/bin                           "make directory"
5$ ls -ld ~/bin
```

Compile the program `prog.C` and place the executable file `prog` in the `bin` subdirectory of your home directory.

```
6$ g++ -o ~/bin/prog prog.C                          minus lowercase O
7$ ls -l ~/bin/prog        Verify that we created an executable file named ~/bin/prog.
8$ prog hello              Run the program, giving it the argument hello.
9$ echo $?                 See the program's exit status; should be zero.
```

```
I received the argument hello.
```

How can << mean "output" as well as "left shift"?

An *overloaded* operator is one that has two or more meanings, depending on the data type(s) of its operand(s). Operator overloading is present even in C, but no one talks about it. For example, the + in line 7 means "integer addition" because its operands are integers; the + in line 8 means "double addition" because its operands are `double`'s. Although we write them with the same operator, these are very

different operations. (The + in line 9 also means double addition. Line 9 copies the value of i into an anonymous temporary of type double, and then adds the temporary and d. Both operands are double's.

In C, we need to know the data types of these sums if we want to output them with the correct format of printf. This will be cleaned up in C++.

```
1     int i = 10;                    /* C example */
2     int j = 20;
3
4     double d = 3.1415926535897932385;
5     double e = 2.7182818284590452353;
6
7     printf("%d\n", i + j);         /* int addition */
8     printf("%f\n", d + e);         /* double addition */
9     printf("%f\n", i + d);         /* double addition */
```

Operator overloading is more noticeable in C++. The << operator in line 12 means ''left shift'' because its left operand is an integer. The << in line 13 means ''output'' because its left operand is an output stream.

```
10    int i = 10;                    //C++ example
11    int j = 20;

12    int k = i << j;                //left shift
13    std::cout << i;                //output
```

Two variables with the same first name

The purpose of a last name is to allow two or more people to have the same first name. Examples in English are in column 1.

```
Bill Clinton                         std::cout
Bill Gates                           different::cout
```

For exactly the same reason, a C++ variable can have a last name. Examples are in column 2 above. The last name of cout is std because cout belongs to the C++ Standard Library. We could also have another cout with a different last name.

In English the first name is written first, with a space between the first and last names. In C++ the last name is written first, with the class scope operator (the double colon) between the names. No space is allowed between the two colons.

A family of variables with the same last name is called a *namespace.* The most common example is the namespace std, many of whose members are declared in the header file iostream. For the present, however, the variables that we create will have no last names.

Remove the std::

I wish we didn't have to call std::cout and std::cerr by their full names all the time. And in fact, we don't. The *using directive* in line 3 will put us on a first-name basis with all the members of the std namespace.

This directive is convenient for our small programs, but might produce unexpected results for larger ones. Namespace std contains hundreds of variables, functions, data types, and ''templates''. See p. 1023 for a way to be selective.

In older versions of C++, cout and cerr had no last name at all. The using directive was not needed or even permitted.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/namespace/using.C

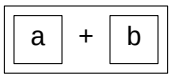
```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;   //using directive: may not be needed on your platform.
 4
 5 int main(int argc, char **argv)
 6 {
 7     if (argc != 2) {
 8         cerr << argv[0];
 9         cerr << ": requires 1 command line argument\n";
10         return EXIT_FAILURE;
11     }
12
13     cout << "I received the argument ";
14     cout << argv[1];
15     cout << ".\n";
16     return EXIT_SUCCESS;
17 }
```

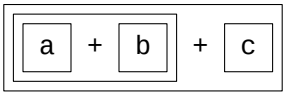
```
I received the argument hello.
```

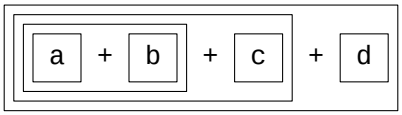
The << operator

Now that we've simplified the `std::cout`, let's work on the `<<` operator. We will walk through two examples with the familiar operators `+` and `=`, and then treat `<<` the same way.

(1) `+` is a binary operator whose operands must both be numbers. Since the resulting expression `a + b` has a value, it can be an operand of another `+`.



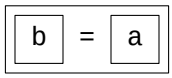


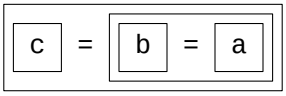


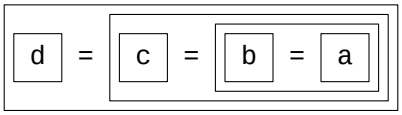
(2) Another binary operator, this time with right-to-left associativity, is the assignment operator `=`. The expression `b = a` installs a new value into `b`. But the expression does more: it has a value of its own. (Every non-`void` expression in C and C++ has a value.) This value is the new value of the left operand, `b`. We can easily verify this in C.

```
1     double a = 3.14159265358979323846;
2     int b = 10;
3
4     printf("%d\n", b = a);   /* The value of the expression b = a is 3. */
```

Since the expression `b = a` has a value, it can be the operand of another `=`.

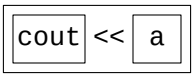


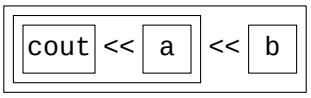


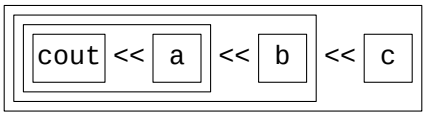


(3) `<<` is a binary operator with left-to-right associativity. If its left operand is an integer, it means left shift; if its left operand is an output stream, it means output. For example, the expression `cout<<a` outputs the value of `a`. But the expression does more: it has a value of its own. Its value is the value of the left operand, `cout`.

Since the expression `cout<<a` has a value that is an output stream, it can be used as the left operand of another `<<` that performs output.







Therefore the lines 8−9 on p. 21

```
5          cerr << argv[0];
6          cerr << ": requires 1 command line argument\n";
```

can be combined to

```
7          cerr << argv[0] << ": requires 1 command line argument\n";
```

The two `<<` operators in the above line 7 will be executed from left to right, because they are adjacent to a common operand and have left-to-right associativity. They are adjacent even though they are separated by an `[]` operator. The subexpression `argv[0]` is evaluated first, i.e., boiled down to a single value, and the two `<<`'s are left competing for this value. They *become* adjacent after the `[]` is gone. Another example is on p. 24.

Similarly, lines 13−15 on p. 21

```
8      cout << "I received the argument ";
9      cout << argv[1];
```

```
10      cout << ".\n";
```

can be combined to

```
11      cout << "I received the argument " << argv[1] << ".\n";
```

▼ **Homework 1.3a: rewrite expressions**

Each of the following questions gives away another's answer.

(1) Write the following three expressions as one big expression that has the same side effects (p. 14) in the same order.

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

(2) Write the following expression as three separate expressions that have the same side effects in the same order.

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

(3) Write the following three expressions as one big expression that has the same side effects in the same order.

```
cout << a
cout << b
cout << c
```

(4) Write the following expression as three separate expressions that have the same side effects in the same order.

```
cout << a << b << c
```

Answer the next pair of questions after seeing the `>>` operator on pp. 30−31.

(5) Write the following three expressions as one big expression that has the same side effects in the same order.

```
cin >> a
cin >> b
cin >> c
```

(6) Write the following expression as three separate expressions that have the same side effects in the same order.

```
cin >> a >> b >> c
```

▲

▼ **Homework 1.3b: combine consecutive output statements**

> It is very difficult to be king when the gods are changing.
>
> —James A. Michener, *Hawaii*, Chapter II

Compile and run the "I received the argument" C++ program in pp. 20−21. You may have to change the program's filename suffix from `.C` to `.CPP`, `.CXX`, or something else. You may have to `#include iostream.h` instead of `iostream`, and/or `stdlib.h` instead of `cstdlib`. The `using` directive might not be needed; in fact, in might not even be allowed. You may need a `system("PAUSE");` immediately before each `return` from `main`.

Combine consecutive output statements into one big output statement as on pp. 22−23. Do this for both `cout` and `cerr`.

▲

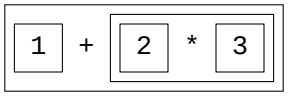
Output an expression with an operator of its own

We will walk through an example with the multiplication operator `*`, and then treat the output operator `<<` the same way.

Line 1 multiplies `2` and `3`, and uses their product as the operand of the next operator, the `+`. No parentheses are required to execute the `*` before the `+`, because `*` has higher precedence than `+`.

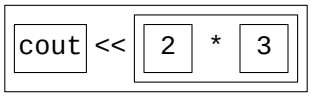
```
1     int i = 1 + 2 * 3;
```

We diagram only the expression `1 + 2 * 3`, not the rest of the statement.



Similarly, Line 2 multiplies `2` and `3`, and uses their product as the operand of the next operator, the `<<`. No parentheses are required to execute the `*` before the `<<`, because `*` has higher precedence than `<<`.

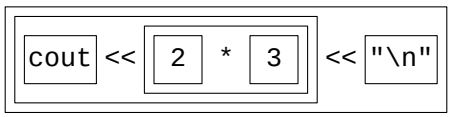
```
2     cout << 2 * 3;
3     cout << "\n";
```




```
6
```

The above lines 2−3 should be combined to

```
4     cout << 2 * 3 << "\n";
```



The subexpression `2 * 3` is boiled down into the single number `6` *before* the two `<<`'s are executed. This causes the `<<`'s to become adjacent (p. 22), allowing precedence and associativity to determine that they are executed from left to right.

Output a low-precedence expression

Line 1 ''bitwise ands'' `2` and `3` and uses the result as the operand of the next operator, the `+`. Parentheses are required to execute the `&` before the `+`, because binary `&` has lower precedence than binary `+`.

Similarly, line 3 ''bitwise ands'' `2` and `3` and uses the result as the operand of the next operator, the `<<`. Parentheses are required to execute the `&` before the `<<`, because binary `&` has lower precedence than `<<`. Another example is in line 17 of `static_cast` on p. 65.

```
1     int i = 1 + (2 & 3);
2
3     cout << (2 & 3);
```

```
4     cout << "\n";
```

```
2
```

The above lines 3−4 should be combined to

```
5     cout << (2 & 3) << "\n";
```

▼ Homework 1.3c: insert parentheses only where necessary

Each statement should evaluate and output the right operand of the <<. Insert parentheses where necessary to force this operand to be everything between the << and the semicolon. For example, in line 1 the right operand of the << should be `a + b`, not merely `a`. Are parentheses need to accomplish this?

The two <<'s in line 11 should be executed from left to right. Each one should output its right operand. The right operand of the first << should be `d`; that of the second should be

```
d == 1 ? " dollar" : " dollars"
```

Line 12 is a silly example. The right << should be a shift operator; the left, the output operator. I want to left-shift the 10 three times and output the result. Each left-shift should double the number, so three left-shifts should octuple it. The result should be 80. Unfortunately, it prints 103 because the right << means ''output''. By inserting parentheses in the correct place, you will change the meaning of the right << to ''left-shift''.

```
 1     cout << a + b;
 2     cout << a += b;
 3     cout << p - a;
 4     cout << a[i];
 5     cout << f(x);       //These parentheses are the function call operator.
 6     cout << -i;
 7     cout << i & 0xF;    //Output the four least significant bits of i.
 8     cout << d == 10;    //prints as 1 for true, 0 for false; see p. 354
 9     cout << old[y][x] ? 'X' : '.';
10     cout << d == 1 ? "dollar" : "dollars";
11     cout << d << d == 1 ? " dollar" : " dollars";
12     cout << 10 << 3;    //want to multiply 10 × 8.
```

▲

All operators obey the same rules

The i/o operators and the arithmetic ones obey the same rules.

(1) Why can line 1 string together as many <<'s as we want? For the same reason that line 2 can string together as many +'s as we want: little expressions may be combined to form bigger ones.

(2) Why are the <<'s in line 1 executed from left to right? For the same reason that the +'s in line 2 are executed from left to right: << and + have left-to-right associativity.

(3) Why is the * in line 1 performed before the two surrounding <<'s? For the same reason that the * in line 2 is performed before the two surrounding +'s: * has higher precedence than << and +.

(4) Why does line 1 need parentheses to perform the & before the << in front of it? For the same reason that line 2 need parentheses to perform its & before the + in front of it: & has lower precedence than << and +.

```
1     cout << b << c << d * e << (f & g);
2        a = b + c + d * e + (f & g);
```

Quotation marks and newlines

C++ has the same single and double quotes that C has. Single quotes must enclose exactly one character; double quotes can enclose zero or more.

I usually write double quotes even when they enclose only one character. It's easier to write only one kind of quote, and I would have to change the single quotes to double quotes anyway when I add extra characters:

```
1     cout << "The coordinates are " << x << ',' << y << '\n';
2     cout << "The coordinates are " << x << ", " << y << ".\n";
```

Although it does take longer to place a double quoted character into the output stream, the extra time is insignificant compared to that needed for the character to be read on a screen by a human being or written to a disk.

Lines 3−5 do the same thing. Please don't write 3 or 4: they would just annoy people. Line 5 is simpler.

```
3     cout << "hello" << '\n';        //bad
4     cout << "hello" << "\n";        //bad
5     cout << "hello\n";              //good
```

Line 6 would be bad in any language. What would be the point of outputting whitespace (e.g., a blank or tab) immediately before a newline? No one would ever see the whitespace. Write line 7 instead, but not because it saves one byte of memory and a millionth of a second. Do it because anyone seeing line 6 would think they have a corrupted version of the source code, since no one in their right mind would write like that.

```
6     cout << "hello \n";             //bad
7     cout << "hello\n";              //good
```

Flush the output buffer

There's another way to output the newline character, but we have to talk about *buffering* first.

We often imagine that each output statement, `printf` in C or `<<` in `C++`, sends data directly to a destination in the outside world. But in real life, the outgoing data may spend time in a holding area in the computer's memory, called an *output buffer.* Each output statement places more data in the buffer. When it is full, all the data in the buffer is flushed to the outside world in one big convoy.

Buffered output is faster than performing a separate output operation for each statement. For example, each write to a file on the disk moves the read/write head in the disk drive. If we consolidate several small disk writes into one big write, fewer motions will be required.

Sometimes, however, we want to flush the buffer before it is full. For example, a critical message cannot be allowed to languish in a buffer before it is displayed. In C, we can flush the standard output buffer at any time by calling `fflush`:

```
1     printf("All bomber groups have reached their Fail-Safe points.\n");
2     fflush(stdout);
```

In C++, we flush an output buffer by outputting a mysterious something named `endl` (“end line”, with a lowercase L). Outputting the `endl` causes two things to happen. It outputs a newline character, and then flushes the output buffer.

```
3     cout << "All bomber groups have reached their Fail-Safe points." << endl;
```

`endl` is an example of an *i/o manipulator:* something that causes a side effect when it is output or input. There will be many others.

Output the various data types

The fourteen *built-in* data types are those built into the language. C++ has all the built-in types that C has (except the `long long` types in the more recent versions of C), plus a ‘‘wide character’’ type `wchar_t` for large sets of characters such as Chinese or Unicode. Use `char` and `wchat_t` for characters; `signed char` and `unsigned char` for narrow integers.

```
bool
wchar_t
char    unsigned char   signed char
short   unsigned short
int     unsigned int
long    unsigned long

float
double
long double
```

C++ also has *user-defined* data types such as enumerations and classes. Finally, it has *derived* data types: pointers to, references to, arrays of, and functions that take and return, all of the above. They are declared the same way as in C, but an initial value should always be provided for each variable. More on this shortly.

The `printf` function in C depends on the `%` formats to tell what type of output to perform: `%d` for integer, `%f` for double, `%s` for string. The `<<` operator in C++ gets this information from the data types of its operands. There are no formats for us to write. (How this works is on pp. 349−350.)

Consider the first (leftmost) `<<` in line 19. Its left operand `cout` is of data type ‘‘output stream’’, so the `<<` means output rather than left shift. Its right operand `i` is of data type `int`, so the `<<` means ‘‘integer output’’, like the C `printf("%d"`. Now consider the second `<<` in line 19. Its left operand `cout << i` is of data type ‘‘output stream’’, so the `<<` means output. Its right operand `"\n"` is of data type ‘‘string of chars’’ because of the double quotes, so the `<<` means ‘‘string output’’, like the C `printf("%s"`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/io/builtin.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     //If you don't have bool, uncomment line 8 to implement it.
 8     //enum bool {false, true};
 9
10     bool b = true;        //or false
11     char c = 'A';
12     int i = 10;
13     double d = 1.0 / 3.0;
14     long double ld = 1.0L / 3.0L;
15     char s[] = "hello";   //Array of 6 char's; put '\0' into s[5].
16
17     cout << b << "\n";
18     cout << c << "\n";
19     cout << i << "\n";
20     cout << d << "\n";
21     cout << ld << "\n";
22     cout << s << "\n";   //Print up to, but not including, the '\0' in s[5].
```

```
23      cout << &i << "\n";
24
25      return EXIT_SUCCESS;
26  }
```

By default, a `bool` prints as `1` or `0`; to print as the words `true` or `false`, see p. 354. A `double` prints with six significant digits; to change this, see pp. 355−356.

```
1               Line 17: bool prints as 1 or 0.
A
10
0.333333        Line 20: double prints as 6 significant digits.
0.333333
hello
0xffbff624      Line 23: pointers print in hexadecimal on most platforms.
```

Warning: the value of `d` is slightly less than one third because the machine is binary, not ternary. A `double` is stored as a power of 2, times a fraction whose denominator is 2^{53}. (We say that the `double` has a <I>matissa</I> of 53 bits.) The closest we can get to one third is

$$2^{-1} \times \frac{6,004,799,503,160,661}{2^{53}} = \frac{6,004,799,503,160,661}{18,014,398,509,481,984}$$

$$= .333333333333333314829616256247390992939472198486328125$$

See `numeric_limits` on pp. 745−747.

The above lines 17−23 should be combined to one statement.

```
27      cout << b << "\n" << c << "\n" << i << "\n" << d << "\n" << ld
28          << "\n" << s << "\n" << &i << "\n";
```

Then split the statement up the way the output appears.

```
29      cout << b << "\n"
30          << c << "\n"
31          << i << "\n"
32          << d << "\n"
33          << ld << "\n"
34          << s << "\n"
35          << &i << "\n";
```

Three more reasons why << in C++ is better than printf in C

(1) The most common `printf` mistakes are not caught until runtime, if at all.

```
1       char c = 'A';       /* C example */
2       printf("%s\n", c);  /* error message (unlikely) or garbage at runtime */
```

The most common `<<` mistakes are caught at compile time. Wherever possible, C++ moves the error messages from runtime to compile time.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/io/error.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 int main()
6 {
```

```
 7     char c = 'A';
 8
 9     cout << c < "\n";
10     cout << c "\n";
11
12     return EXIT_SUCCESS;
13 }
```

Here are the error messages on my platform. I can't pretend that they are of much help, but at least you know at compile time that *something* is wrong.

```
error.C: In function 'int main()':
error.C:9:14: error: no match for 'operator<' in 'std::operator<< [with _Traits
= std::char_traits<char>](((std::basic_ostream<char>&)(& std::cout)), ((int)c))
< "\012"'
error.C:9:14: note: candidates are: operator<(const char*, const char*)
<built-in>
error.C:9:14: note:                 operator<(void*, void*) <built-in>
error.C:10:12: error: expected ';' before string constant
```

(2) `<<` also executes faster than `printf`. Each call to `printf` has to loop through all the characters in its first argument and search for percent signs. Based on the character after the percent sign, it decides upon the output format. In effect, `printf` is an interpreter for a little language.

```
14     /* simplified outline of what printf does */
15
16     char *p;
17
18     for (p = address of first argument of printf; *p != '\0'; ++p) {
19         if (*p == '%') {
20             switch (*++p) {      /* Examine the character after the '%'. */
21             case 'd':
22                 output an integer in decimal;
23                 break;
24
25             case 'f':
26                 output a float or double;
27                 break;
28
29             case 's':
30                 output a string until the terminating '\0';
31                 break;
32
33             /* etc. */
34             }
35         }
36     }
```

The `<<` operator, on the other hand, does this decision making once and for all at compile time, without all the looping and switching. This is important if the `<<` statement is inside a loop (which it usually is).

(3) `printf` is not extensible. This problem is bigger in C++, where we will invent many new data types:

```
37 struct blood_pressure {   /* a new data type */
38     int systolic;         /* bigger number: contract */
39     int diastolic;        /* smaller number: expand */
```

```
40 };
41
42     /* Create a variable of the new data type. */
43     struct blood_pressure b = {120, 80};
44
45     printf("%b\n", b);                              /* Can't invent %b. */
46     printf("%d/%d\n", b.systolic, b.diastolic);    /* Must do this instead. */
```

After we do operator overloading, we'll be able to output a `blood_pressure` with the same `<<` that we use for the built-in types See p. 335.

