CHAPTER 4

EXPRESSIONS

CONTENTS

C++ provides a rich set of operators and defines what these operators do when applied to operands of built-in type. It also allows us to define the meaning of most of the operators when applied to operands of class types. This chapter focuses on the operators as defined in the language and applied to operands of built-in type. We will also look at some of the operators defined by the library. Chapter 14 will show how we can define operators for our own types.

An expression is composed of one or more **operands** and yields a **result** when it is evaluated. The simplest form of an **expression** is a single literal or variable. The result of such an expression is the value of the variable or literal. More complicated expressions are formed from an **operator** and one or more operands.

4.1 Fundamentals

There are a few fundamental concepts that affect how expressions are evaluated. We start by briefly discussing the concepts that apply to most (if not all) expressions. Subsequent sections will cover these topics in more detail.

4.1.1 Basic Concepts

There are both ***unary operators*** and ***binary operators***. Unary operators, such as address-of (`&`) and dereference (`*`), act on one operand. Binary operators, such as equality (`==`) and multiplication (`*`), act on two operands. There is also one ternary operator that takes three operands, and one operator, function call, that takes an unlimited number of operands.

Some symbols, such as `*`, are used as both a unary (dereference) and a binary (multiplication) operator. The context in which a symbol is used determines whether the symbol represents a unary or binary operator. The uses of such symbols are independent; it can be helpful to think of them as two different symbols.

Grouping Operators and Operands

Understanding expressions with multiple operators requires understanding the ***precedence*** and ***associativity*** of the operators and may depend on the ***order of evaluation*** of the operands. For example, the result of the following expression depends on how the operands are grouped to the operators:

```
5 + 10 * 20/2;
```

The operands to the `*` operator could be `10` and `20`, or `10` and `20/2`, or `15` and `20`, or `15` and `20/2`. Understanding such expressions is the topic of the next section.

Operand Conversions

As part of evaluating an expression, operands are often converted from one type to another. For example, the binary operators usually expect operands with the same type. These operators can be used on operands with differing types so long as the operands can be converted (§ 2.1.2, p. 35) to a common type.

Although the rules are somewhat complicated, for the most part conversions happen in unsurprising ways. For example, we can convert an integer to floating-point, and vice versa, but we cannot convert a pointer type to floating-point. What may be a bit surprising is that small integral type operands (e.g., `bool`, `char`, `short`, etc.) are generally **promoted** to a larger integral type, typically `int`. We'll look in detail at conversions in § 4.11 (p. 159).

Overloaded Operators

The language defines what the operators mean when applied to built-in and compound types. We can also define what most operators mean when applied to class types. Because such definitions give an alternative meaning to an existing operator symbol, we refer to them as **overloaded operators**. The IO library `>>` and `<<` operators and the operators we used with `string`s, `vector`s, and iterators are all overloaded operators.

When we use an overloaded operator, the meaning of the operator—including the type of its operand(s) and the result—depend on how the operator is defined. However, the number of operands and the precedence and the associativity of the operator cannot be changed.

Lvalues and Rvalues

Every expression in C++ is either an **rvalue** (pronounced "are-value") or an **lvalue** (pronounced "ell-value"). These names are inherited from C and originally had a simple mnemonic purpose: lvalues could stand on the left-hand side of an assignment whereas rvalues could not.

In C++, the distinction is less simple. In C++, an lvalue expression yields an object or a function. However, some lvalues, such as `const` objects, may not be the left-hand operand of an assignment. Moreover, some expressions yield objects but return them as rvalues, not lvalues. Roughly speaking, when we use an object as an rvalue, we use the object's value (its contents). When we use an object as an lvalue, we use the object's identity (its location in memory).

Operators differ as to whether they require lvalue or rvalue operands and as to whether they return lvalues or rvalues. The important point is that (with one exception that we'll cover in § 13.6 (p. 531)) we can use an lvalue when an rvalue is required, but we cannot use an rvalue when an lvalue (i.e., a location) is required. When we use an lvalue in place of an rvalue, the object's contents (its value) are used. We have already used several operators that involve lvalues.

- Assignment requires a (non`const`) lvalue as its left-hand operand and yields its left-hand operand as an lvalue.
- The address-of operator (§ 2.3.2, p. 52) requires an lvalue operand and returns a pointer to its operand as an rvalue.
- The built-in dereference and subscript operators (§ 2.3.2, p. 53, and § 3.5.2, p. 116) and the iterator dereference and `string` and `vector` subscript operators (§ 3.4.1, p. 106, § 3.2.3, p. 93, and § 3.3.3, p. 102) all yield lvalues.
- The built-in and iterator increment and decrement operators (§ 1.4.1, p. 12, and § 3.4.1, p. 107) require lvalue operands and the prefix versions (which are the ones we have used so far) also yield lvalues.

As we present the operators, we will note whether an operand must be an lvalue and whether the operator returns an lvalue.

Lvalues and rvalues also differ when used with `decltype` (§ 2.5.3, p. 70). When we apply `decltype` to an expression (other than a variable), the result is

a reference type if the expression yields an lvalue. As an example, assume `p` is an `int*`. Because dereference yields an lvalue, `decltype(*p)` is `int&`. On the other hand, because the address-of operator yields an rvalue, `decltype(&p)` is `int**`, that is, a pointer to a pointer to type `int`.