```
47     blood_pressure b(120, 80);      //Don't even need the keyword struct.
48     cout << b << "\n";              //Output a blood_pressure.
```

The >> operator

Input is the counterpart of output, but more can go wrong. The `cin` in lines 11, 14, and 17 is a source of input, called an *input stream*, analogous to the C `stdin`.

The operator `>>` is pronounced "get from". It was chosen to represent input because it points away from the source, in this case `cin`. If its right operand is an `int`, as in line 14, it will perform `int` input. If its right operand is a string, as in line 11, it will perform `string` input. C++ string input is just like the C `scanf("%s", ...)`: it will input only one word, not necessarily the entire line of input. And if the user inputs a word in line 11 that is longer than the array in line 10, the `>>` will overwrite the memory after the array. We'll fix this later when we do class `string`.

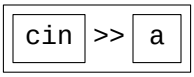
—On the Web at
`http://i5.nyu.edu/~mm64/book/src/io/input.C`

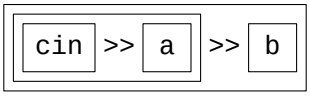
```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     cout << "Please type the date and press RETURN.\n"
 8         << "For example, January 1 2010\n";
 9
10     char month[256];   //uninitialized variable
11     cin >> month;      //Put a '\0' into month after the last character.
12
13     int day;           //uninitialized variable
14     cin >> day;        //scanf("%d", &day); would have needed an ampersand.
15
16     int year;          //uninitialized variable
17     cin >> year;
18
19     cout << "month == " << month << "\n"
20         << "day == " << day << "\n"
21         << "year == " << year << "\n";
22
23     return EXIT_SUCCESS;
24 }
```

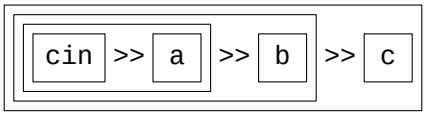
```
Please type the date and press RETURN.
For example, January 1 2010
November 16 2010          The user types this line.
month == November
day == 16
year == 2010
```

>> is a binary operator because it requires two operands. If its left operand is an integer, it means right shift; if its left operand is an input stream, it means input. For example, the expression `cin >> a` installs a new value into `a`. But the expression does more: it has a value of its own. Its value is the value of its left operand, `cin`.

Since the expression `cin >> a` has a value, it can be the left operand of another >>:







Therefore lines 11, 14, 17 of the above program could be combined to

```
25      cin >> month >> day >> year;
```

Concatenate strings at compile time

C lets us write a double-quoted string in two or more parts:

```
 1 printf("hello");        /* output 5 characters */
 2 printf("hel" "lo");     /* output 5 characters */
```

This lets us write a long string on separate source lines:

```
 3 printf("hel"
 4     "lo");              /* output 5 characters */

 5 printf("supercalifragilistic"
 6     "expialidocious");

 7 char a[] = "hel" "lo";   /* an array of 6 characters, including one '\0' */
```

C++ lets us do the same thing, so the above lines 7–8 could be written with only one << operator:

```
 8     cout << "Please type the date and press RETURN.\n"
 9         "For example, January 1 2010\n";
```

In fact, we could even get rid of the <<'s at the start of lines 20 and 21. But don't do it—people would just get confused.

Why >> is less error prone than scanf

`scanf` and `printf` agree on the same format letter for `int` input and output (the `%d` in lines 8–9). But for other data types, they sometimes use different formats.

```
 1     short s;                  /* C example */
 2     int i;
 3     double d;
 4
 5     scanf("%hd", &s);         /* "%s" was already taken for "string". */
 6     printf("%d\n", s);
 7
 8     scanf("%d", &i);
 9     printf("%d\n", i);
10
11     scanf("%lf", &d);         /* can't input a double with "%f" */
12     printf("%f\n", d);        /* but can output a float or double with "%f" */
```

1.4 Declarations and their Placement

Declare a variable in a block

A *block* is a group of zero or more statements enclosed in {curly braces}. The most common examples are the body of a function, loop, `if`, or `else`.

```
 1 void f(int n)
 2 {
 3     //The body of a function is the most common example of a block.
 4 }

 5     for (i = 1; i <= 10; ++i) {
 6         //The body of a for loop is a block.
 7     }

 8     if (a == n) {
 9         //This is a block.
10     } else {
11         //This is another block.
12     }
```

In C and C++, a variable declared in a block can be mentioned only within that block. We say that the variable's *scope* (habitat) extends only from the declaration to the closing curly brace } at the end of the block.

In versions of C prior to C99, a declaration in a block must be at the start of the block. We will demonstrate why this is bad with a program that takes an integer as its command line argument and outputs the sum of the positive integers from 1 up to that one.

The expression `argv[1]` in line 15 is the address of the first character of the first command line argument. The function `atoi` examines this character and the following ones, hopefully all digits, and returns the number that is spelled out by these digits. If the characters are not digits, `atoi` returns zero, making it impossible to tell the difference between a legitimate argument of `"0"` and an argument of garbage. We could remedy this now with the `strtol` function or later with an `istringstream` object.

In the block that extends from lines 5 to 23, the declarations (lines 6–8) must come before the statements (lines 10–22). But this rule often forces us to leave a gap between the variable's declaration and its initialization. For example, `n` contains garbage from line 6 to line 15; `i` contains garbage from 7 to 17. It's dangerous to leave garbage in variables for such long periods.

There is one variable, the `sum` in line 8, that can be initialized in its declaration. But now we have the opposite problem: the initialization will be wasted if the program ends prematurely in lines 10−13. It's not our fault. C is rigid.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/declare/sum.c`

```
 1 #include <stdio.h>     /* C example */
 2 #include <stdlib.h>    /* for atoi */
 3
 4 int main(int argc, char **argv)
 5 {
 6     int n;         /* not initialized until line 15 */
 7     int i;         /* not initialized until line 17 */
 8     int sum = 0;   /* This initialization is wasted effort if argc != 2. */
 9
10     if (argc != 2) {
11         fprintf(stderr, "%s: requires one command line argument\n", argv[0]);
12         return EXIT_FAILURE;
13     }
14
15     n = atoi(argv[1]);
16
17     for (i = 1; i <= n; ++i) {
18         sum += i;   /* means sum = sum + i */
19     }
20
21     printf("The sum of the numbers from 1 to %d is %d.\n", n, sum);
22     return EXIT_SUCCESS;
23 }
```

```
prog 10                    This is the command line that launches the program.
The sum of the numbers from 1 to 10 is 55.
```

In C++, a variable declaration in a block need not be at the start of the block, although it must still be written before the variable is used. Don't declare the variable until you are ready to initialize it; then declare and initialize it in the same statement. *If you find yourself declaring a variable without initializing it in the same statement, you have declared it too soon.*

An extra benefit is that the variable `n` in line 13 is now initialized rather than assigned to. It can therefore be a `const`. (See pp. 302−303 for a weighty discussion of the other advantages of initialization over assignment.)

C++ permits us to tuck the declaration for `i` into the `for` loop at line 15. By declaring `i` at this point, we're announcing that it will be used only inside the loop. On the other hand, the declarations of `sum` and `n` in lines 12−13 announce that they will be used outside the loop. The position of a declaration documents your intent. Position the declaration to make the scope of the variable as small as possible: no variable should outlive its usefulness.

In older versions of C++, the scope of the variable `i` extended from line 15 to line 21. In newer versions, the scope of `i` extends from 15 only to 17. Some versions give you a choice, e.g., with the `-ffor_scope` option of the GNU compiler `g++`. The same rule applies to a variable declared within the parentheses of an `if` or `while`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/declare/sum.C`

```
1 #include <iostream>   //C++ example
2 #include <cstdlib>
```

```
 3 using namespace std;
 4
 5 int main(int argc, char **argv)
 6 {
 7     if (argc != 2) {
 8         cerr << argv[0] << ": requires one command line argument\n";
 9         return EXIT_FAILURE;
10     }
11
12     int sum = 0;
13     const int n = atoi(argv[1]);
14
15     for (int i = 1; i <= n; ++i) {
16         sum += i;
17     }
18
19     cout << "The sum of the numbers from 1 to " << n << " is " << sum << ".\n";
20     return EXIT_SUCCESS;
21 }
```

```
prog 10
The sum of the numbers from 1 to 10 is 55.
```

▼ Homework 1.4a: is the induction variable still in scope after the end of the for loop?

In C and C++, we can't have two variables with the same name in the same scope. Make whatever changes are necessary to avoid compilation errors. On p. 183, we will have an easier way of seeing how long a variable lasts.

In newer version of C++, no change should be needed. The scope of the `i` in line 7 extends only as far as line 15, permitting us to declare another `i` in line 17. Similarly, the scope of the `j` in line 8 extends only as far as line 10, permitting us to declare another `j` in line 12.

But in older versions of C++, the scope of the `i` in line 7 extends all the way to line 24, preventing us from declaring another `i` in line 17. You could rename the `i` in line 17. Or simply remove the keyword `int` from line 17, so that the `i` in 17 will be the same variable as the `i` in 7. Similarly, the scope of the `j` in line 8 extends all the way to line 15, preventing us from declaring another `j` in line 12. You could rename the `j` in line 12. Or simply remove the keyword `int` from line 12, so that the `j` in 12 will be the same variable as the `j` in 8.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/declare/forscope.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     for (int i = 0; i < 3; ++i) {
 8         for (int j = 0; j < 3; ++j) {
 9             cout << i << ", " << j << "\n";
10         }
11
12         for (int j = 0; j < 3; ++j) {
13             cout << i << ", " << j << "\n";
14         }
```

```
15      }
16
17      for (int i = 0; i < 3; ++i) {
18          for (int j = 0; j < 3; ++j) {
19              cout << i << ", " << j << "\n";
20          }
21      }
22
23      return EXIT_SUCCESS;
24  }
```

▲

The real reason to declare and initialize in the same statement

Suppose that C implicitly initialized every `int` to 0 unless you said otherwise:

```
1       int i;              /* Put 0 into i. */
2       int j = 10;
```

Then it would be wasteful to say lines 3−4, since the 0 that line 3 puts into `i` would be wiped out by the 10 in line 4.

```
3       int i;              /* Put 0 into i. */
4       i = 10;             /* Replace the 0 with 10. */
```

Instead of lines 3−4, it would be better to say

```
5       int i = 10;
```

Now let's come back to reality: an `int` is not implicitly initialized to 0 in C and C++. But an "object" may be initialized to 0 or to some other default value. In that case, the initialization in line 6 would be wasted because of line 7.

```
6       obj ob;             //Put 0 (or some default value) into ob.
7       ob = 10;            //Replace the 0 with 10.
```

It takes only a millionth of a second to initialize an `int`, but it might take a hundredth of a second—an eternity—to initialize an object. We never want to do it unnecessarily. Instead of lines 6−7, the program would execute faster if we said

```
8       obj ob = 10;
```

Even if you're using `int`'s, please write in the object style by declaring and initializing a variable in the same statement. This will make it possible later to use the same code for `int`'s and objects by means of a "template". See p. 634.

Five situations in which we can't declare and initialize in the same statement

C++ is not perfect. Here are five cases in which we can't initialize a variable in its declaration. Nonetheless, we should assign a value to the variable as soon as possible after declaring it. Do this by declaring it just before the assignment.

(1) A variable declared outside the body of any function is said to be *global.* A global can be mentioned by all the functions defined below its declaration. But if its initial value comes from a function argument, it must receive the value inside the body of the function.

```
1 const char *progname;               //global variable must be declared outside main
2
3 int main(int argc, char **argv)
4 {
5     progname = argv[0];             //value must be assigned inside main
```

(2) The variable is an array whose initial values are assigned in a loop. (Use the data type `size_t` for the number of elements in an array and for an array subscript. See p. 66.)

```
 6     const size_t n = 1000000;
 7     int a[n];                       //uninitialized variable
 8
 9     for (size_t i = 0; i < n; ++i) {
10         a[i] = i;
11     }
```

(3) The variable's initial value comes from input. There is no way to combine lines 12 and 13.

```
12     int n;                          //uninitialized variable
13     cin >> n;
```

(4) The variable's initial value comes from a function via pass-by-reference (pp. 69−70). There is no way to combine lines 18 and 19.

```
14 void f(int *p);         //function declaration or prototype
15
16 int main(int argc, char **argv)
17 {
18     int i;                          //uninitialized variable
19     f(&i);                          //give value to i
```

Here's the same example, with the argument changed from a pointer to a ''reference'' (pp. 71−72).

```
20 void g(int& r);         //function declaration or prototype
21
22 int main(int argc, char **argv)
23 {
24     int i;                          //uninitialized variable
25     g(i);                           //give value to i
```

(5) The variable is given its initial value inside a loop or other block, but must be declared outside because it will be used after the loop or block is over.

```
26     int firstarg;                   //uninitialized variable
27
28     if (argc < 2) {                 //If there were no arguments
29         firstarg = 0;
30     } else {
31         firstarg = atoi(argv[1]);
32     }
33
34     cout << "The first argument is " << firstarg << ".\n";

35     int r;                          //uninitialized variable
36
37     while ((r = rand()) <= 100) {
38     }
39
40     cout << "The first random number greater than 100 was " << r << ".\n";
```

Since the `then` and the `else` each consist of one assignment statement (lines 29 and 31), we could use the `?:` operator instead of an `if`. This permits the variable `i` to be `const`.

```
41     const int firstarg = argc < 2 ? 0 : atoi(argv[1]);
```

Dead values

Except for the above five cases, every C++ variable should be initialized at its moment of birth. But some initializations are worse than useless.

A *dead* value is one that will never be used again. In the following example, the values 10, 30, and 50 are dead. Never store a dead value into a variable.

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     int i = 10;      //Dead value: never used because wiped out in line 8.
 8     i = 20;
 9     cout << i << "\n";
10
11     i = 30;          //Dead value: never used because wiped out in line 12.
12     i = 40;
13     cout << i << "\n";
14
15     int j = 50;      //Dead value: never used because program ends in line 16.
16     return EXIT_SUCCESS;
17 }
```

No variable should outlive its usefulness

No variable should be born before we have a initial value for it. Similarly, no variable should live beyond its last use. That is why we declared the `i` in the parentheses of the `for` loop in line 15 of `sum.C` in pp. 33−34. Here is another example.

In C, we often assign a value to a variable and test it in the same expression. This C++ program is in the same style (line 9).

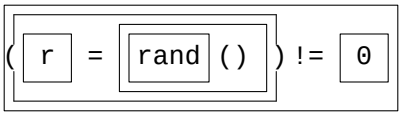
—On the Web at
`http://i5.nyu.edu/~mm64/book/src/declare/if1.C`

```
 1 #include <iostream>
 2 #include <cstdlib>    //for rand, exit, EXIT_SUCCESS
 3 using namespace std;
 4
 5 int main(int argc, char **argv)
 6 {
 7     int r;                     //uninitialized variable
 8
 9     if ((r = rand()) != 0) {   //assign and test
10         cout << "The first random number was " << r << ".\n";
11     } else {
12         cout << "The first random number was zero (" << r << ").\n";
13     }
14
15     return EXIT_SUCCESS;
16 }
```

The output will be the same each time we run the program because we made no call to the function `srand` before the call to `rand`.

```
The first random number was 16838.
```

In the above line 9, the function call operator goes first because of its higher precedence; the assignment operator goes second; and the comparison goes last because it its outside the parentheses. Another example is on pp. 86−87.



The most common examples of this idiom in C are

```
1     int c;
2     while ((c = getchar()) != EOF) {

3     char *p;
4     if ((p = malloc(n)) == NULL) {

5     FILE *out;
6     if ((out = fopen("outfile", "w")) == NULL) {
```

But C++ does not share C's rage to cram as much code as possible into a single expression. The C++ style would be to separate the assignment and the test. The assignment is now an initialization, which permits the variable to be a `const` in line 7.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/declare/if2.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main(int argc, char **argv)
 6 {
 7     const int r = rand();   //initialize
 8
 9     if (r != 0) {           //test
10         cout << "The first random number was " << r << ".\n";
11     } else {
12         cout << "The first random number was zero (" << r << ").\n";
13     }
14
15     return EXIT_SUCCESS;    //r is still alive at this point
16 }
```

```
The first random number was 16838.
```

The variable `r` is intended for use only within the bodies of the `if` and `else` in lines 9−13 of the above program. It has no business being alive all the way down to the last line of `main`. To restrict its scope to the bodies of the `if` and `else`, the following line 7 tucks its declaration and initialization into the parentheses of the `if`. The `if` will be true if the variable declared in the parentheses has a non-zero initial value; more precisely, if it has a `true` value when converted to a `bool`. Now `r` will no longer outlive the `if` and `else`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/declare/if3.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main(int argc, char **argv)
 6 {
 7     if (const int r = rand()) {   //initialize and test
 8         cout << "The first random number was " << r << ".\n";
 9     } else {
10         cout << "The first random number was zero (" << r << ").\n";
11     }
12
13     return EXIT_SUCCESS;          //r no longer exists at this point
14 }
```

```
The first random number was 16838.
```

Why is it so important to extinguish a variable as soon as we are done with it? Admittedly, the death of an integer will free up no resources. But a more complicated variable (an "object") might hold many things during its life: open files and network connections, dynamically allocated memory, locks of various kinds. Killing the variable ("destructing the object") will make it surrender these resources. Please get into the habit of killing off all your variables, even integers, as soon as you are done with them. As we mention in p. 634, programming in the same style with every data type will ease our transition to templates.

We have seen that a variable can be declared and initialized in the parentheses of a `for` loop or `if` statement. We can also do it in the parentheses of a `while` loop. An `r` is born each time we arrive at line 13; it dies each time we reach the `}` in line 15. There is actually a whole series of `const` variables, each with the same name `r`. Contrast the `while` loop in p. 36, in which there is only one variable `r` whose value keeps changing.

```
 1     for (int i = 0; i < 10; ++i) {
 2         cout << i << "\n";
 3     }
 4
 5     //i no longer exists here (in modern versions of C++)

 6     if (const int r = rand()) {
 7         cout << "The first random number was " << r << ".\n";
 8     } else {
 9         cout << "The first random number was zero (" << r << ").\n";
10     }
11
12     //r no longer exists here