4.1.2 Precedence and Associativity

An expression with two or more operators is a **compound expression**. Evaluating a compound expression involves grouping the operands to the operators. Precedence and associativity determine how the operands are grouped. That is, they determine which parts of the expression are the operands for each of the operators in the expression. Programmers can override these rules by parenthesizing compound expressions to force a particular grouping.

In general, the value of an expression depends on how the subexpressions are grouped. Operands of operators with higher precedence group more tightly than operands of operators at lower precedence. Associativity determines how to group operands with the same precedence. For example, multiplication and division have the same precedence as each other, but they have higher precedence than addition. Therefore, operands to multiplication and division group before operands to addition and subtraction. The arithmetic operators are left associative, which means operators at the same precdence group left to right:

- Because of precedence, the expression `3+4*5` is `23`, not `35`.
- Because of associativity, the expression `20-15-3` is `2`, not `8`.

As a more complicated example, a left-to-right evaluation of the following expression yields 20:

```
6 + 3 * 4 / 2 + 2
```

Other imaginable results include 9, 14, and 36. In C++, the result is 14, because this expression is equivalent to

```
// parentheses in this expression match default precedence and associativity
((6 + ((3 * 4) / 2)) + 2)
```

Parentheses Override Precedence and Associativity

We can override the normal grouping with parentheses. Parenthesized expressions are evaluated by treating each parenthesized subexpression as a unit and otherwise applying the normal precedence rules. For example, we can parenthesize the expression above to force the result to be any of the four possible values:

```
// parentheses result in alternative groupings
cout << (6 + 3) * (4 / 2 + 2) << endl;      // prints 36
cout << ((6 + 3) * 4) / 2 + 2 << endl;      // prints 20
cout << 6 + 3 * 4 / (2 + 2) << endl;        // prints 9
```

When Precedence and Associativity Matter

We have already seen examples where precedence affects the correctness of our programs. For example, consider the discussion in § 3.5.3 (p. 120) about dereference and pointer arithmetic:

```
int ia[] = {0,2,4,6,8}; //  array with five elements of type int
int last = *(ia + 4);   //  initializes last to 8, the value of ia[4]
last = *ia + 4;          //  last = 4, equivalent to ia[0] + 4
```

If we want to access the element at the location `ia + 4`, then the parentheses around the addition are essential. Without parentheses, `*ia` is grouped first and `4` is added to the value in `*ia`.

The most common case that we've seen in which associativity matters is in input and output expressions. As we'll see in § 4.8 (p. 155), the operators used for IO are left associative. This associativity means we can combine several IO operations in a single expression:

```
cin >> v1 >> v2;   // read into v1 and then into v2
```

Table 4.12 (p. 166) lists all the operators organized into segments separated by double lines. Operators in each segment have the same precedence, and have higher precedence than operators in subsequent segments. For example, the prefix increment and dereference operators share the same precedence, which is higher than that of the arithmetic operators. The table includes a page reference to each operator's description. We have seen some of these operators already and will cover most of the rest in this chapter. However, there are a few operators that we will not cover until later.

Exercises Section 4.1.2

Exercise 4.1: What is the value returned by `5 + 10 * 20/2`?

Exercise 4.2: Using Table 4.12 (p. 166), parenthesize the following expressions to indicate the order in which the operands are grouped:

(a) `* vec.begin()` (b) `* vec.begin() + 1`

4.1.3 Order of Evaluation

Precedence specifies how the operands are grouped. It says nothing about the order in which the operands are evaluated. In most cases, the order is largely unspecified. In the following expression

```
int i = f1() * f2();
```

we know that `f1` and `f2` must be called before the multiplication can be done. After all, it is their results that are multiplied. However, we have no way of knowing whether `f1` will be called before `f2` or vice versa.

For operators that do not specify evaluation order, it is an error for an expression to *refer to and change* the same object. Expressions that do so have undefined behavior (§ 2.1.2, p. 36). As a simple example, the `<<` operator makes no guarantees about when or how its operands are evaluated. As a result, the following output expression is undefined:

```
int i = 0;
cout << i << " " << ++i << endl;  // undefined
```

Because this program is undefined, we cannot draw any conclusions about how it might behave. The compiler might evaluate `++i` before evaluating `i`, in which case the output will be `1 1`. Or the compiler might evaluate `i` first, in which case the output will be `0 1`. *Or the compiler might do something else entirely.* Because this expression has undefined behavior, the program is in error, regardless of what code the compiler generates.

There are four operators that do guarantee the order in which operands are evaluated. We saw in § 3.2.3 (p. 94) that the logical AND (`&&`) operator guarantees that its left-hand operand is evaluated first. Moreover, we are also guaranteed that the right-hand operand is evaluated only if the left-hand operand is `true`. The only other operators that guarantee the order in which operands are evaluated are the logical OR (`||`) operator (§ 4.3, p. 141), the conditional (`? :`) operator (§ 4.7, p. 151), and the comma (`,`) operator (§ 4.10, p. 157).

Order of Evaluation, Precedence, and Associativity

Order of operand evaluation is independent of precedence and associativity. In an expression such as `f() + g() * h() + j()`:

- Precedence guarantees that the results of `g()` and `h()` are multiplied.
- Associativity guarantees that the result of `f()` is added to the product of `g()` and `h()` and that the result of that addition is added to the value of `j()`.
- There are no guarantees as to the order in which these functions are called.