13     while (const int r = rand()) {
14         cout << "The random number was " << r << ".\n";
15     }
16
17     //r no longer exists here
```

1.5 Translate a C Program into C++

J. H. Conway's Game of Life

John Horton Conway's ''Game of Life'' is the classic example of a ''cellular automaton''. This solitaire game was unleashed upon the world in Martin Gardner's ''Mathematical Games'' column in *Scientific American* magazine.

223 (October 1970): pp. 120−123, *The fantastic combinations of John Conway's new solitaire game "life"*
224 (February 1971): pp. 112−117, *On cellular automata, self-reproduction, the Garden of Eden, and the game "life"*

The playing board is made of rows and columns of square cells. Each cell is occupied or empty. In the days before computers, they used checkers on a checkerboard.

Initially, the user draws whatever picture strikes their fancy; three examples are shown below. The user plays no other rôle. He or she simply sits back and watches the picture evolve.

This happens according to three laws. We will pretend that the playing board is infinite, so we don't have to worry about edge cells. Each cell has eight neighbors.

(1) The Law of Survival says that the contents of a cell remain unchanged if exactly two of its eight neighbors are occupied.

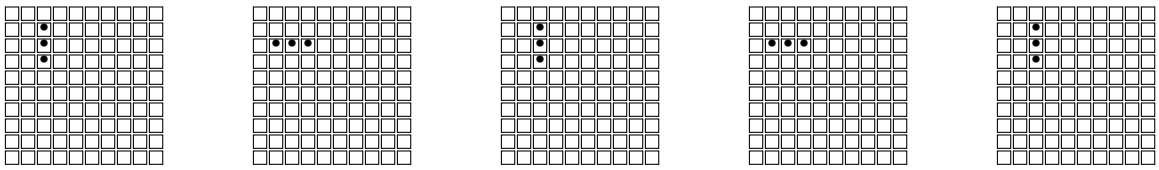
(2) The Law of Birth says that a cell becomes occupied if exactly three of its eight neighbors are occupied. If the cell is already occupied, it remains occupied.

(3) The Law of Death says that a cell becomes empty if less than two or more than three of its eight neighbors are occupied. If the cell is already empty, it remains empty.

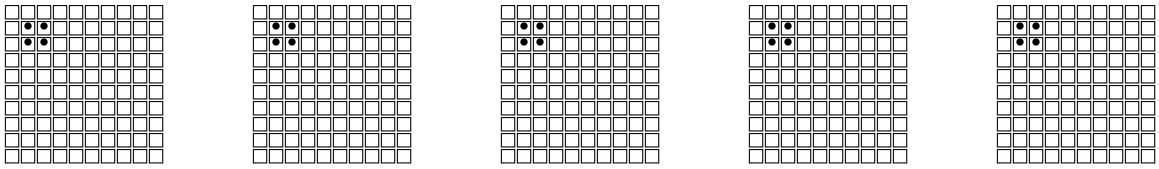
The hard part is that the laws must be applied to each cell simultaneously. We usually leave the picture untouched and build up the new picture, or *generation*, in a temporary array.

In the initial generation below, the occupied cell in the center of the blinker has two occupied neighbors. By the Law of Survival, it remains occupied in the next generation. The other two cells in the blinker each have one occupied neighbor. By the Law of Death, they become empty in the next generation. The two empty cells to the left and right of the central one each have three occupied neighbors. By the Law of Birth, they become occupied in the next generation. All the other cells remain unoccupied, by the Laws of Survival or Death.

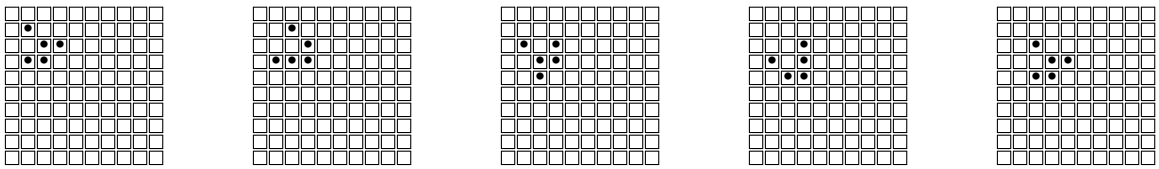
A *blinker* has period 2. It repeats itself every two generations.



A *paw print* is a still life.



A *glider* oozes to the lower right, exuding pseudopodia. It resumes its original shape after four generations.



To simplify lines 35−42, we print the generations vertically. The characters `X` and ''dot'' represent the occupied and unoccupied cells. To simplify lines 54−62, the 10×10 playing board that the user sees is

surrounded with a border of permanently unoccupied cells. This means that the underlying array has to be 12×12.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/life/life.c`

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>   /* for strcmp */
 4
 5 #define LIFE_YMAX 10
 6 #define LIFE_XMAX 10
 7
 8 int main()
 9 {
10     int old[LIFE_YMAX + 2][LIFE_XMAX + 2] = {   /* sorry y before x */
11         {0,  0, 0, 0, 0, 0, 0, 0, 0, 0,  0},
12
13         {0,  0, 0, 0, 0, 0, 0, 0, 0, 0,  0},
14         {0,  0, 1, 0, 0, 0, 0, 0, 0, 0,  0},   /* a glider */
15         {0,  0, 0, 1, 1, 0, 0, 0, 0, 0,  0},
16         {0,  0, 1, 1, 0, 0, 0, 0, 0, 0,  0},
17         {0,  0, 0, 0, 0, 0, 0, 0, 0, 0,  0},
18         {0,  0, 0, 0, 0, 0, 0, 0, 0, 0,  0},
19         {0,  0, 0, 0, 0, 0, 0, 0, 0, 0,  0},
20         {0,  0, 0, 0, 0, 0, 0, 0, 0, 0,  0},
21         {0,  0, 0, 0, 0, 0, 0, 0, 0, 0,  0},
22         {0,  0, 0, 0, 0, 0, 0, 0, 0, 0,  0},
23
24         {0,  0, 0, 0, 0, 0, 0, 0, 0, 0,  0}
25     };
26     int new[LIFE_YMAX + 2][LIFE_XMAX + 2];
27     int generation;
28     char buffer[256];
29     size_t x, y;     /* loop through all elements in the array */
30     int count;
31     size_t x1, y1;   /* subscript of each of the 8 neighbors of x, y */
32
33     for (generation = 0;; ++generation) {
34
35         /* Print the matrix. */
36         for (y = 1; y <= LIFE_YMAX; ++y) {
37             for (x = 1; x <= LIFE_XMAX; ++x) {
38                 /* sorry y before x */
39                 putchar(old[y][x] ? 'X' : '.');
40             }
41             putchar('\n');
42         }
43
44         printf("%d: Press c to continue, q to quit, and RETURN.\n",
45             generation);
46         scanf("%s", buffer);
47         if (strcmp(buffer, "c") != 0) {
48             break;
49         }
50
```

```
51          for (y = 1; y <= LIFE_YMAX; ++y) {
52              for (x = 1; x <= LIFE_XMAX; ++x) {
53
54                  /* How many of the 8 neighbors of element x, y
55                  are turned on? Don't count the element itself.*/
56                  count = -old[y][x];
57
58                  for (y1 = y - 1; y1 <= y + 1; ++y1) {
59                      for (x1 = x - 1; x1 <= x + 1; ++x1) {
60                          count += old[y1][x1];
61                      }
62                  }
63
64                  /* Law of Survival */
65                  if (count == 2) {
66                      new[y][x] = old[y][x];
67                  }
68
69                  /* Law of Birth */
70                  else if (count == 3) {
71                      new[y][x] = 1;
72                  }
73
74                  /* Law of Death */
75                  else {
76                      new[y][x] = 0;
77                  }
78              }
79          }
80
81          /* Copy new into old. */
82          for (y = 1; y <= LIFE_YMAX; ++y) {
83              for (x = 1; x <= LIFE_XMAX; ++x) {
84                  old[y][x] = new[y][x];
85              }
86          }
87      }
88
89      return EXIT_SUCCESS;
90  }
```

The above lines 64−77 may be combined to the single expression

```
91      new[y][x] = count == 2 ? old[y][x] : count == 3;
```

But don't do it. C++ does not share C's rage to cram as much code as possible into a single expression.

▼ Homework 1.5a: translate C to C++

Translate the above program from C to C++. We have not done classes yet, so do not use class `string`. Print the output in a monospace font.

Make these changes:

(1) Rename the program to tell the computer that it is now in C++.

(2) Include `iostream` instead of `stdio.h`. If you don't have `iostream`, you will have to include `iostream.h`.

(3) Include `cstdlib` instead of `stdlib.h`. If you don't have `cstdlib`, you will have to include `stdlib.h`.

(4) Include `cstring` instead of `string.h`. If you don't have `cstring`, you will have to include `string.h`.

(5) Do all i/o with `>>` and `<<`. The `using namespace std;` may not be needed or even allowed. Do not call any of the C i/o functions: `printf`, `putchar`, `scanf`, `fprintf`, etc.

(6) Use the C++ comment delimiter `//` for one-line comments.

(7) You can't use a keyword as the name of a variable, array, function, or anything else that has a name. Here are the 74 C++ keywords.

```
and             continue        goto            public          try
and_eq          default         if              register        typedef
asm             delete          inline          reinterpret_cast typeid
auto            do              int             return          typename
bitand          double          long            short           union
bitor           dynamic_cast    mutable         signed          unsigned
bool            else            namespace       sizeof          using
break           enum            new             static          virtual
case            explicit        not             static_cast     void
catch           export          not_eq          struct          volatile
char            extern          operator        switch          wchar_t
class           false           or              template        while
compl           float           or_eq           this            xor
const           for             private         throw           xor_eq
const_cast      friend          protected       true
```

(8) Change the macros to variables of data type `const size_t`. And now that they are no longer macros, let their names be all lowercase. To keep their scope as small as possible, declare them inside the `main` function.

(9) The `int`'s that represent numbers (`count`, `x`, `y`, etc.) should remain `int`'s. But the `int`'s that represent on/off or true/false (the array elements) should become `bool`'s. Set them to `true` or `false` instead of to `1` or `0` in lines 71 and 76. But to keep the picture legible, let the initial values of the array elements remain `1`'s and `0`'s even though they are now `bool`'s in lines 11–24. `bool`'s can be initialized with `1`'s and `0`'s:

```
92      bool a[] = {1, 0, 1, 0};        //= {true, false, true, false};
```

If your version of C++ has no `bool`, use `bool` anyway. Simply insert line 8 of `builtin.C` in p. 27 (without the comment delimiter) after the `#include`'s.

(10) Unfortunately, the arrays `new` and `buffer` cannot be initialized in their declarations; see p. 36, ¶ (2). But at least you should move their declarations down to the last possible moment, so they will contain garbage for the shortest possible time.

(11) Every other variable must be initialized in its declaration. Declare the loop counters (`y`, `x`, `y1`, etc.) immediately after the left parenthesis of each `for` loop if your version of C++ permits this. You discovered if it does in Homework 1.4a (pp. 34–35).

(12) Extra credit. Instead of writing to the standard output, write onto the screen by calling the `term.h` functions on p. 86. If the game is small enough to fit on the screen, display it in the upper left corner ("upper-left justified"). If the game is too big to fit on the screen, display as much of it as will fit. Display the prompt (`"Press c to continue, ..."`) below the game.

Call the `min` function to find which is smaller: the number of columns in the game or on the screen. (Ditto for the number of rows). Include the header file `<algorithm>` for `min`. Like `cout`, `min` belongs to namespace `std`; be sure to say `using namespace std`; In some versions of Microsoft Visual C++, `min` and `max` are named `_cpp_min` and `_cpp_max`.

The two arguments of `min` have to be the same data type (explanation on p. 652). The number of columns in the game will be a `size_t` (¶ (8) above); the number of columns on the screen will be an `unsigned` (the return type of `term_xmax`). If `size_t` is not another name for `unsigned` on your machine, you will have to cast one of the arguments of `min` to the type of the other.

Warning: the first argument of `term_get` and `term_put` is the column number, but the first subscript of a two-dimensional array is the row number.

For speed, you should `term_put` a new character only when it is different from the old character at that location. Call `term_get` to find out what the old character was. Do not `term_put` a newline onto the screen.

For the extra credit, do not bother to display the generation number: it is too much trouble to convert it into a series of digit characters. But if you feel compelled to display it anyway, construct an object of the class `ostringstream` on pp. 454–456. The `str` member function of class `ostringstream` returns a "`string` object", and the `c_str` member function of the object returns a pointer to a `char`.

```
93 #include <sstream>      //for ostringstream
94 using namespace std;
95
96     ostringstream ost;
97     ost << generation << ": Press c to continue, q to quit, and RETURN.";
98     term_puts(x, y, ost.str().c_str());
```

▲

1.6 Pointers and References

The introduction of object is the take-off point from C into C++. Ultimately this will be a new way to think about programming. Initially, however, our objects will merely be a notation for tying together the structures, pointers, and functions familiar from C. To visualize an object we will need a structure, a pointer thereto, and a function to which the pointer will be passed. Here is a review of this machinery.

1.6.1 Review of Pointers

A pointer to a stand-alone variable

We will begin by considering a *stand-alone* variable, one that is not an element of an array. A variable in memory occupies one or more bytes. The *address* of a variable is the address of the byte that has the lowest address. The value of a variable may change as the program runs, but its address and number of bytes stay the same. A variable cannot ooze around in memory.

Line 10 outputs the address of `i`. Line 13 stores this address into a *pointer,* a variable that can hold an address. Since the value of `p` is the address of `i`, we say that `p` *points to* `i`.

The unary operator `*` in line 15 *dereferences* `p`: it uses the value of `p` to get the value of the variable to which `p` points. The `*` fetches an `int` from memory, as opposed to a `double` or some other data type, because of the declaration of `p` in line 13. The value of the expression `*p` in line 15 is the value of `i`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer/pointer.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     int i = 10;
 8
```

```
 9     cout << "The value of i is " << i << ".\n"
10         << "The address of i is " << &i << ".\n"
11         << "The size in bytes of i is " << sizeof i << ".\n\n";
12
13     int *p = &i;   //Let the value of p be the address of i.
14
15     cout << "The value of i " << *p << ".\n"
16         << "The address of i is " << p << ".\n"
17         << "The size in bytes of i is " << sizeof *p << ".\n\n"
18
19         << "The value of p is " << p << ".\n"
20         << "The address of p is " << &p << ".\n"
21         << "The size in bytes of p is " << sizeof p << ".\n";
22
23     return EXIT_SUCCESS;
24 }
```

You can split the above line 13 into

```
25     int *p;
26     p = &i;
```

But why would you want to allow `p` to momentarily hold garbage?

`The value of i is 10.`	*All platforms output integers in decimal.*
`The address of i is 0xffbff644.`	*My platform outputs addresses in hexadecimal.*
`The size in bytes of i is 4.`	*Ma be different on other platforms.*
`The value of i 10.`	
`The address of i is 0xffbff644.`	
`The size in bytes of i is 4.`	*Ma be different on other platforms.*
`The value of p is 0xffbff644.`	
`The address of p is 0xffbff640.`	
`The size in bytes of p is 4.`	*Ma be different on other platforms.*

Here is a diagram of `i` and `p` in memory, showing that the value of `p` is the address of `i`. There was room to show only the last two hex digits of the address of each byte. Although my platform put `p` immediately before `i` in memory, our convention will be to draw a pointer to the right of the variable to which it points; we'll see why when we talk about constant pointers and pointers to pointers on pp. 51 and 52. The address and size of the variable will be different on each platform. In particular, an integer and a pointer to an integer may be different sizes.

A pointer to an array element

Three more operators can be applied to a pointer that points to an element of an array: `++`, `--`, and `[]`.

Here are two ways to loop through an array. In lines 10−13, the variable `i` holds the subscript of each element of the array. To get the value of each element, the subscripting operator `[]` in line 12 has to do a lot of arithmetic. It multiplies the subscript `i` times the width of each element (the number of bytes in

an `int`.) Then it adds this product to the address of the first element of the array, yielding the address of the desired element. The element's value is then fetched from that address.

Is there a way to avoid this hidden multiplication and addition each time around the `for` loop? In lines 17−20, the pointer `p` holds the address of each element of the array. The dereferencing operator `*` in line 19 does not have to do any arithmetic at all: the address of the desired element is already sitting in `p`. This is one of the reasons why C was given pointers in 1970. Is it still relevant? When we write lines 10−13, some contemporary compilers are smart enough to behave as if we had written lines 17−20. Is your compiler one of them?

As usual, the `++p` in line 16 means `p = p + 1`. This makes `p` point at the next array element, not at the next byte. In low level terms, we say that an integer added to a pointer is implicitly multiplied by the number of bytes in the pointed-to variable. Since line 17 declared `p` to be a pointer to an `int`, the `++` adds `sizeof (int)` to the value of `p`.

Other examples of this multiplication are the expressions `a + i` and `a + n` in lines 11 and 17. The name of an array, unencumbered by a subscript, is a pointer to its first element. The integers `i` and `n` are therefore multiplied by the number of bytes in that element. The address `a + n`, for example, would be `n` × `sizeof (int)` bytes from the start of the array. This is the address of the (non-existent) first element beyond the end of the array.

The subtraction in line 18 yields the distance from the first element to the element that `p` is pointing to. This distance is measured in array elements, not in bytes; it is therefore the subscript of the element that `p` is pointing to. In low level terms, we say that an the difference of two pointers is implicitly divided by the number of bytes in the pointed-to variables. The subtraction will compile only when both pointers are declared to point to the same type of variable.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer/array.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     int a[] = {10, 20, 30};
 8     const size_t n = sizeof a / sizeof a[0];
 9
10     for (size_t i = 0; i < n; ++i) {
11         cout << "a[" << i << "]: address is " << a + i
12             << ", value is " << a[i] << ".\n";
13     }
14
15     cout << "\n";
16
17     for (int *p = a; p < a + n; ++p) {
18         cout << "a[" << p - a << "]: address is " << p
19             << ", value is " << *p << ".\n";
20     }
21
22     return EXIT_SUCCESS;
23 }
```

```
a[0]: address is 0xffbff630, value is 10.    the start of the array
a[1]: address is 0xffbff634, value is 20.    sizeof (int) bytes from start of array
a[2]: address is 0xffbff638, value is 30.    2 * sizeof (int) bytes from start of array

a[0]: address is 0xffbff630, value is 10.
a[1]: address is 0xffbff634, value is 20.
a[2]: address is 0xffbff638, value is 30.
```

Any pointer `p` can access the variable to which it is pointing: simply apply the dereferencing operator `*p`. A pointer to an array element can also access the neighboring elements with the subscripting operator `[]`. `p[0]` is another way to say `*p`, the element to which `p` is pointing. `p[1]`, `p[2]`, `p[3]`, and `p[-1]`, `p[-2]`, `p[-3]`, are the neighbors in each direction. Be careful not to go beyond the ends of the array.

The following program uses this notation to sort an array of integers into ascending order. The strategy is called *bubble sort*. Line 11 initializes `p` to point to the first element of the array. The first time we execute line 12, `p[0]` and `p[1]` are therefore the first two elements. Since we want to sort the elements into ascending order, we hope that `p[0]` is less than or equal to `p[1]`. If this is not the case, lines 13–15 swap the values of the two elements.

The second time we execute line 12, `p` points at the second element of the array. This time, `p[0]` and `p[1]` are the second and third elements. The third time we execute line 12, `p` points at the third element; `p[0]` and `p[1]` are the third and fourth elements. Line 12 makes it look like we had a little portable array named `p` that we could superimpose on each pair of elements. It is a considerable notational convenience.

When the `for` loop in line 11 exhausts itself, the largest number in the array has been borne along to the last element. The other numbers, however, may still be in disarray, so we have to go back to the beginning of the array and start again. That's why the loop in line 11 is enclosed in the larger loop in line 10; it decrements `end` so we don't go all the way to the last element again.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer/sort.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     int a[] = {20, 10, 50, 30, 40};
 8     const size_t n = sizeof a / sizeof a[0];
 9
10     for (int *end = a + n - 1; a < end; --end) {
11         for (int *p = a; p < end; ++p) {
12             if (p[1] < p[0]) {   //if p[0] and p[1] in wrong order,
13                 const int temp = p[0];   //swap them
14                 p[0] = p[1];
15                 p[1] = temp;
16             }
17         }
18     }
19
20     for (int *p = a; p < a + n; ++p) {
21         cout << *p << "\n";
22     }
23
24     return EXIT_SUCCESS;
25 }
```

```
10
20
30
40
50
```

If C++ did not have the subscripting operator, the above line 12 would have to be

```
26          if (*(p + 1) < *p) {
```

The expression `p + 1` is the address of the element after the one to which `p` is pointing; the expression `*(p + 1)` is the value of that element. We would need parentheses to execute the `+` before the `*`.

But C++ does have the subscripting operator, so do not write the above line 26. The `p[1]` in line 12 is a simpler way to do the same thing. The operator `[]` does the work of the `+` and `*`. And now that the two operators are gone, we no longer need the parentheses to make them execute in the correct order. (At the end of line 12, I wrote `p[0]` instead of `*p` for stylistic consistency with the expression `p[1]`.)

The same rule applies to an array and a subscript: always write `a[i]` instead of `*(a+i)`. The name of an array is a pointer to the first element of the array, and a subscript can be applied to any pointer to an array element.

We will turn this function into an ‘‘algorithm’’ on pp. 761−763.

A pointer to a structure

One more operator can be applied to a pointer that points to a structure: the operator `->`.

Line 15 creates a structure. A C program would need to say `struct str` here (p. 30, line 43) but we need only the `str`. Line 16 accesses the structure fields with the `.` operator.

Line 18 creates a pointer to the structure. The dereferencing operator `*` can be applied to any pointer. But the structure is not an element of an array, so the operators `++`, `--`, or `[]` cannot be applied to this pointer.

Let’s walk through the order in which the subexpressions of the `(*p).i` in line 20 are executed. We use the pointer `p` by applying the `*` operator to it, retrieving to the pointed-to variable. In this case, the variable turns out to be a structure. We use a structure by applying the dot operator which we saw in line 16. The `*` operator must therefore be applied before the dot. Since the `*` has lower precedence, we need the parentheses to make the `*` go first. See p. 112 for a similar sequence of subexpressions.

But never write the expression `(*p).i`. The single operator `->` in the expression `p->i` will do all the work of the two operators in `(*p).i`. And now that the two operators are gone, we no longer need the parentheses to make them execute in the correct order.

Line 23 passes `p` to a function. We will see on pp. 111−112 that C++ gives us a better notation for this.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer/struct.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 struct str {
 6     int i;
 7     double d;
 8 };
 9
10 void f(str *p);
11
12 int main()
```

```
13 {
14
15     str s = {10, 3.14};   //don't need the keyword "struct"
16     cout << s.i << " " << s.d << "\n";
17
18     str *p = &s;          //Let the value of p be the address of s.
19
20     cout << (*p).i << " " << (*p).d << "\n"
21         << p->i << " " << p->d << "\n";
22
23     f(p);
24
25     return EXIT_SUCCESS;
26 }
27
28 void f(str *p)
29 {
30     cout << p->i << " " << p->d << "\n";
31 }
```

```
10 3.14
10 3.14
10 3.14
10 3.14
```

▼ Homework 1.6.1a: two ways to simplify the same expression

The following program has an array of structures. The expression `(*(a+2)).d` in line 19 is the field named `d` of the structure at subscript 2. It can be simplified in two ways.

(1) change the `+` and `*` to the operator `[]`

(2) change the `*` and dot to the operator `->`

Try both. Which way lets us get rid of the most parentheses? Which way yields the simpler result?

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer/simplify.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 struct str {
 6     int i;
 7     double d;
 8 };
 9
10 int main()
11 {
12     str a[] = {
13         {10, 15.0},
14         {20, 25.0},
15         {30, 35.0}
16     };
17     const size_t n = sizeof a / sizeof a[0];
18
```

```
19      cout << (*(a + 2)).d << "\n";
20
21      return EXIT_SUCCESS;
22  }
```

```
35
```

▲

1.6.2 Constants, Constant Pointers, and `const_cast`

Constants

A variable whose value is not supposed to change should be declared `const`. This designation will make the program easier to understand and debug.

A constant cannot be assigned to (line 2). It must therefore be initialized at its moment of birth (line 1).

```
1       const int i = 10;
2       i = 20;             //won't compile: can't assign to a const
3
4       const int j;        //won't compile: a const must be initialized
```

Please use a `const` variable instead of a macro. The rules for parentheses are simpler for a variable.

```
5  #define TAX (610 + 395 + 15)        //macro needs parentheses
6  const int tax = 610 + 395 + 15;     //const doesn't need parentheses
```

And a variable can be made local to a function or other block.