If `f`, `g`, `h`, and `j` are independent functions that do not affect the state of the same objects or perform IO, then the order in which the functions are called is irrelevant. If any of these functions do affect the same object, then the expression is in error and has undefined behavior.

EXERCISES SECTION 4.1.3

Exercise 4.3: Order of evaluation for most of the binary operators is left undefined to give the compiler opportunities for optimization. This strategy presents a trade-off between efficient code generation and potential pitfalls in the use of the language by the programmer. Do you consider that an acceptable trade-off? Why or why not?

ADVICE: MANAGING COMPOUND EXPRESSIONS

When you write compound expressions, two rules of thumb can be helpful:

1. **When in doubt, parenthesize expressions to force the grouping that the logic of your program requires.**
2. **If you change the value of an operand, don't use that operand elsewhere in the same expresion.**

An important exception to the second rule occurs when the subexpression that changes the operand is itself the operand of another subexpression. For example, in `*++iter`, the increment changes the value of `iter`. The (now changed) value of `iter` is the operand to the dereference operator. In this (and similar) expressions, order of evaluation isn't an issue. The increment (i.e., the subexpression that changes the operand) must be evaluated before the dereference can be evaluated. Such usage poses no problems and is quite common.

4.2 Arithmetic Operators

Table 4.1: Arithmetic Operators (Left Associative)

Operator	Function	Use
`+`	unary plus	`+ expr`
`-`	unary minus	`- expr`
`*`	multiplication	`expr * expr`
`/`	division	`expr / expr`
`%`	remainder	`expr % expr`
`+`	addition	`expr + expr`
`-`	subtraction	`expr - expr`

Table 4.1 (and the operator tables in subsequent sections) groups the operators by their precedence. The unary arithmetic operators have higher precedence than the multiplication and division operators, which in turn have higher precedence than the binary addition and subtraction operators. Operators of higher precedence group more tightly than do operators with lower precedence. These operators are all left associative, meaning that they group left to right when the precedence levels are the same.

Unless noted otherwise, the arithmetic operators may be applied to any of the arithmetic types (§ 2.1.1, p. 32) or to any type that can be converted to an arithmetic type. The operands and results of these operators are rvalues. As described in § 4.11 (p. 159), operands of small integral types are promoted to a larger integral type, and all operands may be converted to a common type as part of evaluating these operators.

The unary plus operator and the addition and subtraction operators may also be applied to pointers. § 3.5.3 (p. 119) covered the use of binary + and - with

pointer operands. When applied to a pointer or arithmetic value, unary plus returns a (possibly promoted) copy of the value of its operand.

The unary minus operator returns the result of negating a (possibly promoted) copy of the value of its operand:

```
int i = 1024;
int k = -i; // i is -1024
bool b = true;
bool b2 = -b; // b2 is true!
```

In § 2.1.1 (p. 34) we noted that `bool` values should not be used for computation. The result of `-b` is a good example of what we had in mind.

For most operators, operands of type `bool` are promoted to `int`. In this case, the value of `b` is `true`, which promotes to the `int` value `1` (§ 2.1.2, p. 35). That (promoted) value is negated, yielding `-1`. The value `-1` is converted back to `bool` and used to initialize `b2`. This initializer is a nonzero value, which when converted to `bool` is `true`. Thus, the value of `b2` is `true`!

Caution: Overflow and Other Arithmetic Exceptions

Some arithmetic expressions yield undefined results. Some of these undefined expressions are due to the nature of mathematics—for example, division by zero. Others are undefined due to the nature of computers—for example, due to overflow. Overflow happens when a value is computed that is outside the range of values that the type can represent.

Consider a machine on which `short`s are 16 bits. In that case, the maximum `short` is 32767. On such a machine, the following compound assignment overflows:

```
short short_value = 32767; // max value if shorts are 16 bits
short_value += 1; // this calculation overflows
cout << "short_value: " << short_value << endl;
```

The assignment to `short_value` is undefined. Representing a signed value of 32768 requires 17 bits, but only 16 are available. On many systems, there is *no* compile-time or run-time warning when an overflow occurs. As with any undefined behavior, what happens is unpredictable. On our system the program completes and writes

```
short_value: -32768
```

The value "wrapped around": The sign bit, which had been 0, was set to 1, resulting in a negative value. On another system, the result might be different, or the program might behave differently, including crashing entirely.

When applied to objects of arithmetic types, the arithmetic operators, `+`, `-`, `*`, and `/`, have their obvious meanings: addition, subtraction, multiplication, and division. Division between integers returns an integer. If the quotient contains a fractional part, it is truncated toward zero:

```
int ival1 = 21/6;   // ival1 is 3; result is truncated; remainder is discarded
int ival2 = 21/7;   // ival2 is 3; no remainder; result is an integral value
```

The % operator, known as the "remainder" or the "modulus" operator, computes the remainder that results from dividing the left-hand operand by the right-hand operand. The operands to % must have integral type:

```
int ival = 42;
double dval = 3.14;
ival % 12;    // ok: result is 6
ival % dval; // error: floating-point operand
```

In a division, a nonzero quotient is positive if the operands have the same sign and negative otherwise. Earlier versions of the language permitted a negative quotient to be rounded up or down; the new standard requires the quotient to be rounded toward zero (i.e., truncated). C++ 11

The modulus operator is defined so that if m and n are integers and n is nonzero, then (m/n)*n + m%n is equal to m. By implication, if m%n is nonzero, it has the same sign as m. Earlier versions of the language permitted m%n to have the same sign as n on implementations in which negative m/n was rounded away from zero, but such implementations are now prohibited. Moreover, except for the obscure case where -m overflows, (-m)/n and m/(-n) are always equal to -(m/n), m%(-n) is equal to m%n, and (-m)%n is equal to -(m%n). More concretely:

```
 21 % 6;    /* result is 3  */      21 / 6;    /* result is 3  */
 21 % 7;    /* result is 0  */      21 / 7;    /* result is 3  */
-21 % -8;   /* result is -5 */     -21 / -8;   /* result is 2  */
 21 % -5;   /* result is 1  */      21 / -5;   /* result is -4 */
```

Exercises Section 4.2

Exercise 4.4: Parenthesize the following expression to show how it is evaluated. Test your answer by compiling the expression (without parentheses) and printing its result.

```
12 / 3 * 4 + 5 * 15 + 24 % 4 / 2
```

Exercise 4.5: Determine the result of the following expressions.

(a) `-30 * 3 + 21 / 5` (b) `-30 + 3 * 21 / 5`
(c) `30 / 3 * 21 % 5` (d) `-30 / 3 * 21 % 4`

Exercise 4.6: Write an expression to determine whether an int value is even or odd.

Exercise 4.7: What does overflow mean? Show three expressions that will overflow.

4.3 Logical and Relational Operators

The relational operators take operands of arithmetic or pointer type; the logical operators take operands of any type that can be converted to bool. These operators all return values of type bool. Arithmetic and pointer operand(s) with a value of zero are false; all other values are true. The operands to these operators are rvalues and the result is an rvalue.

Table 4.2: Logical and Relational Operators			
Associativity	**Operator**	**Function**	**Use**
Right	`!`	logical NOT	`!expr`
Left	`<`	less than	`expr < expr`
Left	`<=`	less than or equal	`expr <= expr`
Left	`>`	greater than	`expr > expr`
Left	`>=`	greater than or equal	`expr >= expr`
Left	`==`	equality	`expr == expr`
Left	`!=`	inequality	`expr != expr`
Left	`&&`	logical AND	`expr && expr`
Left	`||`	logical OR	`expr || expr`

Logical AND and OR Operators

The overall result of the logical AND operator is `true` if and only if both its operands evaluate to `true`. The logical OR (`||`) operator evaluates as `true` if either of its operands evaluates as `true`.

The logical AND and OR operators always evaluate their left operand before the right. Moreover, the right operand is evaluated *if and only if* the left operand does not determine the result. This strategy is known as **short-circuit evaluation**:

- The right side of an `&&` is evaluated if and only if the left side is `true`.
- The right side of an `||` is evaluated if and only if the left side is `false`.

Several of the programs in Chapter 3 used the logical AND operator. Those programs used the left-hand operand to test whether it was safe to evaluate the right-hand operand. For example, the `for` condition on page 94:

```
index != s.size() && !isspace(s[index])
```

first checks that `index` has not reached the end of its associated `string`. We're guaranteed that the right operand won't be evaluated unless `index` is in range.

As an example that uses the logical OR, imagine we have some text in a `vector` of `strings`. We want to print the `strings`, adding a newline after each empty `string` or after a `string` that ends with a period. We'll use a range-based `for` loop (§ 3.2.3, p. 91) to process each element:

```
// note s as a reference to const; the elements aren't copied and can't be changed
for (const auto &s : text) { // for each element in text
    cout << s;          // print the current element
    // blank lines and those that end with a period get a newline
    if (s.empty() || s[s.size() - 1] == '.')
        cout << endl;
    else
        cout << " ";   // otherwise just separate with a space
}
```

After we print the current element, we check to see if we need to print a newline. The condition in the `if` first checks whether `s` is an empty `string`. If so, we need to print a newline regardless of the value of the right-hand operand. Only if the `string` is not empty do we evaluate the second expression, which checks whether the `string` ends with a period. In this expression, we rely on short-circuit evaluation of `||` to ensure that we subscript `s` only if `s` is not empty.

It is worth noting that we declared `s` as a reference to `const` (§ 2.5.2, p. 69). The elements in `text` are `string`s, and might be large. By making `s` a reference, we avoid copying the elements. Because we don't need to write to the elements, we made `s` a reference to `const`.

Logical NOT Operator

The logical NOT operator (`!`) returns the inverse of the truth value of its operand. We first used this operator in § 3.2.2 (p. 87). As another example, assuming `vec` is a `vector` of `int`s, we might use the logical NOT operator to see whether `vec` has elements by negating the value returned by `empty`:

```
// print the first element in vec if there is one
if (!vec.empty())
    cout << vec[0];
```

The subexpression

```
!vec.empty()
```

evaluates as `true` if the call to `empty` returns `false`.

The Relational Operators

The relational operators (`<`, `<=`, `>`, `<=`) have their ordinary meanings and return `bool` values. These operators are left associative.