```
7  void f()
8  {
9  #define TAX (610 + 395 + 15)
10      const int tax = 610 + 395 + 15;
11 }   //The variable tax disappears at this point.
12
13 void g()
14 {
15      //The macro TAX still exists at this point,
16      //but the variable tax does not.
17 }
```

A pointer can be constant in two different ways.

Constants that are pointers are twice as complicated as plain old constants; those that are pointers to pointers are four times as complicated. But don't worry: we will never go beyond two levels. Pointers to pointers to pointers are rarely used.

The variable to which a pointer points will be called the *target variable.* The strings in lines 1−2 will be our target variables. The diagram shows the first string at address 1000 and the first pointer at address 2000, but these numbers will be different on each platform.

Lines 4−8 show what a pointer can do. Line 5 reads the target variable; line 6 writes it. Since our pointer can do both, it is a *read/write* pointer. Line 7 shows that a pointer can access not only the target variable, but also the target variable's neighbors. Line 8 shows that a pointer can be given a new value. It now points to a different target.

We now demonstrate the two different types of constant pointers. In the name of a data type, the keyword `const` may appear at the beginning (lines 10 and 22) or immediately after any asterisk (lines 16 and 22).

In the name of a pointer data type, a `const` at the beginning of the name means that the pointer cannot write its target variable. In this case we say that the pointer is *read-only.* For example, the pointer `p1` in line 10 can be used to read the target variable in line 11, but not to write the target variable or its neighbors in lines 12 and 13. It can, however, still have its value changed in line 14. It now points to a different target. (''Read-only'' refers not to the value of the pointer, but to the what the pointer can do to the value its target variable.)

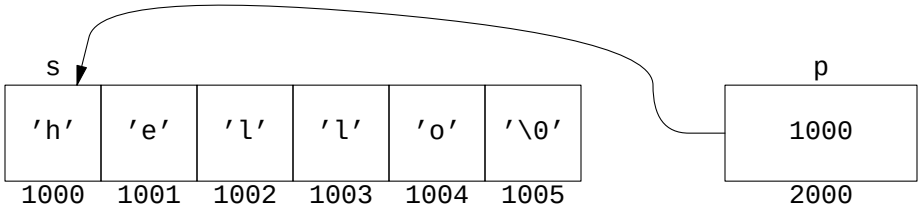
Note that a ''pointer to a function'' cannot be `const` in this sense. A function cannot be overwritten.

In the name of a pointer data type, a `const` immediately after an asterisk means that the pointer cannot be given a new value. It therefore always points to the same target. For example, the pointer `p2` in line 16 always points to the target `s`. It can, however, still read and write the target in lines 17−19.

We have seen the two positions where the keyword `const` can be inserted in the name of a pointer data type. The two areas of memory in the following diagram correspond to these two positions. A `const` in the left position means that we cannot use the pointer to change the value of the target variable (the data in the left part of the diagram). A `const` in the right position means that we cannot change the value of the pointer (the data in the right part of the diagram).

A pointer can be made `const` in both ways simultaneously (line 22).

Even with all these various types of constant pointers aimed at it, the target `s` is still not constant. Line 29 demonstrates that `s` can be changed very easily. We just cannot change `s` by means of the read-only pointers `p1` and `p3`.




```
 1     char s[] = "hello";
 2     char t[] = "goodbye";
 3
 4     char *p = s;              //s means &s[0]
 5     cout << *p << "\n";       //Output the 'h'.
 6     *p = 'H';                 //Change the 'h' to 'H'.
 7     p[1] = 'E';               //Change the 'e' to 'E'.
 8     p = t;                    //Make p point to t.
 9
10     const char *p1 = s;       //p1 gives read-only access to s
11     cout << *p1 << "\n";
12     *p1 = 'H';                //won't compile: can't use p1 to change s[0]
13     p1[1] = 'E';              //won't compile: can't use p1 to change s[1]
14     p1 = t;                   //okay
15
16     char *const p2 = s;       //p2 must be initialized
17     cout << *p2 << "\n";
18     *p2 = 'H';                //okay
19     p2[1] = 'E';              //okay
20     p2 = t;                   //won't compile: can't make p2 point away from s
21
22     const char *const p3 = s; //p3 must be initialized because of 2nd const
```

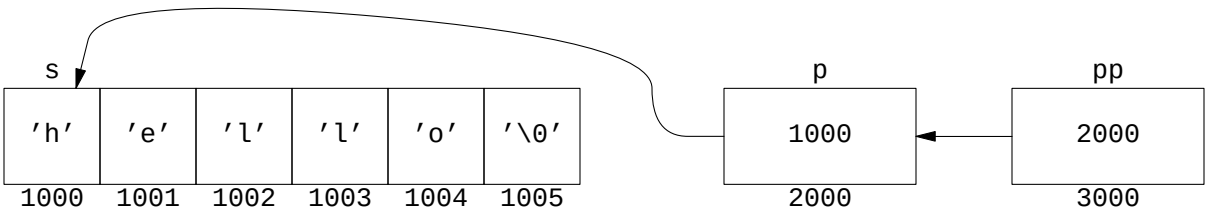
```
23                                  //in line 22
24      cout << *p3 << "\n";
25      *p3 = 'H';                  //won't compile because of 1st const in line 22
26      p3[1] = 'E';                //won't compile because of 1st const in line 22
27      p3 = t;                     //won't compile because of 2nd const in line 22
28
29      s[0] = 'H';                 //okay, even with all the const pointers
30                                  //pointing at s[0].
```

A pointer to a pointer can be constant in three different ways.

The strings in lines 1 and 2 will be the target variables for our pointers `p` and `q`. These pointers are themselves targets for the pointer to pointer `pp`. Let's say that `p` and `q` will be the *immediate targets* of `pp`, and `s` and `t` will be the *ultimate targets* of `pp`.

Lines 8−10 show what a pointer to pointer can do with differing numbers of asterisks. We access three areas of memory, corresponding to the three areas in the following diagram. Line 8 writes into the ultimate target of `pp` (the data in the left part of the diagram); line 9 writes into the immediate target of `pp` (the data in the middle of the diagram); line 10 writes into `pp` itself (the data in the right part of the diagram).

Once again, we can insert the keyword `const` at the start of the name of a data type, or immediately after any asterisk. Thus there are three positions where we can insert the keyword into the name of the type of a pointer to pointer. The three areas of memory in the following diagram correspond to these three positions. A `const` in the left position (lines 12−13) means that the pointer to pointer cannot be used to change the value of the ultimate target (the data in the left part of the diagram). A `const` in the middle position (lines 15−16) means that the pointer to pointer cannot be used to change the value of the immediate target (the data in the left part of the diagram). A `const` in the right position (lines 18−19) means that we cannot change the value of the pointer to pointer itself (the data in the right part of the diagram).




```
 1      char s[] = "hello";
 2      char t[] = "goodbye";
 3
 4      char *p = s;                //p points to the string s.
 5      char *q = t;
 6
 7      char **pp = &p;             //pp points to the pointer p in line 4.
 8      **pp = 'H';                 //Change the 'h' to 'H'.
 9       *pp = t;                   //Make p point to a different string.
10        pp = &q;                  //Make pp point to a different pointer.
11
12      const char **pp1 = &p;
13      **pp1 = 'H';                //won't compile: can't use pp1 to change s[0].
14
15      char *const *pp2 = &p;
16      *pp2 = t;                   //won't compile: can't use pp2 to change p.
17
18      char **const pp3 = &p;      //pp3 must be initialized
19      pp3 = &q;                   //won't compile: can't make pp3 point away from p
```

A realistic example

The following program starts in English mode (line 19), but we can change it to Spanish in line 26.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/pointer/language.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     const char *const english[] = {          //constant in 2 out of 2 ways
 8         "Monday",
 9         "Tuesday",
10         "Wednesday"
11     };
12
13     const char *const spanish[] = {
14         "Lunes",
15         "Martes",
16         "Miercoles"
17     };
18
19     const char *const *language = english;   //constant in 2 out of 3 ways
20
21     cout << "The first days of the week are " << language[0]
22         << " and " << language[1] << ".\n"
23         << "The first characters of the first day are "
24         << language[0][0] << " and " << language[0][1] << ".\n\n";
25
26     language = spanish;
27
28     cout << "The first days of the week are " << language[0]
29         << " and " << language[1] << ".\n"
30         << "The first characters of the first day are "
31         << language[0][0] << " and " << language[0][1] << ".\n";
32
33     return EXIT_SUCCESS;
34 }
```

The array in the above line 7 contains three pointers. Because of the right `const` in line 7, the pointers always point to the same place. This prevents the following from compiling.

```
35     //Try to make the first pointer in the array point somewhere else.
36     english[0] = "Bloomsday";
```

Because of the left `const` in line 7, the pointers give us read-only access to the characters to which they point. This prevents the following from compiling.

```
37     //Try to use the pointer english[0]
38     //to change the 'M' in "Monday" to lowercase.
39     english[0][0] = 'm';
```

The variable `language` in line 19 is a pointer to a pointer, like the `argv` in line 4 of `sum.c` on p. 33. It points to the first element of one or the other array. This first element is a pointer, which is why `language` has to be a pointer to a pointer.

Lines 21−24 demonstrate how `language` can be used. Since it is a pointer to a pointer, it can be dereferenced once or twice. Dereferencing it once, with the application of a leading `*` or a trailing subscript, will access a pointer in one of the arrays. Think of this as a journey in the above diagram starting at `pp` and following one arrow; we land at a pointer to a character. Dereferencing it twice, with two applications of these operators, will access a character in one of the strings. Think of this as a journey starting at `pp` and following both arrows; we land at a character.

Line 26 demonstrates that `language` can be pointed at a different variable. It will compile because of the absence of a `const` after the right asterisk in line 19. But `language` cannot be used to damage the pointers in the arrays or the characters to which they point. Because of the `const` after the left asterisk in line 19, `language` is read-only with respect to its immediate target, which is a pointer in one of the arrays. Line 42 will not compile.

```
40      //Try to make the first pointer in one of the arrays
41      //point somewhere else.
42      language[0] = "Bloomsday";
```

And because of the `const` at the start of line 19, `language` is also read-only with respect to its ultimate target, which is a character in one of the strings. Line 44 will not compile.

```
43      //Try to change the 'M' in "Monday" or 'L' in "Lunes".
44      language[0][0] = 'A';
```

```
The first days of the week are Monday and Tuesday.
The first characters of the first day are M and o.

The first days of the week are Lunes and Martes.
The first characters of the first day are L and u.
```

To get the above line 39 to compile, we would have to remove the left `const` from line 7. To get line 44 to compile, we would also have to remove the left `const` from 19. But to get lines 39 and 44 to execute, we have to do even more.

A double-quoted string literal is an array of constants. For example, the `"Monday"` in the above line 8 is of type `const char[7]`. Thanks to a loophole in C++, a non-const pointer can to point at a string literal. This is why line 7 will still compile even without the left `const`. But if we try to use such a pointer to overwrite the characters, we get undefined behavior.

```
45      char *const p = "Monday";       //legal but deprecated
46      p[0] = 'm';                     //undefined behavior
```

To execute lines 39 and 44 correctly, the character arrays must be changed from arrays of `const char` to arrays of `char`. For example,

```
47      char monday[] = "Monday";       //The array monday can be overwritten.
48      char tuesday[] = "Tuesday";
49      char wednesday[] = "Wednesday";
50
51      char *const english[] = {
52          monday,
53          tuesday,
54          wednesday
55      };
```

Subvert a read-only pointer

The pointer in line 2 gives us read-only access to the target variable in line 1. If line 4 tries to use the pointer to write the target variable, it will not compile.

The amateur knows how to follow the rules; but the professional knows how to break the rules. With due caution, the professional can use the pointer as if were not `const` by writing the `const_cast` in line 5. This kind of cast temporarily removes the ‘‘read-onlyness’’ from a pointer. (See pp. 63−65 for the other kinds of cast.) It can be applied to a read-only pointer, but only if the target variable is not `const`. If the target is `const`, the program will still compile but its behavior will be undefined. If you’re lucky, it will blow up.

```
1     char s[] = "hello";             //target variable is not const
2     const char *p = s;              //p normally gives read-only access to s
3
4     *p = 'H';                       //won't compile: try to change the 'h' to 'H'
5     *const_cast<char *>(p) = 'H';   //will compile: change the 'h' to 'H'
```

Who can point to a const variable?

Only a read-only pointer can point to a `const` object:

```
1 const char s[] = "hello";
2
3 const char *p = s;      //will compile: p is a read-only pointer
4 char *q = s;            //won't compile: q is a read/write pointer
```

If the above line 4 were legal, the user could then say

```
5     *q = 'H';
```

circumventing the `const` in line 1.

1.6.3 Type Conversion, including Pointer Conversion

Unsigned integers

An 8-bit integer can hold $256 = 2^8$ different values. Now which 256 values should they be?

An integer that we use to hold only non-negative values, starting at zero and working upwards, is called an *unsigned integer.* An 8-bit unsigned integer, for example, can count from 0 to 255 inclusive.

binary	*decimal*	
11111111	255	$= 2^8 - 1$
11111110	254	
11111101	253	
11111100	252	
.		
.		
.		
00000011	3	
00000010	2	
00000001	1	
00000000	0	

A 16-bit unsigned integer can count from 0 to 65,535 inclusive.

binary	*decimal*	
1111111111111111	65,535	$= 2^{16} - 1$
1111111111111110	65,534	
1111111111111101	65,533	
1111111111111100	65,532	
.		
.		
.		
0000000000000011	3	
0000000000000010	2	
0000000000000001	1	
0000000000000000	0	

A 32-bit unsigned integer can count from 0 to 4,294,967,295 inclusive.

binary	*decimal*	
11111111111111111111111111111111	4,294,967,295	$= 2^{32} - 1$
11111111111111111111111111111110	4,294,967,294	
11111111111111111111111111111101	4,294,967,293	
11111111111111111111111111111100	4,294,967,292	
.		
.		
.		
00000000000000000000000000000011	3	
00000000000000000000000000000010	2	
00000000000000000000000000000001	1	
00000000000000000000000000000000	0	

A 64-bit unsigned integer can count from 0 to 18,446,744,073,709,551,615 inclusive.

binary	*decimal*	
1111111111111111111111111111111111111111111111111111111111111111	18,446,744,073,709,551,615	$= 2^{64} - 1$
1111111111111111111111111111111111111111111111111111111111111110	18,446,744,073,709,551,614	
1111111111111111111111111111111111111111111111111111111111111101	18,446,744,073,709,551,613	
1111111111111111111111111111111111111111111111111111111111111100	18,446,744,073,709,551,612	
.		
.		
.		
0000000000000000000000000000000000000000000000000000000000000011	3	
0000000000000000000000000000000000000000000000000000000000000010	2	
0000000000000000000000000000000000000000000000000000000000000001	1	
0000000000000000000000000000000000000000000000000000000000000000	0	

▼ Homework 1.6.3a: do you have these data types?

Does your C++ have an 8-bit unsigned integer data type? It's probably the type `unsigned char`. Do you have a 16-bit one? It's probably the data type `unsigned short`.

To find out, use the `<climits>` macros. (They will be superseded by the `numeric_limits` "class" on pp. 745−747.)

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/cast/char_bit.C`

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <climits>   //for CHAR_BIT, UCHAR_MAX, USHRT_MAX
```

```
 4 using namespace std;
 5
 6 int main()
 7 {
 8     cout
 9         << "On my platform, a byte contains " << CHAR_BIT
10         << " bits.\n\n"
11
12         << "On every platform, a char is by definition 1 byte.\n"
13         << "On my platform, the largest number that an unsigned char"
14         << " can hold is " << UCHAR_MAX << ".\n\n"
15
16         << "On my platform, an unsigned short contains "
17         << sizeof (unsigned short) << " bytes or "
18         << CHAR_BIT * sizeof (unsigned short) << " bits.\n"
19         << "The largest number that an unsigned short can hold is "
20         << USHRT_MAX << ".\n";
21
22     return EXIT_SUCCESS;
23 }
```

```
On my platform, a byte contains 8 bits.

On every platform, a char is by definition 1 byte.
On my platform, the largest number that an unsigned char can hold is 255.

On my platform, an unsigned short contains 2 bytes or 16 bits.
The largest number that an unsigned short can hold is 65535.
```

For 32 bits, try the data type `unsigned` with the macro `UINT_MAX`. For 64 bits, try `long unsigned` with the macro `ULONG_MAX`.

▲

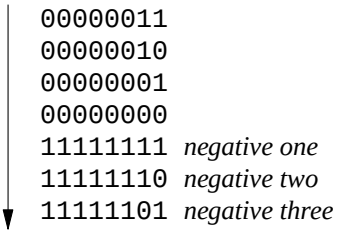
Signed integers and the two's complement representation

An integer that we use to hold negative and non-negative numbers is called a *signed integer.* A negative number is usually written with a negative sign (−1). But how can the negative be represented when the computer's memory holds only `1`'s and `0`'s?

Automobile manufacturers solved this problem a long time ago. In an odometer running backwards, `99999999` represents −1 and `99999998` represents −2.

```
00000003
00000002
00000001
00000000
99999999  negative one
99999998  negative two
99999997  negative three
```

In the binary world, the moral equivalents of `99999999` and `99999998` are `11111111` and `11111110`. This way of representing negative numbers is called the *two's complement* notation. We will assume our hardware uses it, although other representations are possible.



Does `11111111` mean 255 or −1? It depends on whether the 8-bit integer is unsigned or signed. The range of unsigned values starts at zero and goes up. The range of signed values is centered as nearly as possible around zero. For an 8-bit signed integer, it goes from −128 to 127 inclusive.

binary	*decimal*	
`01111111`	127	$= 2^7 - 1$
`01111110`	126	
`01111101`	125	
`01111100`	124	
.		
.		
.		
`00000011`	3	
`00000010`	2	
`00000001`	1	
`00000000`	0	
`11111111`	−1	
`11111110`	−2	
`11111101`	−3	
.		
.		
.		
`10000011`	−125	
`10000010`	−126	
`10000001`	−127	
`10000000`	−128	$= -2^7$

The two's complement notation gives us an easy way to tell if a signed integer is negative. The left-most bit will be `1` for a negative number, `0` otherwise. We call it the *sign bit.*

We actually had no choice when representing −1 as ''all ones''. Let's use 8-bit integers to show why. If we want zero to be `00000000` and 1 to be `00000001`, and if we want the sum of 1 and −1 to be zero, then the representation of −1 as ''all ones'' is forced upon us. No other bit pattern will give us a sum of zero.

```
  00000001   positive one
+ 11111111   negative one
----------
  00000000   zero
```

Similarly, if we want the sum of 2 and −2 to be zero, the representation of −2 as `11111110` is forced upon us.

```
  00000010   positive two
+ 11111110   negative two
----------
  00000000   zero
```

For a 16-bit signed integer, the range of values goes from −32,768 to 32,767 inclusive.

binary	*decimal*	
0000000001111111	32,767	$= 2^{15} - 1$
0000000001111110	32,766	
0000000001111101	32,765	
0000000001111100	32,764	
.		
.		
.		
0000000000000011	3	
0000000000000010	2	
0000000000000001	1	
0000000000000000	0	
1111111111111111	−1	
1111111111111110	−2	
1111111111111101	−3	
.		
.		
.		
1000000000000011	−32,765	
1000000000000010	−32,766	
1000000000000001	−32,767	
1000000000000000	−32,768	$= -2^{15}$

For a 32-bit signed integer, the range goes from goes from −2,147,483,648 to 2,147,483,647 inclusive.

binary	*decimal*	
01111111111111111111111111111111	2,147,483,647	$= 2^{31} - 1$
01111111111111111111111111111110	2,147,483,646	
01111111111111111111111111111101	2,147,483,645	
01111111111111111111111111111100	2,147,483,644	
.		
.		
.		
00000000000000000000000000000011	3	
00000000000000000000000000000010	2	
00000000000000000000000000000001	1	
00000000000000000000000000000000	0	
11111111111111111111111111111111	−1	
11111111111111111111111111111110	−2	
11111111111111111111111111111101	−3	
.		
.		
.		
10000000000000000000000000000011	−2,147,483,645	
10000000000000000000000000000010	−2,147,483,646	
10000000000000000000000000000001	−2,147,483,647	
10000000000000000000000000000000	−2,147,483,648	$= -2^{31}$

For a 64-bit signed integer, the range goes from goes from −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 inclusive.

binary	*decimal*
01111111111111111111111111111111111111111111111111111111111111111	9,223,372,036,854,775,807
01111111111111111111111111111111111111111111111111111111111111110	9,223,372,036,854,775,806
01111111111111111111111111111111111111111111111111111111111111101	9,223,372,036,854,775,805
01111111111111111111111111111111111111111111111111111111111111100	9,223,372,036,854,775,804
.	
.	
.	
00000000000000000000000000000000000000000000000000000000000000011	3
00000000000000000000000000000000000000000000000000000000000000010	2
00000000000000000000000000000000000000000000000000000000000000001	1
00000000000000000000000000000000000000000000000000000000000000000	0
11111111111111111111111111111111111111111111111111111111111111111	−1
11111111111111111111111111111111111111111111111111111111111111110	−2
11111111111111111111111111111111111111111111111111111111111111101	−3
.	
.	
.	
10000000000000000000000000000000000000000000000000000000000000011	−9,223,372,036,854,775,805
10000000000000000000000000000000000000000000000000000000000000010	−9,223,372,036,854,775,806
10000000000000000000000000000000000000000000000000000000000000001	−9,223,372,036,854,775,807
10000000000000000000000000000000000000000000000000000000000000000	−9,223,372,036,854,775,808

▼ Homework 1.6.3b: do you have these data types?

Does your C++ have signed integers of 8, 16, 32, and 64 bits? The macro `CHAR_BIT` is the number of bits in any type of `char`, signed or unsigned. Each signed data type has a pair of macros, giving the minimum and maximum values for that type.

data type	*minimum*	*maximum*
`signed char`	`SCHAR_MIN`	`SCHAR_MAX`
`int`	`INT_MIN`	`INT_MAX`
`long`	`LONG_MIN`	`LONG_MAX`

▲

▼ Homework 1.6.3c: infinite loops

Why do the following loops iterate forever? Fix them.