Because the relational operators return `bool`s, the result of chaining these operators together is likely to be surprising:

```
// oops! this condition compares k to the bool result of i < j
if (i < j < k) //  true if k is greater than 1!
```

This condition groups `i` and `j` to the first `<` operator. The `bool` result of that expression is the left-hand operand of the second less-than operator. That is, `k` is compared to the `true`/`false` result of the first comparison! To accomplish the test we intended, we can rewrite the expression as follows:

```
// ok: condition is true if i is smaller than j and j is smaller than k
if (i < j && j < k) { /* ... */ }
```

Equality Tests and the `bool` Literals

If we want to test the truth value of an arithmetic or pointer object, the most direct way is to use the value as a condition:

```
if (val)  { /* ... */ } //  true if val is any nonzero value
if (!val) { /* ... */ } //  true if val is zero
```

In both conditions, the compiler converts `val` to `bool`. The first condition succeeds so long as `val` is nonzero; the second succeeds if `val` is zero.

We might think we could rewrite a test of this kind as

```
if (val == true)  { /* ... */ } //  true only if val is equal to 1!
```

There are two problems with this approach. First, it is longer and less direct than the previous code (although admittedly when first learning C++ this kind of abbreviation can be perplexing). Much more importantly, when `val` is not a `bool`, this comparison does not work as expected.

If `val` is not a `bool`, then `true` is converted to the type of `val` before the `==` operator is applied. That is, when `val` is not a `bool`, it is as if we had written

```
if (val == 1) { /* ... */ }
```

As we've seen, when a `bool` is converted to another arithmetic type, `false` converts to `0` and `true` converts to `1` (§ 2.1.2, p. 35). If we really cared whether `val` was the specific value `1`, we should write the condition to test that case directly.

It is usually a bad idea to use the boolean literals `true` and `false` as operands in a comparison. These literals should be used only to compare to an object of type `bool`.

Exercises Section 4.3

Exercise 4.8: Explain when operands are evaluated in the logical AND, logical OR, and equality operators.

Exercise 4.9: Explain the behavior of the condition in the following `if`:

```
const char *cp = "Hello World";
if (cp && *cp)
```

Exercise 4.10: Write the condition for a `while` loop that would read `int`s from the standard input and stop when the value read is equal to `42`.

Exercise 4.11: Write an expression that tests four values, `a`, `b`, `c`, and `d`, and ensures that `a` is greater than `b`, which is greater than `c`, which is greater than `d`.

Exercise 4.12: Assuming `i`, `j`, and `k` are all `int`s, explain what `i != j < k` means.

4.4 Assignment Operators

The left-hand operand of an assignment operator must be a modifiable lvalue. For example, given

```
int i = 0, j = 0, k = 0; //  initializations, not assignment
const int ci = i;        //  initialization, not assignment
```

Each of these assignments is illegal:

```
1024 = k;       // error: literals are rvalues
i + j = k;      // error: arithmetic expressions are rvalues
ci = k;         // error: ci is a const (nonmodifiable) lvalue
```

The result of an assignment is its left-hand operand, which is an lvalue. The type of the result is the type of the left-hand operand. If the types of the left and right operands differ, the right-hand operand is converted to the type of the left:

```
k = 0;          // result: type int, value 0
k = 3.14159;    // result: type int, value 3
```

Under the new standard, we can use a braced initializer list (§ 2.2.1, p. 43) on the right-hand side:

C++ 11

```
k = {3.14};                     // error: narrowing conversion
vector<int> vi;                 // initially empty
vi = {0,1,2,3,4,5,6,7,8,9};     // vi now has ten elements, values 0 through 9
```

If the left-hand operand is of a built-in type, the initializer list may contain at most one value, and that value must not require a narrowing conversion (§ 2.2.1, p. 43).

For class types, what happens depends on the details of the class. In the case of `vector`, the `vector` template defines its own version of an assignment operator that can take an initializer list. This operator replaces the elements of the left-hand side with the elements in the list on the right-hand side.

Regardless of the type of the left-hand operand, the initializer list may be empty. In this case, the compiler generates a value-initialized (§ 3.3.1, p. 98) temporary and assigns that value to the left-hand operand.

Assignment Is Right Associative

Unlike the other binary operators, assignment is right associative:

```
int ival, jval;
ival = jval = 0; // ok: each assigned 0
```

Because assignment is right associative, the right-most assignment, `jval = 0`, is the right-hand operand of the left-most assignment operator. Because assignment returns its left-hand operand, the result of the right-most assignment (i.e., `jval`) is assigned to `ival`.

Each object in a multiple assignment must have the same type as its right-hand neighbor or a type to which that neighbor can be converted (§ 4.11, p. 159):

```
int ival, *pval; // ival is an int; pval is a pointer to int
ival = pval = 0; // error: cannot assign the value of a pointer to an int
string s1, s2;
s1 = s2 = "OK";  // string literal "OK" converted to string
```

The first assignment is illegal because `ival` and `pval` have different types and there is no conversion from the type of `pval` (`int*`) to the type of `ival` (`int`). It is illegal even though zero is a value that can be assigned to either object.