```
 1     //Should iterate 11 times.
 2     for (unsigned char c = 10; c >= 0; --c) {
 3         //do something
 4     }

 5     //Should iterate UCHAR_MAX + 1 times.
 6     for (unsigned char c = 0; c <= UCHAR_MAX; ++c) {
 7         //do something
 8     }
```

The problem is not limited to the unsigned data types:

```
 9     //Should iterate CHAR_MAX - CHAR_MIN + 1 times.
10     for (char c = CHAR_MIN; c <= CHAR_MAX; ++c) {
11         //do something
12     }
```

▲

Integral conversions

The *integral* data types are `int` (signed and unsigned, long and short), `char` (signed, unsigned, and neither), `wchar_t`, and `bool`. They are the built-in data types that are neither fractional nor pointers. Enumerations are integral in C but not in C++, since they will take ''overloaded operators'' in C++. In the following rules, however, enumerations will behave as signed integers.

When converting one data type to another, the values are called the *source* and *destination.* If the source and destination types are both integral, the conversion is called an *integral conversion.* An integral conversion from a narrower source type to a wider destination type is called a *widening.*

(1) In an integral conversion that is a widening, the result is determined by the source type. First, the bit pattern of the source is copied into the rightmost bits of the destination. Then if the source is unsigned, the new bits will be all zeroes. We say that an unsigned source is *zero-extended.* This ensures that a non-negative value will remain non-negative. Here is a widening from 8 to 32 bits.

```
                        11111111    unsigned source is 255
00000000000000000000000011111111    destination, whether signed or unsigned, is 255
```

If the source is signed, the new bits will be copies of the source's sign bit. We say that a signed source is *sign-extended,* and that its sign bit is *propagated.* This ensures that a non-negative value will remain non-negative, and a negative value will remain negative if it possibly can (i.e., if the destination is signed).

```
                        01111111    signed source is 127
00000000000000000000000001111111    destination, whether signed or unsigned, is 127

                        11111111    signed source is −1
11111111111111111111111111111111    see below
```

The last destination above will represent −1 if signed, $4,294,967,295 = 2^{32} - 1$ if unsigned. (Mathematicians can take comfort that these two possibilities, seemingly so different, are actually congruent mod 2^{32}.)

(2) In an integral conversion that is not a widening, the result is determined by the destination type. A source that is too big or too small to fit in a signed destination yields an ''implementation defined'' result. Otherwise, the bit pattern of the source, or as much of it as will fit, is simply copied into the destination. Here are two conversions from 32 to 32 bits.

```
11111111111111111111111111111111    signed source is −1
11111111111111111111111111111111    unsigned destination is 4,294,967,295 = 2^32 − 1

11111111111111111111111111111111    unsigned source is 4,294,967,295 = 2^32 − 1
????????????????????????????????    signed destination is implementation defined
```

The last destination above is implementation defined because a signed 32-bit integer cannot hold values beyond $2,147,483,647 = 2^{31} - 1$.

Here are four conversions from 32 to 8 bits.

```
11111111111111111111111111111111    unsigned source is 4,294,967,295 = 2^32 − 1
                        11111111    unsigned destination is 255 = 2^8 − 1

11111111111111111111111111111111    signed source is −1
                        11111111    signed destination is −1

11111111111111111111111111111111    signed source is −1
                        11111111    unsigned destination is 255 = 2^8 − 1
```

```
11111111111111111111111111111111      unsigned source is 4,294,967,295 = 2^32 − 1
                        ????????      signed destination is implementation defined
```

The last destination above is implementation defined because a signed 8-bit integer cannot hold values beyond $127 = 2^7 - 1$.

Implicit type conversion

Four kinds of conversion can be performed simply by assigning a value of one type to a variable of another type. We call them *implicit* conversions, since they can be written without a cast. The last three are pointer conversions; another pointer conversion will be on p. 486.

(1) Convert between the built-in data types (p. 27). The conversion in line 4 avoids integer division in line 5 and the resulting truncation.

```
1     int dividend = 22;
2     int divisor = 7;
3
4     double d = dividend;      //convert int to double
5     double quotient = d / divisor;
```

(2) Convert the `int` value `0` to any pointer type.

```
6     int *p1 = 0;              //convert 0 to "pointer to int"
```

(3) Convert any pointer type to `bool`. The result will be `false` if the pointer is zero, `true` otherwise.

```
7     int i = 10;
8     int *p = &i;
9     bool b = p;               //convert "pointer to int" to bool
```

We say that a data type is *convertible to* `bool` if it can be implicitly converted to `bool`. This convertibility makes it easy to write a statement that checks if a pointer is zero. The expression in the parentheses of an `if`, `while`, or `for` is always converted to a `bool` if it is not one already. We can therefore use a pointer in this context. The C tests in lines 14 and 17 will continue to work in C++.

```
10 #include <stdio.h>           /* for fopen: C example */
11 #include <stdlib.h>          /* for malloc */
12
13    FILE *const fp = fopen("filename", "r");
14    if (fp) { /* convert "pointer to FILE" to bool; true if fopen succeeded */
15
16    char *const p = malloc(10);
17    if (p) {  /* convert "pointer to char" to bool; true if malloc succeeded */
```

(4) Convert any ‘‘pointer to variable’’ to ‘‘pointer to `void *`’’.

```
18    int i = 10;
19    int *p = &i
20    void *q = p;              //convert "pointer to int" to "pointer to void"
```

A ‘‘pointer to function’’ cannot be converted implicitly to a ‘‘pointer to `void *`’’.

```
21    void f();                 //function declaration
22    void (*p)() = f;          //pointer to the function
23    //void *q = p;            //won't compile
```

A new notation for a cast

The above conversions are usually performed more easily by casting than by assignment. C++ has six different notations for casts. Here we will endorse two and reject two; elsewhere we will cover `const_cast` (pp. 54–55) and `dynamic_cast` (pp. 1012–1015).

The C cast in line 4 avoids integer division and the resulting truncation. The C++ cast in line 5 does the same thing. The expression to be converted must be in `(`parentheses`)` and the desired data type in `<`angle brackets`>`. This punctuation is borrowed from the ‘‘explicit template argument’’ on p. 660.

To find every cast in the program, we can now search for the conspicuous word `static_cast`. This is one advantage of the C++ cast over the C cast.

```
1     int dividend = 22;
2     int divisor = 7;
3
4     double quotient = (double)dividend / divisor;                /* C */
5     double quotient = static_cast<double>(dividend) / divisor;   //C++
```

Explicit type conversion

There is one conversion that could be implicit in C but must be explicit in C++. C can convert a ‘‘pointer to `void`’’ to any other type of pointer without a cast; see pp. 68–69. For example, line 3 converts the return value of `malloc` from ‘‘pointer to `void`’’ to ‘‘pointer to `char`’’.

```
1 #include <stdlib.h>     /* for malloc: C example */
2
3     char *p = malloc(10);   /* convert "pointer to void" to "pointer to char" */
4     if (p) {
```

But in C++ a `static_cast` is needed to convert a ‘‘pointer to `void`’’ to a ‘‘pointer to variable’’. (We will get rid of the `malloc` entirely when we do `new` on p. 394.)

```
5 #include <cstdlib>      //for malloc: C++ example
6
7     char *p = static_cast<char *>(malloc(10));
8     if (p) {
```

A `static_cast` that prevents a program from crashing

Lines 10 and 14 of the following program are realistic examples of `static_cast`. We output a character if it is printable, or its character code (probably an ASCII code) if it is not.

Line 8 of the following `static_cast.C` deliberately puts a non-printable bit pattern, ‘‘all ones’’, into the variable `c`. Writing the pattern as `0xFF` or `255` would have given us ‘‘all ones’’ only on platforms where the `char` was no wider than 8 bits. Writing the pattern as `0xFFFF` would have put an implementation defined result into the `char` on platforms where the data type `char` is signed and `0xFFFF` is outside the range of values for a `char`. Instead, we took an `int` zero, guaranteed to be at least as wide as a `char`, and made a photographic negative with the ‘‘bitwise not’’ operator. (We could have written `-1`, but the intent of this `char` is to hold a pattern of bits rather than a signed integer.)

Since a `char` is signed on some platforms, we must take care when passing it to the function `isprint` in the following line 10. This function belongs to the C Standard Library (declared in `<ctype.h>`), and therefore also to the C++ Standard Library (declared in `<cctype>`). The argument of `isprint` is an `int`; a narrower value will be implicitly widened.

`isprint` returns a non-zero `int` if the character is printable, zero otherwise. It would make more sense for `isprint` to return `bool`. But `isprint` was originally written in C, and C has no `bool`. Without the `!= 0` in line 10, the `int` return value of `isprint` would be implicitly converted to `bool`, causing a warning message on some compilers.

```
1 //Excerpt from the header file <cctype>
```

```
2 //(or from another header file included therein).
3
4 int isprint(int c);
```

Although a negative character code might be printable (pp. 1032−1034), we must never give a negative argument to `isprint`. This function is expected to work only if its argument is the end-of-file indicator `EOF` (probably −1) or a value in the range of an `unsigned char`. Any other argument could legally crash the program. `isprint` was given this latitude so that it could be implemented by the following array lookup.

`isprint` verifies that its argument is not `EOF` and then looks it up in an array. The array has one element for each value in the range of an `unsigned char`. The value of each element is non-zero if the character is printable, zero otherwise. We can now see why the argument of `isprint` must be `EOF` or a value in the range of an `unsigned char`. Any other value would be an out-of-range subscript in line 7 and could crash the program. (The `&&` operator will evaluate its right operand only if its left operand is true; see pp. 13−14.)

```
5 int isprint(int c)
6 {
7     static const int a[] = {0, 0, 0, /* etc. */ };
8     return c != EOF && a[c];
9 }
```

Let's assume that the data type `char` is an 8-bit signed `int` on our platform. If the following line 9 gave `c` directly to `isprint`,

```
10     if (isprint(c) != 0) {
```

`c` would be sign extended to `int`, resulting in a value of −1.

```
                                    11111111
11111111111111111111111111111111
```

If `EOF` were −1, `isprint` would mistake this argument for `EOF`. Even worse, if `EOF` were not −1, this argument would be outside the range of values for an `unsigned char` and could crash the program in the above line 7.

By casting `c` to `unsigned char`, line 10 ensures that it will be zero extended to `int`. The resulting value is in the range of an `unsigned char` and is a legal argument for `isprint`.

```
                                    11111111
00000000000000000000000011111111
```

The expression `c` in line 11 is of data type `char`, so the preceding `<<` displays it as a character. But we don't want to attempt this in lines 13−17, where the value of `c` is known to be unprintable. Instead, line 14 will output the character code of `c` as a number in the range of an `unsigned char`.

To do this requires two casts. The entire expression in line 14 is of data type `unsigned`, so the preceding `<<` will display it as a non-negative integer in decimal. First, however, we must cast `c` to `unsigned char` to ensure that it will be zero extended when widened to `unsigned`. (The same double cast will appear in line 40 of `terminal.C` on p. 161.) Lines 15 and 16 show what goes wrong if only one cast is used; we will have to remember this on pp. 877 and 892.

Line 17 is a fast and dirty way of printing a character code as a non-negative integer. It trims away all but the bottom 8 bits of `c` with "bitwise and". (The parentheses are required because the precedence of `&` is lower than `<<`; see pp. 24−25.) But the bit pattern `0xFF` works only if a `char` is exactly eight bits, which is precisely the kind of assumption that should not be embedded in our code. Please program with data types (line 14) not arithmetic (line 17).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/cast/static_cast.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cctype>   //for isprint
 4 using namespace std;
 5
 6 int main()
 7 {
 8     char c = ~0;   //0 is all zeroes; ~0 is all ones.
 9
10     if (isprint(static_cast<unsigned char>(c)) != 0) {
11         cout << "The character is '" << c << "'.\n";
12     } else {
13         cout << "The character code is "
14             << static_cast<unsigned>(static_cast<unsigned char>(c)) << ".\n"
15             << static_cast<unsigned>(c) << "\n"
16             << static_cast<int>(c) << "\n"
17             << (c & 0xFF) << "\n";
18     }
19
20     return EXIT_SUCCESS;
21 }
```

```
The character code is 255.      line 14: 2^8 − 1
4294967295                      line 15: 2^32 − 1
-1                              line 16
255                             line 17
```

Why not avoid the whole problem of sign extension by declaring all our character variables to be `unsigned char`? Well, many of the standard library functions expect arguments that are pointers to plain old `char`.

```
1 //Excerpts from <string.h> in C, <cstring> in C++
2
3 size_t strlen(const char *);
4 char *strcpy(char *dest, const char *source);
```

We can implicitly convert from `unsigned char` to `char`, but not from "pointer to `unsigned char`" to "pointer to `char`". As we are about to see, even an explicit `static_cast` cannot perform this conversion.

Dangerous conversions with `reinterpret_cast`

Certain pointer casts must be marked with a different keyword because they are so dangerous. We write `reinterpret_cast` instead of `static_cast` in the following three situations.

(1) We need a `reinterpret_cast` to convert between pointers to different types of variables. Line 11 of the following program converts the expression `a` from "pointer to `unsigned char`" to "pointer to `char`" (see p. 81 for another way to do this). Line 14 converts the expression `&s` from "pointer to `short`" to "pointer to `char`"; it then dereferences the latter to access the first `char` of the `short`. This tells us the order of the bytes within the `short`, which is of concern when we do networking. The Internet expects to receive the bytes of a `short` in big-endian order.

(2) We need a `reinterpret_cast` to convert between pointers to different types of functions. The `f` in line 2 is a pointer to a function that returns `void`; the `p` is a pointer to a function that returns `int`.

```
1     void f();
2     int (*p)() = reinterpret_cast<int (*)()>(f);
```

(3) We saw on p. 62 that a pointer to a variable can be implicitly converted to a `bool`. But all other conversions between a pointer and a non-pointer require a `reinterpret_cast`.

```
3 #include <cstddef>      //for size_t
4
5     int i = 10;
6     int *p = &i
7
8     size_t s = reinterpret_cast<int>(p);   //convert pointer to non-pointer
9     p = reinterpret_cast<int *>(s);        //convert non-pointer to pointer
```

The data type `size_t` should be used for any variable that holds an array subscript, or the number of elements in an array, or the number of bytes in a block of memory. `size_t` is therefore the data type of the return value of the C function `strlen`, the argument of the C function `malloc`, and the value of the `sizeof` operator. It ought to be big enough to hold the value of a pointer.

Do not use the data type `int` for these purposes: it might not be big enough. (The one exception is the `argv` array, which was invented before `size_t`. For this array, the number of elements (`argc`) has always been an `int`.)

`size_t` is another name for `unsigned` or `unsigned long`, depending on the hardware. It is defined in the header file `<stddef.h>` in C, `<cstddef>` in C++. But we usually don't need to include these files directly. They are already included by `<stdio.h>` in C and by `<iostream>` in C++.

We emphasize `size_t`, and its signed counterpart `ptrdiff_t`, because there will be a parallel pair of typedefs for ''containers'' in C++: `size_type` and `difference_type`.

One kind of conversion is so dangerous that neither a `static_cast` nor a `reinterpret_cast` will do it. This is a conversion between a ''pointer to function'' and a ''pointer to non-function'' (including a ''pointer to `void`''). An example is in line 24 of the following program, which print the address of the function `f` in hexadecimal (or whatever the platform's conventional address format is).

Line 20 tries to give the address of `f` directly to the `<<` operator. But the `<<` in the C++ Standard Library will not accept a pointer to a function. The only acceptable data type to which the pointer can be implicitly converted is `bool`, a substantial loss of information. `bool` is printed as `1` or `0`.

Line 22 converts the pointer into a `size_t`, the integer type that should be big enough to hold it. Like any integer, a `size_t` is printed in decimal. But we would like our pointer to print in hexadecimal. Line 24 therefore casts the `size_t` into a `void *`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/cast/reinterpret_cast.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <cstring>   //for strlen
 4 using namespace std;
 5
 6 void f();
 7
 8 int main()
 9 {
10     unsigned char a[] = "hello";
11     cout << "The length is " << strlen(reinterpret_cast<char *>(a)) << ".\n";
12
13     short s = 0x1234;
14     if (*reinterpret_cast<char *>(&s) == '\x12') {
15         cout << "Host is big endian.\n";
16     } else {
17         cout << "Host is little endian.\n";
18     }
```

```
19
20      cout << f << "\n"
21          << "The address of f in decimal is "
22          << reinterpret_cast<size_t>(f) << ".\n"
23          "The address of f is "
24          << reinterpret_cast<void *>(reinterpret_cast<size_t>(f)) << ".\n";
25
26      return EXIT_SUCCESS;
27  }
28
29  void f()
30  {
31      cout << "f\n";
32  }
```

```
The length is 5.
Host is big endian.
1                                          line 20: a bool prints as 1 or 0
The address of f in decimal is 68072.      lines 21−22: a size_t prints in decimal
The address of f is 0x109e8.               lines 23−4: a void * prints in hex on my platform
```

Casts we will not use

C++ has one cast we will not use, and one we will not use now. The first is the C cast we saw back on p. 63.

```
1       double quotient = (double)dividend / divisor;
```

It is capable of doing anything that a `static_cast` or `reinterpret_cast` can do. In fact, it can do more. The conversions in the above line 24 could be written with a single C cast.

```
2           << (void *)f << ".\n";
```

But don't use this cast in C++. It is of no help when we have to search for every place where a cast is performed. There is no conspicuous word or combination of punctuation marks.

The other notation for a cast looks like a function.

```
3       double quotient = double(dividend) / divisor;
```

It is not really a cast at all; we will see that it is a one-argument ''constructor''. Like the C cast, it is hard to search for. Furthermore, it can be used only when the name of the destination type is a single word. It can convert an expression to `int`, but not to `unsigned long` or to `int *`. It is intended for use only in a ''template'', where the data type will always have a one-word name. Don't use it yet. (It will appear in passing on p. 134.)

1.6.4 Write 0 instead of NULL

A pointer can legally point to three places.

(1) A pointer can point to a variable, including an array element or a structure member.

```
1       int i = 10;
2       int *p = &i;
```

However, a pointer cannot point to a structure bit-field.

(2) A pointer can point to the location after the last element of an array where the next element would be.

```
 3     const size_t n = 10;           //number of elements in the array
 4     int a[n];
 5     int *p = a + n; //point to where the element with subscript 10 would be
```

However, a pointer cannot point to the location before the first element.

(3) A pointer can point to a function, even if the function is ‘‘inline’’ (pp. 97−100).

```
 6     void f();                      //function declaration
 7     void (*p)() = f;               //Let p point to f.
 8
 9     (*p)();                        //one way to call f
10     p();                           //a simpler way to call f
11     f();                           //the simplest way to call f (of course)
```

A pointer that is supposed to not point to any of the above three places should have the value zero, to ensure that it is not accidentally pointing to one of them. In C, this zero was written as the macro `NULL`. In C++, we write it as a plain old `0`. The `0` is an integer, but it can be converted implicitly to a pointer in both languages.

```
12     int *p = NULL;                 /* pointer in C */
13     int *p = 0;                    //pointer in C++
```

The definition of NULL in C

How was the macro `NULL` defined in C, and why don’t we use it in C++? A first attempt at definition would be

```
 1 #define NULL 0                     /* provisional: doesn't work yet */
```

But this runs into trouble when we pass `NULL` to a function whose arguments are not declared. The `printf` function, for example, is declared with the ellipsis dots in line 2. Because of the `%d` and `%p` formats in line 4, the first zero must be passed as an integer argument, the second zero (written as `NULL`) as a pointer argument. One reason the data types must be correct is because an integer and a pointer are different sizes on some platforms; `printf` would be confused if we passed the wrong amount of data to it. The definition of `NULL` in the above line 1, however, will make the computer think that the second argument is an integer. The computer pays no attention to the `%` formats in the first argument of `printf`, and the ellipsis dots are certainly of no help.

```
 2 int printf(const char *format, ...);      /* declaration in <stdio.h> */
 3
 4     printf("%d %p\n", 0, NULL);          /* pass an int and a pointer */
```

For this reason, `NULL` is defined in C as an expression of type pointer:

```
 5 #define NULL ((void *)0)
```

Now the computer will know that the last argument in the above line 4 is indeed a pointer.

In C++ this is not an issue, because function arguments in that language are almost always declared. In fact, the only common functions that use ellipsis are `printf` and `scanf`, and their cousins `fprintf`, `sprintf`, etc.

Why we don’t use NULL in C++

In both languages, a conversion from one type of pointer to another always requires an explicit cast, with the one exception discussed below.

```
 1     int i = 10;
 2     int *p = &i;
 3     char *q = (char *)p;  /* explicit cast to convert int * to char * */
```

The loophole is that in C, a conversion between a `void *` and another type of pointer can be done implicitly, with no cast:

```
4     int i = 10;
5     int *p = &i;
6     void *q = p;                    /* convert int * to void * */
```

Now that we know the definition of `NULL`, we can see that the above line 12 performs a pointer conversion, from "pointer to `void`" to "pointer to `int`". The loophole allows line 12 to compile in C without an explicit cast. But the loophole is closed in C++. Line 12 will not compile in that language unless we change the definition of `NULL` back to zero. The integer zero can be converted, without a cast, to any type of pointer.