On the other hand, the second assignment is fine. The string literal is converted to `string`, and that `string` is assigned to `s2`. The result of that assignment is `s2`, which has the same type as `s1`.

Assignment Has Low Precedence

Assignments often occur in conditions. Because assignment has relatively low precedence, we usually must parenthesize the assignment for the condition to work properly. To see why assignment in a condition is useful, consider the following loop. We want to call a function until it returns a desired value—say, 42:

```
// a verbose and therefore more error-prone way to write this loop
int i = get_value();  // get the first value
while (i != 42) {
    // do something . . .
    i = get_value();  // get remaining values
}
```

Here we start by calling `get_value` followed by a loop whose condition uses the value returned from that call. The last statement in this loop makes another call to `get_value`, and the loop repeats. We can write this code more directly as

```
int i;
// a better way to write our loop---what the condition does is now clearer
while ((i = get_value()) != 42) {
    // do something . . .
}
```

The condition now more clearly expresses our intent: We want to continue until `get_value` returns `42`. The condition executes by assigning the result returned by `get_value` to `i` and then comparing the result of that assignment with `42`.

Without the parentheses, the operands to `!=` would be the value returned from `get_value` and `42`. The `true` or `false` result of that test would be assigned to `i`—clearly not what we intended!

Because assignment has lower precedence than the relational operators, parentheses are usually needed around assignments in conditions.

Beware of Confusing Equality and Assignment Operators

The fact that we can use assignment in a condition can have surprising effects:

```
if (i = j)
```

The condition in this `if` assigns the value of `j` to `i` and then tests the result of the assignment. If `j` is nonzero, the condition will be `true`. The author of this code almost surely intended to test whether `i` and `j` have the same value:

```
if (i == j)
```

Bugs of this sort are notoriously difficult to find. Some, but not all, compilers are kind enough to warn about code such as this example.

Compound Assignment Operators

We often apply an operator to an object and then assign the result to that same object. As an example, consider the sum program from § 1.4.2 (p. 13):

```
int sum = 0;
// sum values from 1 through 10 inclusive
for (int val = 1; val <= 10; ++val)
    sum += val;  // equivalent to sum = sum + val
```

This kind of operation is common not just for addition but for the other arithmetic operators and the bitwise operators, which we cover in § 4.8 (p. 152). There are compound assignments for each of these operators:

```
 +=    -=    *=    /=    %=        // arithmetic operators
<<=   >>=   &=    ^=    |=        // bitwise operators; see § 4.8 (p. 152)
```

Each compound operator is essentially equivalent to

```
a = a op b;
```

with the exception that, when we use the compound assignment, the left-hand operand is evaluated only once. If we use an ordinary assignment, that operand is evaluated twice: once in the expression on the right-hand side and again as the operand on the left hand. In many, perhaps most, contexts this difference is immaterial aside from possible performance consequences.

EXERCISES SECTION 4.4

Exercise 4.13: What are the values of `i` and `d` after each assignment?

```
int i;   double d;
(a) d = i = 3.5;      (b) i = d = 3.5;
```

Exercise 4.14: Explain what happens in each of the `if` tests:

```
if (42 = i)   // ...
if (i = 42)   // ...
```

Exercise 4.15: The following assignment is illegal. Why? How would you correct it?

```
double dval; int ival; int *pi;
dval = ival = pi = 0;
```

Exercise 4.16: Although the following are legal, they probably do not behave as the programmer expects. Why? Rewrite the expressions as you think they should be.

```
(a) if (p = getPtr() != 0)     (b) if (i = 1024)
```

4.5 Increment and Decrement Operators

The increment (++) and decrement (--) operators provide a convenient notational shorthand for adding or subtracting 1 from an object. This notation rises above

mere convenience when we use these operators with iterators, because many iterators do not support arithmetic.

There are two forms of these operators: prefix and postfix. So far, we have used only the prefix form. This form increments (or decrements) its operand and yields the *changed* object as its result. The postfix operators increment (or decrement) the operand but yield a copy of the original, *unchanged* value as its result:

```
int i = 0, j;
j = ++i; //  j = 1, i = 1: prefix yields the incremented value
j = i++; //  j = 1, i = 2: postfix yields the unincremented value
```

These operators require lvalue operands. The prefix operators return the object itself as an lvalue. The postfix operators return a copy of the object's original value as an rvalue.

ADVICE: USE POSTFIX OPERATORS ONLY WHEN NECESSARY

Readers from a C background might be surprised that we use the prefix increment in the programs we've written. The reason is simple: The prefix version avoids unnecessary work. It increments the value and returns the incremented version. The postfix operator must store the original value so that it can return the unincremented value as its result. If we don't need the unincremented value, there's no need for the extra work done by the postfix operator.

For `int`s and pointers, the compiler can optimize away this extra work. For more complicated iterator types, this extra work potentially might be more costly. By habitually using the prefix versions, we do not have to worry about whether the performance difference matters. Moreover—and perhaps more importantly—we can express the intent of our programs more directly.

Combining Dereference and Increment in a Single Expression

The postfix versions of `++` and `--` are used when we want to use the current value of a variable and increment it in a single compound expression.