```
7 #define NULL 0
```

It would be possible to have two different definitions of `NULL`, one for each language, but it's simpler to dispense with `NULL` in C++.

The other type of zero

A zero of data type `char` should be written with single quotes and a backslash in both languages.

```
1     char c = '\0';                  //a char, not a pointer
2     wchar_t wc = L'\0';             //a wide char, not a pointer
```

In C, the quotes and backslash are merely helpful documentation to show the intent of the zero. `0` is an integer, `'\0'` is a character.

```
3 void f(int i);          /* C example: declare a function */
4
5     f(0);               /* Call the function in line 3. */
6     f('\0');            /* Call the same function. */
```

In C++, however, the quotes and backslash could make the program do something different because of unction name overloading" (pp. 89–94).

```
7 void f(int i);          //C++ example: declare two functions with the same name
8 void f(char c);
9
10    f(0);               //Call the function in line 7.
11    f('\0');            //Call the function in line 8.
```

1.6.5 Pass-by-Value vs. Pass-by-Reference

C and C++ have two ways of passing an argument to a function. The most common, *pass-by-value*, creates a copy of the value of the argument and passes this copy to the function. The function cannot change the value of the argument, because the function never receives the argument. Only a copy falls into the function's hands.

To permit a function to change the value of an argument we must perform *pass-by-reference*, in which the address of the argument, rather than a copy of its value, is passed to the function. Knowing where the original argument lives, the function can install a new value into it.

The classic example of pass-by-value is `printf`, not counting the format string. The classic example of pass-by-reference is `scanf`, again not counting the format string which only coincidentally is passed by reference.

```
1     int i = 10;           /* C example */
2
3     printf("%d\n", i);    /* printf can use the value of i but can't change it. */
4     scanf("%d", &i);      /* scanf can change the value of i. */
```

Here is the definition of a function whose first argument is passed by value and whose second is passed by reference.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/pointer/byvalue.C

```
 1 #include <iostream>         //C++ example
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void f(int copy, int *p);   //function declaration
 6
 7 int main()
 8 {
 9     int i = 10;
10     int j = 20;
11
12     f(i, &j);
13
14     cout << "i == " << i << "\n"
15         << "j == " << j << "\n";
16
17     return EXIT_SUCCESS;
18 }
19
20 void f(int copy, int *p)    //function definition
21 {
22     ++copy;   //has no effect on i
23     ++*p;     //adds 1 to j; means *p = *p + 1
24 }
```

```
i == 10
j == 21
```

Pass-by-value is usually avoided where we can get away with pass-by-reference. For a deliberate use of pass-by-value, see lines 75–77 of `date.h` on p. 274.

Read-only pointer arguments

When called from line 12, the function `f` has read and write access to the array `a`, but read-only access to the array `b`.

See the `const` argument(s) in the declaration of `strcpy` and the other familiar string functions.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/pointer/pointer_argument.C

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void f(int *p, const int *q);
 6
 7 int main()
 8 {
 9     int a[] = {10, 20, 30};
10     int b[] = {40, 50, 60};
11
```

```
12     f(a, b);
13     return EXIT_SUCCESS;
14 }
15
16 void f(int *p, const int *q)
17 {
18     p[0] = 70;     //change the 10 to 70
19     //q[0] = 80;   //won't compile: try to change the 40 to 80
20 }
```

1.6.6 References

References an alternative notation for pointers

Column 1 creates the variable `i`, initializes and assigns to it, and outputs its value, address, and size. Column 2 performs the same operations on the same variable, accessing it via the pointer `p`. When we put the address of a variable into a pointer, we must always apply the `&` operator to the variable. See line 7 of column 2. When we dereference the pointer to get back to the target variable, we must always apply the `*` operator to the pointer. Line 9 of column 2 puts 20 into `i`.

A *reference* is another notation for a pointer. Deep inside the machine, the reference `r` in column 3 is exactly the same as the pointer `p` in column 2. The pointer and the reference contain the address of the same variable, `i`. The only two differences between them are on the surface: in the source code of the program.

(1) When we put the address of a variable into a reference, we do not apply the `&` operator to the variable. Line 7 of column 3 appears to be putting the value of `i` into `r`. But `r` can't hold an `int`: it can hold only an address. We are actually putting the address of `i` into `r`.

(2) When we dereference the reference to get back to the target variable, we do not apply the `*` operator to the reference. Line 9 of column 3 appears to be putting 20 into `r`. But `r` can't hold an `int`: it can hold only an address. We are actually putting 20 into `i`.

Why did they invent a way to take the address of a variable without applying `&` to it, and dereference a pointer without applying `*` to it? A hint will come on p. 76, but the real story will have to wait until we do ''operator overloading''.

A reference always contains the address of the *same* variable. Our reference `r` is therefore like the `*const` pointer `p` in column 2.

A reference has no memory address of its own. Line 13 of column 3 therefore prints the address of `i`, not of `r`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/reference/reference.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     int i = 10;              int *const p = &i;         int& r = i;
 8
 9     i = 20;                  *p = 20;                   r = 20;
10     ++i;                     ++*p;                      ++r;
11
12     cout << i << "\n";       cout << *p << "\n";        cout << r << "\n";
13     cout << &i << "\n";      cout << p << "\n";         cout << &r << "\n";
```

```
14     cout << sizeof i << "\n"; cout << sizeof *p << "\n"; cout << sizeof r << "\n";
15
16     return EXIT_SUCCESS;
17 }
```

```
23              line 12: the value of i
23
23
0xffbff63c      line 13: the address of i
0xffbff63c
0xffbff63c
4               line 14: the sizeof i
4
4
```

Read-only reference

A read-only reference is just like a read-only pointer.

```
1     int i = 10;
2
3     const int *const p = &i;
4     *p = 20;   //try to change i to 20: won't compile due to 1st const in line 3
5
6     const int& r = i;
7     r = 30;    //try to change i to 30: won't compile due to const in line 6
```

The word "reference" now means two different things.

(1) Pass-by-value vs. pass-by-reference:

```
1     printf("%d\n", i);  /* i is passed by value. */
2     scanf("%d", &i);    /* i is passed by reference. */
```

(2) Pointer notation vs. reference notation:

```
3     int i = 10;
4     int *const p = &i;  //p is a pointer to i.
5     int& r = i;         //r is a reference to i.
```

To avoid confusion, we will no longer use the word "reference" in the first sense. Instead, we will now say

```
6     printf("%d\n", i);  /* i is passed by value. */
7     scanf("%d", &i);    /* pass the address of i to the function */
```

When to use a reference

There are two reasons for passing the address of a variable to a function. Each reason will now have a separate notation.

(1) We want to let the function change the value of a variable; the classic example is `scanf`. In this case, let the argument be a read/write pointer to the variable as in C. See the argument `p` in line 22.

(2) We want to save time by avoiding the construction, and eventual destruction, of a copy of the variable. In this case, let the argument be a read-only reference to the variable. See the argument `r` in line 22. In this case, the variable is merely an `int`. But if the variable was larger, it would be worthwhile to pass it as a reference.

If both reasons apply, let the argument be a read/write pointer to the variable.

Deep in the machine, the last two arguments in line 13 are passed the same way. In both cases we are passing the address of the variable to the function.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/reference/pass_int.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void f(int copy, int *p, const int& r);
 6
 7 int main()
 8 {
 9     int i1 = 10;
10     int i2 = 20;
11     int i3 = 30;
12
13     f(i1, &i2, i3);
14
15     cout << "i1 == " << i1 << "\n"
16         << "i2 == " << i2 << "\n"
17         << "i3 == " << i3 << "\n";
18
19     return EXIT_SUCCESS;
20 }
21
22 void f(int copy, int *p, const int& r)
23 {
24     cout << "Arguments are " << copy << ", " << *p << ", " << r << ".\n";
25
26     ++copy;    //has no effect on i1
27     ++*p;      //add 1 to i2; means *p = *p + 1
28     //++r;     //won't compile: r is a read-only reference
29 }
```

```
Arguments are 10, 20, 30.
i1 == 10
i2 == 21
i3 == 30
```

A read/write reference argument is deceptive

A reference argument should almost always be read-only, like the `r` in the above line 28. It can be a read/write only when the name of the function clearly indicates that the function changes the value of its argument.

Here are the only examples we will encounter in this book. All but the last two are from the C++ language itself or its standard library.

(1) the `operator++` functions for enumerations on p. 290;

(2) The second argument of the `operator>>` function is the variable that receives a new value from input via the operator `>>`. (The first argument of `operator>>` and `operator<<` is also a read/write reference. In fact, a "stream" argument is always a read/write reference. See pp. 324−326.)

(3) the `get` member function on p. 329;

(4) the `swap` algorithm on p. 649;

(5) the `advance` algorithm on p. 914;

(6) the `put` member function of a facet on p. 1049;

(7) my `terminal::next` member function on p. 158, abolished on p. 966. This function is a stop-gap measure until we acquire the machinery to do the job correctly.

(8) my `decrement` function object on p. 880. The author hopes the name is sufficiently explicit.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/reference/readwrite_reference.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void increment(int *p, int& r);
 6
 7 int main()
 8 {
 9     int i = 10;
10     int j = 20;
11
12     //Obvious that line 14 can change i,
13     //dangerously unobvious that line 14 can also change j.
14     increment(&i, j);
15
16     cout << "i == " << i << "\n"
17         << "j == " << j << "\n";
18
19     return EXIT_SUCCESS;
20 }
21
22 void increment(int *p, int& r)
23 {
24     ++*p;   //means *p = *p + 1
25     ++r;    //means r = r + 1
26 }
```

```
i == 11
j == 21
```

Return the address of a variable from a function

An expression that can be used as the left operand of the assignment operator `=` is called an *lvalue* (pp. 12−13); the L stands for ‘‘left’’. For example, a variable is an lvalue.

```
1     x = 10;
```

For reasons we will explain on p. 76, the return value of a function must sometimes be used as an lvalue. When line 17 tries to do this with the expression `f()`, however, it does not compile. A value that has been returned by value is not an lvalue.

But a value that has been returned by reference is an lvalue (lines 18 and 19). Deep in the machine, the functions `g` and `h` both return the address of `i`. The return value of `g` needs the dereferencing operator `*` before it can be used as an lvalue in line 18. The return value of `h` in line 19 doesn’t need the asterisk.

For even more arcane reasons, also to be explained on p. 76, the return value of a function must sometimes be used as an lvalue without an asterisk. We will therefore declare the return type to be the reference in line 38 rather than the pointer in line 33.

To qualify as an lvalue, it must also be possible to apply the ‘‘address of’’ operator & to the expression. Once again, the `g()` in line 22 must have an asterisk before it can be used as an lvalue. (The & and * in this line can cancel each other out.) The `h()` in line 23 needs no asterisk. For an example, see p. 900.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/reference/return_int.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int i = 10;   //statically allocated so it won't be destroyed by a return
 6
 7 int f();
 8 int *g();
 9 int& h();
10
11 int main()
12 {
13     cout << "f() == " << f() << "\n"
14         << "*g() == " << *g() << "\n"
15         << "h() == " << h() << "\n";
16
17     //f() = 20;   //won't compile
18     *g() = 30;    //Change the value of i to 30.  Must have the asterisk.
19     h() = 40;     //Change the value of i to 40.  Don't need asterisk.
20
21     //cout << &f() << "\n";              //won't compile
22     cout << &*g() << " " << g() << "\n"; //Print the address of i.
23     cout << &h() << "\n";                //Print the address of i.
24
25     return EXIT_SUCCESS;
26 }
27
28 int f()
29 {
30     return i;    //Create a copy of the value of i and return the copy.
31 }
32
33 int *g()
34 {
35     return &i;   //Return the address of i.
36 }
37
38 int& h()
39 {
40     return i;    //Return the address of i.
41 }
```

```
f() == 10
*g() == 10
h() == 10
0x20dac 0x20dac
0x20dac
```

Why did Stroustrup invent references?

Why is it so important to be able to use the return value of a function, without an asterisk, as an lvalue (line 19 above)? It seems like an unnatural thing to do:

```
1     x = sqrt(y);   //Natural: x is an lvalue.
2     sqrt(x) = y;   //Unnatural: sqrt(x) is not an lvalue.  Won't compile.
```

But when we perform operator overloading with ''objects'', we'll have to do this all the time. Let's assume that an object is a variable. We will apply a subscript to an object (`v` in the following example) just like we do to an array. We then use the subscripted object as an lvalue:

```
3     //Store the number 20 inside the object v at position 10.
4     v[10] = 20;
```

When we write the above line 4, the computer behaves as if we had written line 5, calling a function with the admittedly strange name `v.operator[]`. The subscript in the [square brackets] in line 4 is passed as the argument to this function:

```
5     v.operator[](10) = 20;          //This is what line 4 actually does.
```

To make it possible to use the return value as an lvalue, `v.operator[]` must return an address. (In fact, the function returns the address within in `v` where the number 20 is to be stored.) One way to return an address is as a pointer. If the return type of `v.operator[]` were a pointer, we would have to write line 4 as

```
6     *v[10] = 20;
```

to make the computer behave as if we had written

```
7     *v.operator[](10) = 20;
```

But we want the syntax of line 4 to mimic the familiar syntax of an array. The return type of `v.operator[]` is therefore a reference. This permits us to write line 4 without the asterisk.

Never return the address of a non-static local variable.

A non-static local variable evaporates as we return from the function in which it is defined. If we return its address, we are returning the address of garbage. We should return the addresses of only those variables that do *not* evaporate as we return.

The following functions are wrong for the same reason. We now have two different notations in which to write the same mistake in C++.

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int *g();
 6 int& h();
 7
 8 int main()
 9 {
10     cout << "*g() == " << *g() << "\n"
11          << "h() == " << h() << "\n";
```

```
12
13      return EXIT_SUCCESS;
14  }
15
16  int *g()
17  {
18      int i = 10;
19      return &i;          //i evaporates as g returns.
20  }
21
22  int& h()
23  {
24      int i = 10;
25      return i;           //i evaporates as h returns.
26  }
```

A reference to a structure

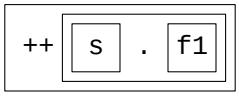
The notation for a pointer to a structure is different from the notation for a pointer to a non-structure. For example, the `p->f1` in line 14 of column 2 means `(*p).f1`: the operator `->` does the work of the operators asterisk and dot. And now that the expression has only one operator, there is no longer a need for the parentheses.

Since the notation is so different, here is the original pointer vs. reference example again, this time with pointers and references to a structure.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/reference/structure.C`

```
 1  #include <iostream>
 2  #include <cstdlib>
 3  using namespace std;
 4
 5  struct str {
 6      int f1;
 7      int f2;
 8  };
 9
10  int main()
11  {
12      str s = {10, 20};         str *const p = &s;         str& r = s;
13
14      s.f1 = 30;                p->f1 = 30;                r.f1 = 30;
15      ++s.f1;                   ++p->f1;                   ++r.f1;
16
17      cout << s.f1 << "\n";     cout << p->f1 << "\n";     cout << r.f1 << "\n";
18
19      cout << &s << "\n";       cout << p << "\n";         cout << &r << "\n";
20      cout << &s.f1 << "\n";    cout << &p->f1 << "\n";    cout << &r.f1 << "\n";
21
22      cout << sizeof s << "\n"; cout << sizeof *p << "\n"; cout << sizeof r << "\n";
23      cout << sizeof s.f1<<"\n";cout << sizeof p->f1<<"\n";cout << sizeof r.f1<<"\n";
24
25      return EXIT_SUCCESS;
26  }
```

The ++ in line 15 of column 1 adds 1 to `s.f1`, not to `s`. The box around the subexpression `s.f1` causes the `s.f1` to be treated as a unit by the operators outside of it. The ++ cannot reach into the box and single out the sub-subexpression `s`.




```
33              line 17
33
33
0xffbff638      line 19
0xffbff638
0xffbff638
0xffbff638      line 20: the same address
0xffbff638
0xffbff638
8               line 22
8
8
4               line 23
4
4
```

Pass the address of a structure to a function

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/reference/pass_structure.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 struct str {
 6     int f1;
 7     int f2;
 8 };
 9
10 void f(str copy, str *p, const str& r);
11
12 int main()
13 {
14     str a1 = {10, 20};
15     str a2 = {30, 40};
16     str a3 = {50, 60};
17
18     f(a1, &a2, a3);
19
20     cout << "a1.f1 == " << a1.f1 << "\n"
21         << "a2.f1 == " << a2.f1 << "\n"
22         << "a3.f1 == " << a3.f1 << "\n";
23
24     return EXIT_SUCCESS;
25 }
```

```
26
27 void f(str copy, str *p, const str& r)
28 {
29     ++copy.f1;   //has no effect on a1.f1; means copy.f1 = copy.f1 + 1
30     ++p->f1;     //adds 1 to a2.f1; means p->f1 = p->f1 + 1
31     //++r.f1;    //won't compile
32 }
```

```
a1.f1 == 10
a2.f1 == 31
a3.f1 == 50
```

Return the address of a structure from a function

Deep inside the machine, the functions g and h return the address of s:

—On the Web at
http://i5.nyu.edu/~mm64/book/src/reference/return_structure.C

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 struct str {
 6     int f1;
 7     int f2;
 8 };
 9
10 str s = {10, 20};
11
12 inline str f();
13 inline str *g();
14 inline str& h();
15
16 int main()
17 {
18     cout << "f().f1 == " << f().f1 << "\n"
19         << "g()->f1 == " << g()->f1 << "\n"
20         << "h().f1 == " << h().f1 << "\n";
21
22     g()->f1 = 30;   //Change s.f1 to 30.  Must use arrow.
23     h().f1 = 40;    //Change s.f1 to 40.  Must use dot.
24
25     return EXIT_SUCCESS;
26 }
27
28 str f()
29 {
30     return s;
31 }
32
33 str *g()
34 {
35     return &s;
36 }
```

```
37
38 str& h()
39 {
40     return s;
41 }
```

```
f().f1 == 10
g()->f1 == 10
h().f1 == 10
```

References to the derived types

There are references to pointers.

```
 1     int i = 10;                    //an int
 2     int *p = &i                    //a pointer to the int
 3     int *& r = p;                  //a reference to the pointer to the int
 4
 5     cout << "The address of i is " << r << ".\n" //the value of p
 6         << "The value of i is " << *r << ".\n";  //the value of *p (i.e., i)
```

Another example is in line 17 of `point.C` in p. 373.

There are references to arrays.

```
 7     int a[3] = {10, 20, 30};       //an array
 8     int (&r)[3] = a;               //a reference to the array: () and 3 required
 9
10     cout << r[0] << "\n"           //the value of a[0]
11         << r[1] << "\n"           //the value of a[1]
12         << r[2] << "\n";          //the value of a[2]
```

There are even references to functions:

```
13 void f();                          //function declaration
14
15     void (&r)() = f;               //a reference to the function
16     r();                           //Call the function.
```

But we are never allowed to take the address of a reference. The address might exist, but like an object behind the event horizon of a black hole, we are never allowed to see it. This is not as strange as it sounds. We already know many things whose memory address we are not allowed to see. For example, a literal such as `10` or `'A'` has no visible address:

```
17     const int *p = &10;            //won't compile
18     const char *p = &'A';          //won't compile
```

If we do try to take the address of a reference, all we get is the address of the original variable. Line 21 outputs the address of `i`, not the address of `r`. `r` has no address, or at least no address that we can see.

```
19     int i = 10;
20     int& r = i;
21     cout << &r << "\n";            //Output the address of i.
```

Since we are not allowed to take the address of a reference, there are no pointers to references, references to references, or arrays of references. By definition, an array is a series of elements at equally spaced memory addresses. But a reference has no visible address, so it cannot be an array element. Later, we will see an improved array called a `vector`. There will be no `vector`'s of references, or any other containers of references.

▼ **Homework 1.6.6a: a cast to a reference**

Line 14 of `reinterpret_cast.C` on p. 66 casts one type of pointer into another type of pointer.

```
1     if (*reinterpret_cast<char *>(&s) == '\x12') {
```

Change it to cast a variable into a reference to another type of variable. (This is called *type punning*.)

```
2     if (reinterpret_cast<char &>(s) == '\x12') {
```

Does it still work? Is it simpler? Is it easier to understand—which is not at all the same thing? Can you get used to it? Other examples will be on pp. 655 and 857.
▲

1.7 Enhancements to Functions

1.7.1 Functions with No Arguments

```
1 /* C example */
2
3 void f(void);     /* This function takes no arguments. */
4 void f();         /* This function could take any arguments (obsolete). */
5 void f(...);      /* This function could take any arguments (current). */
```

In C++, the following line 8 declares a function that takes no arguments. Line 9 does the same thing, but don't write it until p. 84.

```
6 //C++ example
7
8 void f();         //This function takes no arguments.
9 void f(void);     //This function takes no arguments.
10 void f(...);     //This function could take any arguments.
```

1.7.2 Call a C Function from a C++ Program

In real life, a C++ program has to call functions written in other languages. Here is how a C++ program can call a function written in C.

A C function and a C++ function cannot be defined in the same file. This implies that a C++ program that calls C functions must be split into at least two source files.

A program that comprises two or more source files

Let's begin with a multi-file program all in the same language. If the same declarations need to be present at the start of each file, they can be written once and for all in a header file.

A header file might contain statements that are legal to write once but not twice. We might therefore get error messages if we `#include` the same header file twice.