As one example, we can use postfix increment to write a loop to print the values in a `vector` up to but not including the first negative value:

```
auto pbeg = v.begin();
//  print elements up to the first negative value
while (pbeg != v.end() && *beg >= 0)
    cout << *pbeg++ << endl; //  print the current value and advance pbeg
```

The expression `*pbeg++` is usually confusing to programmers new to both C++ and C. However, because this usage pattern is so common, C++ programmers must understand such expressions.

The precedence of postfix increment is higher than that of the dereference operator, so `*pbeg++` is equivalent to `*(pbeg++)`. The subexpression `pbeg++` increments `pbeg` and yields a copy of the previous value of `pbeg` as its result. Accordingly, the operand of `*` is the unincremented value of `pbeg`. Thus, the statement prints the element to which `pbeg` originally pointed and increments `pbeg`.

This usage relies on the fact that postfix increment returns a copy of its original, unincremented operand. If it returned the incremented value, we'd dereference the incremented value, with disastrous results. We'd skip the first element. Worse, if the sequence had no negative values, we would attempt to dereference one too many elements.

ADVICE: BREVITY CAN BE A VIRTUE

Expressions such as `*pbeg++` can be bewildering—at first. However, it is a useful and widely used idiom. Once the notation is familiar, writing

```
cout << *iter++ << endl;
```

is easier and less error-prone than the more verbose equivalent

```
cout << *iter << endl;
++iter;
```

It is worthwhile to study examples of such code until their meanings are immediately clear. Most C++ programs use succinct expressions rather than more verbose equivalents. Therefore, C++ programmers must be comfortable with such usages. Moreover, once these expressions are familiar, you will find them less error-prone.

Remember That Operands Can Be Evaluated in Any Order

Most operators give no guarantee as to the order in which operands will be evaluated (§ 4.1.3, p. 137). This lack of guaranteed order often doesn't matter. The cases where it does matter are when one subexpression changes the value of an operand that is used in another subexpression. Because the increment and decrement operators change their operands, it is easy to misuse these operators in compound expressions.

To illustrate the problem, we'll rewrite the loop from § 3.4.1 (p. 108) that capitalizes the first word in the input. That example used a `for` loop:

```
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it); //  capitalize the current character
```

which allowed us to separate the statement that dereferenced `beg` from the one that incremented it. Replacing the `for` with a seemingly equivalent `while`

```
//  the behavior of the following loop is undefined!
while (beg != s.end() && !isspace(*beg))
    *beg = toupper(*beg++);    //  error: this assignment is undefined
```

results in undefined behavior. The problem is that in the revised version, both the left- and right-hand operands to `=` use `beg` *and* the right-hand operand changes `beg`. The assignment is therefore undefined. The compiler might evaluate this expression as either

```
*beg = toupper(*beg);          //  execution if left-hand side is evaluated first
*(beg + 1) = toupper(*beg);    //  execution if right-hand side is evaluated first
```

or it might evaluate it in yet some other way.

Exercises Section 4.5

Exercise 4.17: Explain the difference between prefix and postfix increment.

Exercise 4.18: What would happen if the `while` loop on page 148 that prints the elements from a `vector` used the prefix increment operator?

Exercise 4.19: Given that `ptr` points to an `int`, that `vec` is a `vector<int>`, and that `ival` is an `int`, explain the behavior of each of these expressions. Which, if any, are likely to be incorrect? Why? How might each be corrected?

(a) `ptr != 0 && *ptr++` (b) `ival++ && ival`
(c) `vec[ival++] <= vec[ival]`

4.6 The Member Access Operators

The dot (§ 1.5.2, p. 23) and arrow (§ 3.4.1, p. 110) operators provide for member access. The dot operator fetches a member from an object of class type; arrow is defined so that *ptr*`->`*mem* is a synonym for `(*`*ptr*`).`*mem*:

```
string s1 = "a string", *p = &s1;
auto n = s1.size(); // run the size member of the string s1
n = (*p).size();    // run size on the object to which p points
n = p->size();      // equivalent to (*p).size()
```

Because dereference has a lower precedence than dot, we must parenthesize the dereference subexpression. If we omit the parentheses, this code means something quite different:

```
// run the size member of p, then dereference the result!
*p.size();      // error: p is a pointer and has no member named size
```

This expression attempts to fetch the `size` member of the object `p`. However, `p` is a pointer, which has no members; this code will not compile.

The arrow operator requires a pointer operand and yields an lvalue. The dot operator yields an lvalue if the object from which the member is fetched is an lvalue; otherwise the result is an rvalue.

Exercises Section 4.6

Exercise 4.20: Assuming that `iter` is a `vector<string>::iterator`, indicate which, if any, of the following expressions are legal. Explain the behavior of the legal expressions and why those that aren't legal are in error.

(a) `*iter++;` (b) `(*iter)++;` (c) `*iter.empty()`
(d) `iter->empty();` (e) `++*iter;` (f) `iter++->empty();`

4.7 The Conditional Operator

The conditional operator (the **`?:` operator**) lets us embed simple if-else logic inside an expression. The conditional operator has the following form:

```
cond ? expr1 : expr2;
```

where *cond* is an expression that is used as a condition and *expr1* and *expr2* are expressions of the same type (or types that can be converted to a common type). This operator executes by evaluating *cond*. If the condition is `true`, then *expr1* is evaluated; otherwise, *expr2* is evaluated. As one example, we can use a conditional operator to determine whether a grade is pass or fail:

```
string finalgrade = (grade < 60) ?  "fail" : "pass";
```

The condition checks whether `grade` is less than `60`. If so, the result of the expression is `"fail"`; otherwise the result is `"pass"`. Like the logical AND and logical OR (`&&` and `||`) operators, the conditional operator guarantees that only one of *expr1* or *expr2* is evaluated.