```
1 #include <stdio.h>      /* C example */
2 #include <stdio.h>
```

The same problem would occur even if we `#include`'d two different header files

```
3 #include <stdio.h>
4 #include <another.h>
```

if `another.h` contained the line `#include <stdio.h>`.

The preprocessor directives in line 1, 2, and 6 allow the header file to be compiled only the first time that it is `#include`'d in a given `.C` file. The first time the computer reads line 1, the `#ifndef` is true: there is no macro named `FGH`. The computer then reads everything from line 1 to the `#endif` in line 6, and the first thing it does in these lines is to define the macro in line 2. (The macro counts as being defined even though it contains only the null string.) If the header file is ever `#include`'d again in the same `.C` file, line 1 will now be false. We will skip directly to the `#endif`, ignoring the entire header file. This will be our only use of a macro in C++.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sources/fg.h`

```
 1 #ifndef FGH
 2 #define FGH
 3
 4 void f();   //function declaration
 5 int g(int i);
 6 #endif
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sources/fg.C`

```
 1 #include <iostream>
 2 #include "fg.h"
 3 using namespace std;
 4
 5 void f()   //function definition
 6 {
 7     cout << "f\n";
 8 }
 9
10 int g(int i)
11 {
12     return 2 * i;
13 }
```

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/sources/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include "fg.h"
 4 using namespace std;
 5
 6 int main()
 7 {
 8     f();
 9     cout << g(10) << "\n";
10
11     return EXIT_SUCCESS;
12 }
```

When compiling on Unix we mention only the names of the `.C` files, not the names of the `.h` files.

```
1$ g++ -o ~/bin/prog main.C fg.C
2$ ls -l ~/bin/prog
3$ prog
```

```
f
20
```

Call C functions from a C++ program

The name of each file tells the computer what language the file is written in.

(1) In Unix, define the C++ functions in files whose names end with uppercase `.C`, and the C functions in files whose names end with lowercase `.c`.

(2) In Microsoft Visual C++ and the Project Builder IDE, define the C++ functions in files whose names end with `.cpp`, and the C functions in files whose names end with `.c`.

(3) In Borland Turbo C++, define the C++ functions in files whose names end with `.CPP`, and the C functions in files whose names end with `.C`.

Here is the half that is written in C++. A function must always be declared before it is called. If the function is written in C, use the funny declarations in lines 6−7.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/callC/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 //These 2 declarations should be written in a header file, and soon will be.
 6 extern "C" void f(); //uppercase C in double quotes
 7 extern "C" int g(int i);
 8
 9 int main()
10 {
11     f();
12     cout << g(10) << "\n";
13
14     return EXIT_SUCCESS;
15 }
```

We will probably be calling many C functions. Instead of writing a separate `extern "C"` for each declaration in the above lines 6−7, we can write a single `extern "C"` with curly braces.

```
16 extern "C" {
17     void f();
18     int g(int i);
19 }                       //no semicolon
```

Here is the other half of the program, written in C. The `void`'s in the parentheses in the declaration in line 4 and the definition in line 7 are optional in C++ but required in C.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/callC/fg.c`

```
 1 #include <stdio.h>
 2
 3 /* These 2 declarations should be written in a header file, and soon will be. */
 4 void f(void);
 5 int g(int i);
 6
 7 void f(void)   /* function definition */
 8 {
 9     printf("f\n");
```

```
10 }
11
12 int g(int i)
13 {
14     return 2 * i;
15 }
```

The `-c` option tells `gcc` that the file `fg.c` is not intended to be a complete program. It isn't—it has no `main` function. It is only one file of a larger program.

```
1$ gcc -c fg.c                          Create the "object" file fg.o.
2$ ls -l fg.o                                          minus lowercase L

3$ g++ -o ~/bin/prog main.C fg.o        Create the executable file ~/bin/prog.
4$ ls -l ~/bin/prog
5$ prog
```

```
f
20
```

A header file acceptable to both languages

We declared the functions `f` and `g` twice, in lines 5–7 of the above `main.C` and lines 3–5 of `fg.c`. Instead of writing the declarations in each file, we should write them once and for all in a header file.

The following header can be included in both of the above files because we restricted ourselves to features that are legal in both languages.

(1) Comments are delimited by `/*` and `*/`, not by `//`.

(2) Functions with no arguments are declared with an argument list of `(void)`.

(3) C functions are declared without the `extern "C"`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/callC/fg.h`

```
 1 #ifndef FGH
 2 #define FGH
 3
 4 void f(void);   /* The void in parens is optional in C++ but required in C. */
 5 int g(int i);
 6 #endif
```

We can now change lines 3–5 of `fg.c` to

```
 7 #include "fg.h"
```

and lines 5–7 of `main.C` to

```
 8 extern "C" {
 9 #include "fg.h"
10 }
```

The above lines 8–10 show the real purpose of the `extern "C"` with curly braces in the above lines 16–19. It has nothing to do with avoiding the repetition of the words `extern "C"`. The real intent is to banish these words from the lines that declare the functions. What remains on those lines will now be acceptable to both languages, and can be moved to a header file. The `(void)` in the header file is permitted in C++ for the same reason: so we can write a declaration acceptable to both languages.

1.7.3 A Portable Interface to the Terminal

We will write a video game to explore object-oriented programming, inheritance, templates, and the C++ Standard Library. To focus on these imposing topics, we will ignore color, pixels, and sound files. Our game will treat the screen as a monochrome display of rows and columns of characters, emitting a monotone beep. Input will come only from the keyboard, not from the mouse.

Each platform—Unix, Microsoft, Borland—provides functions for graphics and other special effects. Unfortunately, these native functions have different names, arguments, and return types on each platform. To keep the game independent of platform, we will make no direct calls to these functions. Instead, we will call the following `term_` functions which will call the native functions for us. The `term_` functions constitute the platform-independent base upon which the game will be built. They are written in C. Their header file `term.h` can be included in a C or a C++ file; they can be called from either language.

The function `term_put` in line 15 writes one character at the given (`x`, `y`) position on the screen. The `term_puts` in line 16 writes a string of characters, not counting the terminating `'\0'`, starting at the given position. The `x`'s are the column numbers; the `y`'s, the row numbers. Only printable characters can be written—those for which the C Standard Library function `isprint` returns non-zero. All other characters will cause an error message. For example, the newline and tab characters are unprintable, but there is never any need to write them on the screen. We can space horizontally and vertically by providing the appropriate `x` and `y` values.

A character that has been written on the screen can be read back with `term_get` in line 19. It returns the character, or a blank if no character has been written there yet.

On every platform, the `x`'s go from left to right and the `y`'s from top to bottom. Both start at zero, so the origin (`0`, `0`) is at the upper left corner of the screen. The number of columns and rows will be different on each platform. But on any platform, the functions `term_xmax` and `term_ymax` in lines 11–12 will return these numbers. If the number of columns is 80, for example, the legal values of the `x` arguments will range from 0 to 79 inclusive. An out of range `x` or `y` will cause an error message.

When representing coördinates that start at zero we will always follow the convention of the C++ Standard Library and use unsigned numbers. This will prevent the coördinates from ever being negative. Examples in this group of functions are the `x` and `y` arguments of `term_put`, `term_puts`, and `term_get`, and the return values of `term_xmax` and `term_ymax`. A previous example was the unsigned data type `size_t` used for array subscripts (p. 66).

Calls to the function `term_key` in line 23 return the characters typed at the keyboard. If every character has already been returned, or before any character has yet been typed, it will return the character `'\0'`. `term_key` differs from the C function `getchar` in that it always returns immediately, without waiting for the user to press `RETURN`. In other words, it gives us a live keyboard.

The function `term_wait` in line 25 pauses for the specified number of milliseconds. `term_beep` in line 26 beeps the terminal.

We saved the two most important `term_` functions for last. Before we can do any special effects, there is always some setup to be done. On some platforms we have to put the screen into graphics mode; on others, we have to pop up a graphics window. Similarly, there is always some cleanup at the end: we have to put the screen back into text mode, or make the graphics window disappear.

On every platform, the functions `term_construct` and `term_destruct` in lines 5 and 6 will do whatever setup and cleanup are necessary. They must be the first and last `term_` functions called. The other `term_` functions are guaranteed to work correctly only between the calls to these two.

Failure to call `term_destruct` may leave your terminal in an unusable state. For example, under normal conditions every character that we type is echoed onto the screen. But during the interval between `term_construct` and `term_destruct`, this echoing is turned off. To see the characters as they are typed, we must write them on the screen ourselves (line 32 of the following `main.C`).

In C, it is up to the programmer to make sure that both functions are called once, `term_construct` before `term_destruct`. In C++, the language will pair these function calls for us when we have ‘‘constructors’’ and ‘‘destructors’’. See pp. 164–166, 163.

For the time being, we will restrict ourselves to having only one terminal. The first step toward lifting this restriction will be on pp. 994−999.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/term/term.h

```
 1 #ifndef TERMH
 2 #define TERMH
 3
 4 /* Must be called before and after the other term_ functions. */
 5 void term_construct(void);
 6 void term_destruct(void);
 7
 8 /* Legal x values go from 0 to term_xmax() - 1 inclusive.
 9    Legal y values go from 0 to term_ymax() - 1 inclusive. */
10
11 unsigned term_xmax(void);   /* number of columns of characters */
12 unsigned term_ymax(void);   /* number of rows of characters */
13
14 /* Display a character or string on the screen. */
15 void term_put (unsigned x, unsigned y, char c);
16 void term_puts(unsigned x, unsigned y, const char *s);
17
18 /* Return the character at the given position on the screen. */
19 char term_get(unsigned x, unsigned y);
20
21 /* Return immediately with the key the user pressed.  If no key was pressed,
22 return immediately with '\0'. */
23 char term_key(void);
24
25 void term_wait(int milliseconds);    /* 1000 milliseconds == 1 second */
26 void term_beep(void);
27 #endif
```

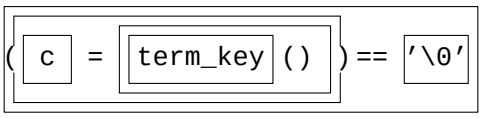
▼ Homework 1.7.3a: test the terminal interface

Run the following C++ program on your machine to test the `term_` functions.

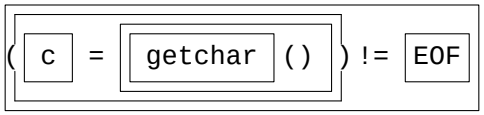
Lines 10 and 39 set up and clean up the terminal. Lines 11−12 get the dimensions of the screen; lines 14−15 compute the coördinates of the center point.

Line 17 writes an `'X'` at this position and 18 reads it back. Line 19 writes the read-back character next to the original one. You should see the name of the Mexican beer Dos Equis (XX) at the center of the screen. Line 21 writes a string of characters, starting at the upper left corner of the screen.

The `term_key` function always returns immediately. To wait until a character is typed, we must call it in the little `while` loop in lines 25−26. We remain trapped in this loop until we are brave enough to type a character. This wastes processing power, but is a simple way to get the job done. The expression in line 25,



executes the same operators (substituting `==` for `!=`), in the same order, as the following classic idiom in C.



Earlier examples of this idiom were on p. 38.

We break out of the `while` loop when we type a character, and write it onto the screen in line 32. (Remember, the normal echoing of characters is turned off between the calls to `term_construct` and `term_destruct`.) What happens if you type an unprintable character such as newline or tab?

The code we have just walked through in lines 25−32 is inside of the classic nested pair of `for` loops in lines 23−24. It is executed over and over, writing each character we type onto the screen at the next position. When we break out with a `'q'`, lines 37−38 test the wait and beep.

A C program would have to declare all its variables immediately after the `{` in line 9. Our C++ program declares its variables when we have values to put into them in lines 11−15 and 18.

For the time being, we are passing a group of variables (`x` and `y`) to a series of function calls. The code will become simpler and faster when they are merged into a single variable (p. 177). Also for the time being, we need two `for` loops with two counters (the same `x` and `y`) because the screen is two-dimensional. When they are merged into a single variable (called an ‘‘iterator’’), we will be able to loop through the screen with only one loop and one counter.

The file `main.C` is not the complete program. We also need the `term.h` file of function declarations, and the `term.c` file of function definitions. These two files are in the same directory on the web.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/term/main.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 extern "C" {
 4 #include "term.h"
 5 }
 6 using namespace std;
 7
 8 int main()
 9 {
10     term_construct();
11     const unsigned xmax = term_xmax();
12     const unsigned ymax = term_ymax();
13
14     unsigned x = xmax / 2;   //center of screen
15     unsigned y = ymax / 2;
16
17     term_put(x, y, 'X');     //Draw XX at center of screen.
18     char c = term_get(x, y);
19     term_put(x + 1, y, c);
20
21     term_puts(0, 0, "Please type printable characters ending with a q.");
22
23     for (y = 1; y < ymax; ++y) {
24         for (x = 0; x < xmax; ++x) {
25             while ((c = term_key()) == '\0') {
26             }
27
28             if (c == 'q') {   //quit
29                 goto done;
```

```
30              }
31
32              term_put(x, y, c);  //Echo the character the user typed.
33          }
34      }
35
36      done:;
37      term_wait(1000);    //one full second
38      term_beep();
39      term_destruct();
40      return EXIT_SUCCESS;
41  }
```

▲

List of the three source files that constitute the test program

(1) `term.h` and `term.c` (both online in the same directory on pp. 86−87). They are the only two written in C; the other is in C++.

(2) `main.C` (pp. 87−88)

Select the platform.

The `term.c` file is written in C. Be sure that you don't accidentally tell your computer that it's written in C++ by giving it the wrong filename extension. Before compiling, uncomment (i.e., remove the comment delimiters from) exactly one of the following three macro definitions at the top of this file:

```
1 /* #define UNIX */
2 /* #define MICROSOFT */
3 /* #define BORLAND */
```

For example, if you were compiling under Microsoft, you would change line 2 to

```
4 #define MICROSOFT
```

Make no other change to `term.c`. Unix pros can use the `-D` option of `gcc` and `g++` instead of uncommenting.

Compile under Unix

See `curses`(3curses) at `http://i5.nyu.edu/~mm64/man/`, or *Programming with curses* by John Strang; O'Reilly & Associates, 1986; ISBN 0−937175−02−1.

`http://www.oreilly.com/catalog/curses/`

The "minus uppercase `I` dot" option tells `gcc` and `g++` to `#include` the `term.h` file in the current directory instead of any `term.h` file that might be in other directories.

The `-DUNIX=` option defines the macro `UNIX` to be the null string, eliminating the need for the above uncommenting. Bloomberg people should also give the compiler the option `-D_WIDEC_H=` to prevent the compiler from including the file `/usr/include/widec.h`. Remember to use this option whenever compiling `term.c`.

The minus lowercase `c` option tells `gcc` to create a `.o` file instead of an executable file. The minus lowercase L option tells `g++` to link in the library `libcurses.a`.

```
1$ gcc -I. -DUNIX= -c term.c        Create the object file term.o.
2$ ls -l term.o                     minus lowercase L
```

```
3$ g++ -I. -o ~/bin/tester main.C term.o -lcurses
4$ ls -l ~/bin/tester
5$ tester                Run it; but first make sure your terminal is set to vt100, bit ansi.
```

If you do not see the two X's, set your TERM environment variable to vt100 (lowercase VT one hundred) To do this, Korn shell users should say

```
6$ echo $TERM            See what's already in TERM.
7$ export TERM=vt100
8$ echo $TERM            Verify that we put vt100 into TERM.
```

C shell users should say

```
9$ echo $TERM            See what's already in TERM.
10$ setenv TERM vt100
11$ echo $TERM           Verify that we put vt100 into TERM.
```

Then try again.

Compile under Microsoft

Create a ''Win32 Console Application''. In Microsoft Visual C++ (part of Visual Studio), you have to create a blank project with no files in it. If you try to modify the main.cpp file of their ''Hello, world'' program, you get ''unresolved external symbol __CrtDbgReport''.

1.7.4 Function Name Overloading

An overloaded function name

In a C program, every function has to have a different name.* Lines 17−19 call functions with three different names to print arguments of different data types: int, char, and double.

The printf in line 31 outputs the ASCII code of the character in decimal. The first argument of printf is declared to be a string. The remaining arguments have no declarations.

```
1 int printf(const char *format, ...);            //ellipsis dots
```

A char argument passed to printf would therefore be widened to int. If the data type char were signed on this platform, the widening would be accomplished by sign extension. For example, an 8-bit char would appear as a number in the range −128 to 127 inclusive. Line 31 prevents this by casting the char to unsigned char, which will be widened by zero extension. Now the char will appear in the range 0 to 255 inclusive. We saw the same cast in C++ in line 14 of static_cast.C on p. 65. The %u format, without the cast, would not be enough. See what a pain it is to call a function whose arguments are undeclared?

Line 41 breaks the double into its mantissa and exponent: $65 = \frac{65}{128} \times 128 = .5078125 \times 2^7$.

—On the Web at
http://i5.nyu.edu/~mm64/book/src/foverload/print.c

```
1 #include <stdio.h>      /* C example */
2 #include <stdlib.h>
3 #include <ctype.h>      /* for isprint */
4 #include <math.h>       /* for frexp */
5
6 void print_int(int i);  /* function declaration */
```

* One exception: a static function defined in one .C file could have the same name as a function defined in another .C file.

```
 7 void print_char(char c);
 8 void print_double(double d);
 9
10 int main()
11 {
12     int i = 65;
13     char c = 'A';
14     double d = 65.0;
15
16     print_int(i);
17     print_char(c);
18     print_double(d);
19
20     return EXIT_SUCCESS;
21 }
22
23 void print_int(int i)   /* function definition */
24 {
25     printf("%d\n", i);
26 }
27
28 void print_char(char c)
29 {
30     /* Output the character code of c as a non-negative decimal integer. */
31     printf("%u", (unsigned char)c);   /* cast to avoid sign extension */
32     if (isprint((unsigned char)c)) {
33         printf("\t'%c'", c);
34     }
35     printf("\n");
36 }
37
38 void print_double(double d)
39 {
40     int exponent;
41     double mantissa = frexp(d, &exponent);
42
43     printf("%g, mantissa == %g, exponent == %d\n", d, mantissa, exponent);
44 }
```

```
65
65      'A'
65, mantissa == 0.507812, exponent == 7
```

Our functions have the names in column 1. But other naming conventions are possible: column 2 has embedded uppercase letters, and column 3 has the `print` at the end of the identifier. Some data types have more than one name: `long unsigned` vs. `unsigned long`. Whichever convention we adopt will have to be consistently enforced.

```
print_int               printInt               int_print
print_char              printChar              char_print
print_double            printDouble            double_print

print_long_unsigned     printLongUnsigned      long_unsigned_print
print_unsigned_long     printUnsignedLong      unsigned_long_print
```

If there are many data types (and there will be), it is just not practical to give a different name to each function. And in C++ we don't have to. C++ can have several functions with the same name if their arguments are of different data types or if they have different numbers of arguments. The shared name is said to be *overloaded;* the functions that share the name are called the *overloads* of the name.

To make our example simpler, we can use the same name for all three `print` functions. Lines 18−21 call different functions, even though they have the same name.

Had there been a function whose argument was a `short`, line 21 would have called it. Since there isn't, it selects the `int print` rather than the `char print`. The computer prefers conversions that do not throw away information: promotions, rather than truncations.

The inner cast in line 34 prevents sign extension, like the cast in the previous program. The outer cast causes the `<<` to print the value as a decimal integer, not as a character.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/foverload/print.C`

```
 1 #include <iostream>   //C++ example
 2 #include <cstdlib>
 3 #include <cctype>     //for isprint
 4 #include <cmath>      //for frexp
 5 using namespace std;
 6
 7 void print(int i);
 8 void print(char c);
 9 void print(double d);
10
11 int main()
12 {
13     int i = 65;
14     char c = 'A';
15     double d = 65.0;
16     short s = 65;
17
18     print(i);    //the function declared in line 7
19     print(c);    //the function declared in line 8
20     print(d);    //the function declared in line 9
21     print(s);    //the function declared in line 7
22
23     return EXIT_SUCCESS;
24 }
25
26 void print(int i)
27 {
28     cout << i << "\n";
29 }
30
31 void print(char c)
32 {
33     //Output character code of c as a non-negative decimal integer.
34     cout << static_cast<unsigned>(static_cast<unsigned char>(c));
35     if (isprint(static_cast<unsigned char>(c)) != 0) {
36         cout << "\t'" << c << "'";
37     }
38     cout << "\n";
39 }
40
```

```
41 void print(double d)
42 {
43     int exponent;   //uninitialized variable
44     double mantissa = frexp(d, &exponent);
45
46     cout << d << ", mantissa == " << mantissa
47         << ", exponent == " << exponent << "\n";
48
49 }
```

```
65
65       'A'
65, mantissa == 0.507812, exponent == 7
65
```

Recall the functions `term_put` and `term_puts` in lines 15−16 of `term.h` on p. 86. They needed different names because they were written in C. To a C++ programmer, the different names would be an annoying redundancy. The arguments suffice to distinguish the functions.

If you remain unconvinced, consider the ‘‘absolute value’’ functions in the C Standard Library. Each one has to have a different name: `abs`, `labs`, `fabs`, with the recent addition of `llabs`, `fabsf`, and `fabsl`. Which one is for `float`? Which one is for `double`?

The compiler considers only the number and data types of the function's arguments, not the data type of the return value, when deciding which function to call:

```
 1 int f(int i);              //This pair is okay.
 2 int f(double d);
 3
 4 int g(int i);
 5 double g(int i);
 6
 7 int main()
 8 {
 9     int i = 10;
10     double d = 3.14159265358979323846;
11
12     f(i);            //the function declared in line 1
13     f(d);            //the function declared in line 2
14
15     g(i);            //won't compile: can't tell which g to call
```

The number of arguments, as well as the data type of the arguments, can distinguish two functions with the same name.

```
16 void f(int i);               //this pair is okay
17 void f(double d);
18
19 void g(int i);               //this pair is okay
20 void g(int i, int j);
21
22 void h(int i, double d);     //asking for trouble
23 void h(double d, int i);
24
25 int main()
26 {
27     int i = 10;
```

```
28     double d = 3.14159265358979323846;
29
30     f(i);                    //the function declared in line 16
31     f(d);                    //the function declared in line 17
32
33     g(i);                    //the function declared in line 19
34     g(i, i);                 //the function declared in line 20
35
36     h(i, d);                 //the function declared in line 22
37     h(d, i);                 //the function declared in line 23
38
39     h(i, i);                 //won't compile: can't tell which h to call
40     h(d, d);                 //won't compile: can't tell which h to call
```

Function name overloading plays three important rôles in the C++ Standard Library. It lets us do i/o without the % formats of `printf` and `scanf` (pp. 349–350). It lets us establish a different pair of memory allocation and deallocation functions for each data type (pp. 415–419). It is the hidden machinery for ''dispatching'' the different categories of iterators in the Standard Template Library (p. 915).

"Intersection of sets of functions"

> If a feature is so unusual or unclear that to understand it you need to consult a ''language lawyer''—an expert in reading language definitions—don't use it.
>
> —Brian W. Kernighan & Rob Pike, The Practice of Programming, p. 191

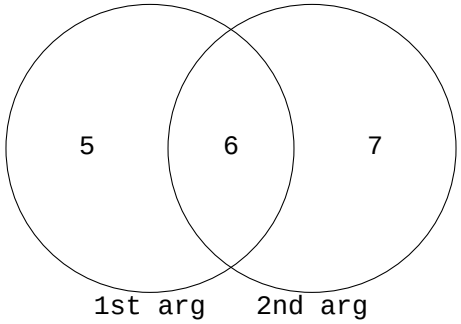
The rules for deciding which function to call are complicated. My advice is to use the same name only for functions whose arguments are so different that you never have to consult the rules.

Let's see how line 14 decide which function to call. First, the function in line 8 is disqualified because the second argument in line 14 cannot be converted to the data type `int *` without an explicit cast.

Then we find the function that best matches the first argument in line 14, or the ones that are tied for being the best match. The function in line 8 would be the best match, but it has already been disqualified. We settle for the functions in lines 5 and 6, and draw a circle around them. They are better than the one in line 7, which would require a `short`-to-`int` promotion followed by an `int`-to-`double` conversion.

Next we consider the second argument in line 14. It can be promoted to the `int` in lines 6–7, or truncated to the `char` in line 5. We prefer promotion because no information is lost, and indicate this by drawing the second circle.

For each argument, we form the set of functions that best match the argument. If the intersection of the sets is exactly one function, then that function is called. Otherwise, we get an error message.



—On the Web at
`http://i5.nyu.edu/~mm64/book/src/foverload/intersection.C`

```
1 #include <iostream>
2 #include <cstdlib>
```

```
 3 using namespace std;
 4
 5 void f(int i, char c);
 6 void f(int i, int j);
 7 void f(double d, int i);
 8 void f(short s, int *p);
 9
10 int main()
11 {
12     short s = 10;
13
14     f(s, s);   //call the function declared in line 6
15     return EXIT_SUCCESS;
16 }
17
18 void f(int i, char c)   {cout << "int char\n";}
19 void f(int i, int j)    {cout << "int int\n";}
20 void f(double d, int i) {cout << "double int\n";}
21 void f(short s, int *p) {cout << "short int *\n";}
```

```
int int
```

1.7.5 Default Values for Function Arguments

Default value for a function argument

Here is another example of function name overloading. The `oct`, `dec`, and `hex` in lines 23, 25, and 27 are i/o manipulators, like the `endl` on p. 26. They are invisible, but outputting them causes all subsequent integers output to the same destination to be written in the specified base (pp. 350−351).

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/default/overload.C`

```
 1 #include <iostream>   //for cout, cerr, <<, oct, dec, hex
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void print(int n, int base);
 6 void print(int n);
 7
 8 int main()
 9 {
10     int i = 255;
11
12     print(i, 10);   //the function in line 5
13     print(i, 8);    //the function in line 5
14     print(i, 16);   //the function in line 5
15     print(i);       //the function in line 6
16
17     return EXIT_SUCCESS;
18 }
19
20 void print(int n, int base)
21 {
```

```
22     if (base == 8) {
23         cout << oct << n << "\n";
24     } else if (base == 10) {
25         cout << dec << n << "\n";
26     } else if (base == 16) {
27         cout << hex << n << "\n";
28     } else {
29         cerr << "base " << base << " must be 8, 10, or 16\n";
30         exit(EXIT_FAILURE);
31     }
32 }
33
34 void print(int n)
35 {
36     print(n, 10);   //call-through to the function in line 20
37 }
```

The function in line 34 is merely a *call-through:* a function that does all its work by calling another one.

```
255          line 12: base 10
377          line 13: base 8
ff           line 14: base 16
255          line 15: base 10
```

But we do not have to bother with the call-through in the above line 34. A simpler way to get the same effect is to provide a default value for the last argument in the following line 5. The default value must be written in the function declaration in line 5, not in the function definition in line 19.

Only trailing arguments can have a default value. In other words, every argument with a default value must come to the right of every argument without a default value. When I write a new function, this influences the order in which I declare the arguments. I put an argument at the end of the list if I think it may have a default value in the future.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/default/default.C`

```
 1 #include <iostream>   //for cout and <<
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 void print(int n, int base = 10);   //only one print function
 6
 7 int main()
 8 {
 9     int i = 255;
10
11     print(i, 10);
12     print(i, 8);
13     print(i, 16);
14     print(i);
15
16     return EXIT_SUCCESS;
17 }
18
19 void print(int n, int base)
20 {
```

```
21     if (base == 8) {
22         cout << oct << n << "\n";
23     } else if (base == 10) {
24         cout << dec << n << "\n";
25     } else if (base == 16) {
26         cout << hex << n << "\n";
27     } else {
28         cerr << "base " << base << " must be 8, 10, or 16\n";
29         exit(EXIT_FAILURE);
30     }
31 }
```

```
255          line 11: base 10
377          line 12: base 8
ff           line 13: base 16
255          line 14: base 10
```

Bound at compile time, evaluated at runtime

The default value does not have to be a constant. If the default value is a variable or an expression containing variables, the variables are bound at compile time and evaluated at runtime.

In the following example, ‘‘bound at compile time’’ means that when the function declared in line 7 is called with one argument, the default argument will be the `default_base` whose declaration has been seen before line 7: the variable in line 6, not the one in line 12. ‘‘Evaluated at runtime’’ means that the default value used in line 14 will be the value of `default_base` as line 14 is executed: the value in line 11, not the one in line 6.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/default/bound.C`

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <cstdlib>
 4 using namespace std;
 5
 6 int default_base = 10;
 7 void print(int n, int base = default_base);
 8
 9 int main()
10 {
11     default_base = 16;       //Change value of variable declared in line 6.
12     int default_base = 8;    //Create another variable with the same name.
13
14     print(255); //variable declared in line 6, with value assigned in line 11.
15     return EXIT_SUCCESS;
16 }
17
18 void print(int n, int base)
19 {
20     if (base == 8) {
21         cout << oct << n << "\n";
22     } else if (base == 10) {
23         cout << dec << n << "\n";
24     } else if (base == 16) {
25         cout << hex << n << "\n";
```

```
26     } else {
27         cerr << "base " << base << " must be 8, 10, or 16\n";
28         exit(EXIT_FAILURE);
29     }
30 }
```

```
ff          base 16
```

1.7.6 Inline Functions

This program has a chunk of repeated code: lines 9 and 11 have the same computation for computing the average of two integers.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/inline/outline.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int main()
 6 {
 7     int i = 10;
 8     int j = 20;
 9     int k = (i + j) / 2;   //Let k be the average of i and j.
10
11     cout << (j + 40) / 2 << "\n";
12     return EXIT_SUCCESS;
13 }
```

```
30
```

One way to eliminate the repetition in the above lines 9 and 11 is with a macro. But macros are deprecated in C++ because they are so different from the rest of the language in their definition and usage.

For example, whitespace is optional in front of every left parentheses in C and C++ (p. 101). The one exception is in front of the first left parentheses in line 5, where whitespace is forbidden. If we had whitespace there, we would be defining a macro with no arguments.

For another example, it is never necessary to parenthesize an individual variable in C or C++. The one exception is in the *replacement text* of a macro definition—the string `(((a) + (b)) / 2)` in line 5. In a replacement text, each argument of the macro must be parenthesized. (These are the pairs around the `a` and `b`.) The entire replacement text must also be parenthesized if it consists of more than one token.

More macro anomalies are on pp. 649–652, where we consider another alternative to a macro: a ‘‘template function’’.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/inline/macro.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 #define AVERAGE(a, b) (((a) + (b)) / 2)
 6
 7 int main()
```

```
 8 {
 9     int i = 10;
10     int j = 20;
11     int k = AVERAGE(i, j);
12
13     cout << AVERAGE(j, 40) << "\n";
14     return EXIT_SUCCESS;
15 }
```

A function is a much nicer notation for eliminating the repetition:

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/inline/function.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 int average(int a, int b);  //function declaration
 6
 7 int main()
 8 {
 9     int i = 10;
10     int j = 20;
11     int k = average(i, j);
12
13     cout << average(j, 40) << "\n";
14     return EXIT_SUCCESS;
15 }
16
17 int average(int a, int b)   //function definition
18 {
19     return (a + b) / 2; //3 fewer pairs of parentheses than line 5 of macro.C
20 }
```

The macro has no effect on the executable program's size and speed, but the function makes it smaller (good) and slower (bad). Paradoxically, the loss of speed is most galling when the body of the function takes very little time. For example, imagine that it takes a millionth of a second to call the function, a millionth to execute its body, and a millionth to return. Then we're spending fully two-thirds of our time in transit.

To retain the original speed when using the function notation, make the function *inline.* Combine the function declaration and definition, write them where the declaration used to go, and add the keyword `inline`.

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/inline/inline.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 inline int average(int a, int b)   //function declaration and definition
 6 {
 7     return (a + b) / 2;
 8 }
 9
10 int main()
```

```
11 {
12     int i = 10;
13     int j = 20;
14     int k = average(i, j);
15
16     cout << average(j, 40) << "\n";
17     return EXIT_SUCCESS;
18 }
```

When we write the above program, the computer behaves as if we had written the original program `outline.C`. There are no longer any function calls or returns. It behaves as if we wrote the entire body of the inline function, `(a+b)/2`, in place of each call to the function.

In greater detail, line 14 of the above `inline.C` behaves as if we had written lines 19−22. The temporary variables `a`, `b`, and `retval` are destroyed after line 22.

```
19     int a = i;            //a and b are the arguments of the inline function
20     int b = j;
21     int retval = (a + b) / 2; //(a+b)/2 is the body of the inline function
22     int k = retval;
```

For extra speed, most computers would then ‘‘optimize’’ the `a`, `b`, and `retval` out of existence, leaving line 14 as if we had written

```
23     int k = (i + j) / 2;
```

Only a small function of one or two statements should be inline. If a ten-page function was inline and we called it in ten different places, we would be inserting 100 pages into our program. Remember that excessive size can slow down a program because of slower loading and more frequent paging.

Inline functions are static

A *global* variable is one that is defined outside the body of any function. If a global variable is defined (not merely declared) in a header file, we will get a ‘‘multiply defined’’ error if the header file is included in more than one `.C` file of the same program. We can fix this by declaring the variable to be static. The multiple definitions will still be there, but they won’t interfere with each other.

```
 1 //Declaration of non-static variable: can go in header file even if
 2 //the header file is included in more than one .C file of the same program.
 3 extern int i;
 4
 5 //Definition of non-static variable: cannot go in header file that is
 6 //included in more than one .C file ofthe same program.
 7 int i = 10;
 8
 9 //Definition of static variable: can go in header file even if
10 //the header file is included in more than one .C file of the same program.
11 static int i = 10;
```

The same rules apply to a function. If a non-static function is defined (not merely declared) in a header file, we will get a ‘‘multiply defined’’ error if the header file is included in more than one `.C` file of the same program. That’s why we usually write only the function declaration, not its definition, in a header file. But an inline function is static by default, so it can be defined in a header file. The multiple definitions will not interfere with each other.

An even simpler example of an inline function

—On the Web at
`http://i5.nyu.edu/~mm64/book/src/inline/stark.C`

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 using namespace std;
 4
 5 inline int maxwindowsize() {return 100;}
 6
 7 int main()
 8 {
 9     cout << maxwindowsize() << "\n";
10     return EXIT_SUCCESS;
11 }
```

The above line 9 behaves as if we had written

```
12     cout << 100 << "\n";
```

```
100
```

Why would anyone write a function that merely returns the value of a variable or literal? We'll see when we do classes and private data members.

1.8 Spacing and Indentation

Spacing rules

A group of one or more consecutive blanks, tabs, and/or newlines is called *whitespace.* There is one rule about where whitespace is prohibited, and one rule about where it is required. These rules are stated in terms of *tokens,* which are the words, numbers, quoted characters or strings, operators, or other punctuation marks that make up the source code of the program. The tokens fall into two groups, alphanumeric and non-alphanumeric. Here are examples of both kinds.

alphanumeric tokens	*non-alphanumeric tokens*
`main`	`10`	`+`	`[`
`int`	`010`	`-`	`]`
`void`	`0x10`	`.`	`?`
`const`	`10U`	`->`	`:`
`sizeof`	`10L`	`++`	`::`
`if`	`10UL`	`==`	`;`
`for`	`10.0`	`*=`	`,`
`typedef`	`10.0F`	`&=`	`{`
`extern`	`10.0L`	`&&`	`(`
`cout`	`10.0e5`	`<<`	`<`
`i`	`10.0e5F`	`<<=`	`'A'`
`my_func`	`10.0e5L`	`"a quoted string"`	

Here are the spacing rules.

(1) Whitespace is prohibited inside a token. Don't try to write

```
1 ma in
2 c out
3 < <
```

```
4 10 e 5
```

A comment delimiter is not a token. Even so, do not write

```
5 / /bad comment delimiter
6 / * bad opening comment delimiter */
7 /* bad closing comment delimiter * /
```

(2) Whitespace is required between two consecutive tokens that would otherwise be mistaken for a single token or a comment delimiter. There are three cases.

(2a) If the two tokens are alphanumeric, whitespace is always required between them. For example, whiespace is required between the `int` and the `i` in line 9. If we forget the whitespace, line 9 might still compile (thanks to line 8) but it would have a different meaning.

```
8 int inti = 10;
9 int i = 20;
```

(2b) If the two tokens are non-alphanumeric, whitespace is required between them only in the following exceptional combinations. The first four are present in both languages; the rest are new in C++.

With no space in line 10 between the ‘‘plus’’ and the ‘‘positive’’, the computer would read te `++` as an increment and the line would not compile. With no space in line 11 between the plus and the increment, the computer would read the `+++` as an increment and an addition: it always thinks that the leftmost token is the longest one. The line would still compile, but it would have a different meaning. With no space in line 14, the computer would think that the `//` was a comment delimiter. With no space in lines 15−16, the computer would think that the leftmost two colons were the global scope operator and the lines would not compile. The function declarations in lines 17−19 have a default value but no name for each argument. Without the space in line 17, the computer would read the `*=` as the multiply-and-assign operator.

```
10 a = b + +c;  //"b plus positive c"; ditto for "b minus negative c"
11
12 a = b + ++c; //the computer would read a = b +++c; as a = b ++ +c;
13 a = b & &c;
14 a = b / *p;

15 quotient = dividend / /* comment */ divisor;
16 label: ::f();               //unary :: is the global scope operator
17 a ? b : ::c;
18 void f(int * = 0);          //this argument is a pointer
19 void f(const int & = 0);    //this argument is a reference
20 void f(vector<int> = v);    //this argument is a template
21 vector<vector<int> > v;     //nested template
22 b = operator< <int>(10, 20); //explicit template argument
```

(2c) If the two tokens are alphanumeric and non-alphanumeric, in either order, whitespace is required between them in only one pathological case. See the definition of the macro with arguments in p. 97.

(3) White space is optional everywhere else: between two non-alphanumeric tokens (with the exceptions in ¶(2b)), or between an alphanumeric and a non-alphanumeric token (with the exception in ¶ (2c)). For example, both of the following are lexically correct:

```
23 if (i == j) {     //easy to read
24 if(i==j){         //harder to read
```

(4) The above rules do not apply to preprocessor lines: those that start with #. Type them exactly the way I do.

(5) Although the rules of C++ do not require it, be consistent or the user will think that the source code has been corrupted.

```
25      a + b       //good
26      a+b         //almost as good
27      a+ b        //annoying
28      a +b        //annoying
```

(6) Although the rules of C++ do not require it, space your punctuation as in English. Put a space after a comma, but none before it. Put no space before a semicolon. Put no space after a `(` or before a `)`.

Indentation rules for C or C++

A *control structure* is a `for` or `while` loop, `if` or `else`, etc. If the body of a control structure contains only one statement, the curly braces around it are optional. But pretend you never heard me say that. I want you to always write the pairs of `{ }` around the body of a control structure. The `{` goes at the end of the line immediately before the body, and the matching `}` goes at the start of the line immediately after the body. The `}` at the end of a `for`, `while`, and `if` should always be on a line by itself.

```
 1      for (;;) {
 2          body
 3      }
 4
 5      while () {
 6          body
 7      }
 8
 9      do {
10          body
11      } while ();
12
13      if () {
14          body
15      }
16
17      if () {
18          body
19      } else {
20          body
21      }
22
23      class c {
24          private members
25      public:
26          public members
27      };
```

(1) Do not indent the following lines at the beginning of every program.

```
1 int main()
2 {
```

Similarly, do not indent the first line of the definition of any other function nor the `{` line immediately below it.

```
3 void f()
4 {
```

(2) If a line ends with a `{`, then the following line should be indented one tab stop farther (e.g., lines 7−8, 8−9, 10−11, 15−16, 20−21 below). If a line begins with a `}`, then it should be indented one tab stop less than the previous line (e.g., lines 11−12, 12−13, 17−18, 21−22, 24−25). If neither of these rules apply,

simply indent the line to the same tab stop as the previous line (e.g., lines 9−10, 13−15, 16−17, 18−20, 22−24). If you did everything right, the } at the end of each function will not be indented.

```
 1 //Print the end of the Beatle's "She Loves You".
 2 #include <iostream>
 3 #include <cstdlib>
 4 using namespace std;
 5
 6 int main()
 7 {
 8     for (int i = 1; i <= 2; ++i) {
 9         cout << "She loves you\n";
10         for (int y = 1; y <= 3; ++y) {
11             cout << "Yeah!\n";
12         }
13     }
14
15     for (i = 1; i <= 3; ++i) {
16         cout << "With a love like that\n";
17         cout << "You know you should be glad.\n";
18     }
19
20     for (i = 1; i <= 10; ++i) {
21         cout << "Yeah!\n";
22     }
23
24     return EXIT_SUCCESS;
25 }
```

(3) The above rules for { and } cancel each other when writing an empty loop:

```
26     //Empty loop to waste time if you have no sleep function.
27     for (int i = 0; i <= 30000; ++i) {
28     }
```

(4) If a statement does not fit on one line, indent the continuation line(s) one more tab stop than the first line:

```
29         cout << "With a love like that\n"
30             << "You know you should be glad.\n";


31     cout << month << "/" << day << "/" << year << "\n";
32
33     cout << month << "/" << day << "/" << year
34         << " " << hour << ":" << minute << ":" << second << "\n";
35
36     cout << month << "/" << day << "/" << year
37         << " " << hour << ":" << minute << ":" << second
38         << " " << star_date << " " << warp_factor << "\n";
```

(5) Tab stops must be at equal intervals. Any distance is okay, as long as you use the same distance for each tab stop. *Indent with tabs, not blanks.*

```
39     for (int i = 0; i < 10; ++i) {                                  //good
40         for (int j = 0; j < 10; ++j) {
41             for (int k = 0; k < 10; ++k) {
```

```
42                  cout << i << ", " << j << ", " << k << "\n";
43              }
44          }
45      }


46      for (int i = 0; i < 10; ++i) {                                  //bad
47        for (int j = 0; j < 10; ++j) {
48              for (int k = 0; k < 10; ++k) {
49                          cout << i << ", " << j << ", " << k << "\n";
50              }
51        }
52      }
```