That result of the conditional operator is an lvalue if both expressions are lvalues or if they convert to a common lvalue type. Otherwise the result is an rvalue.

Nesting Conditional Operations

We can nest one conditional operator inside another. That is, the conditional operator can be used as the *cond* or as one or both of the *expr*s of another conditional expression. As an example, we'll use a pair of nested conditionals to perform a three-way test to indicate whether a grade is a high pass, an ordinary pass, or fail:

```
finalgrade = (grade > 90) ? "high pass"
                          : (grade < 60) ? "fail" : "pass";
```

The first condition checks whether the grade is above `90`. If so, the expression after the `?` is evaluated, which yields `"high pass"`. If the condition fails, the `:` branch is executed, which is itself another conditional expression. This conditional asks whether the `grade` is less than `60`. If so, the `?` branch is evaluated and yields `"fail"`. If not, the `:` branch returns `"pass"`.

The conditional operator is right associative, meaning (as usual) that the operands group right to left. Associativity accounts for the fact that the right-hand conditional—the one that compares `grade` to `60`—forms the `:` branch of the left-hand conditional expression.

Nested conditionals quickly become unreadable. It's a good idea to nest no more than two or three.

Using a Conditional Operator in an Output Expression

The conditional operator has fairly low precedence. When we embed a conditional expression in a larger expression, we usually must parenthesize the conditional subexpression. For example, we often use the conditional operator to print one or

another value, depending on the result of a condition. An incompletely parenthesized conditional operator in an output expression can have surprising results:

```
cout << ((grade < 60) ?  "fail" : "pass"); //  prints pass or fail
cout << (grade < 60) ?  "fail" : "pass";   //  prints 1 or 0!
cout << grade < 60 ?  "fail" : "pass"; //  error: compares cout to 60
```

The second expression uses the comparison between `grade` and `60` as the operand to the `<<` operator. The value `1` or `0` is printed, depending on whether `grade < 60` is true or false. The `<<` operator returns `cout`, which is tested as the condition for the conditional operator. That is, the second expression is equivalent to

```
cout << (grade < 60);     //  prints 1 or 0
cout ?  "fail" : "pass"; //  test cout and then yield one of the two literals
                         //  depending on whether cout is true or false
```

The last expression is an error because it is equivalent to

```
cout << grade;    //  less-than has lower precedence than shift, so print grade first
cout < 60 ? "fail" : "pass"; //  then compare cout to 60!
```

Exercises Section 4.7

Exercise 4.21: Write a program to use a conditional operator to find the elements in a `vector<int>` that have odd value and double the value of each such element.

Exercise 4.22: Extend the program that assigned high pass, pass, and fail grades to also assign low pass for grades between 60 and 75 inclusive. Write two versions: One version that uses only conditional operators; the other should use one or more `if` statements. Which version do you think is easier to understand and why?

Exercise 4.23: The following expression fails to compile due to operator precedence. Using Table 4.12 (p. 166), explain why it fails. How would you fix it?

```
string s = "word";
string pl = s + s[s.size() - 1] == 's' ? "" : "s" ;
```

Exercise 4.24: Our program that distinguished between high pass, pass, and fail depended on the fact that the conditional operator is right associative. Describe how that operator would be evaluated if the operator were left associative.

4.8 The Bitwise Operators

The bitwise operators take operands of integral type that they use as a collection of bits. These operators let us test and set individual bits. As we'll see in § 17.2 (p. 723), we can also use these operators on a library type named `bitset` that represents a flexibly sized collection of bits.

As usual, if an operand is a "small integer," its value is first promoted (§ 4.11.1, p. 160) to a larger integral type. The operand(s) can be either signed or unsigned.

Table 4.3: Bitwise Operators (Left Associative)

Operator	Function	Use
~	bitwise NOT	`~expr`
<<	left shift	`expr1 << expr2`
>>	right shift	`expr1 >> expr2`
&	bitwise AND	`expr1 & expr2`
^	bitwise XOR	`expr1 ^ expr2`
|	bitwise OR	`expr1 | expr2`

If the operand is signed and its value is negative, then the way that the "sign bit" is handled in a number of the bitwise operations is machine dependent. Moreover, doing a left shift that changes the value of the sign bit is undefined.

Because there are no guarantees for how the sign bit is handled, we strongly recommend using `unsigned` types with the bitwise operators.

Bitwise Shift Operators

We have already used the overloaded versions of the >> and << operators that the IO library defines to do input and output. The built-in meaning of these operators is that they perform a bitwise shift on their operands. They yield a value that is a copy of the (possibly promoted) left-hand operand with the bits shifted as directed by the right-hand operand. The right-hand operand must not be negative and must be a value that is strictly less than the number of bits in the result. Otherwise, the operation is undefined. The bits are shifted left (<<) or right (>>). Bits that are shifted off the end are discarded:

These illustrations have the low-order bit on the right
These examples assume `char` has 8 bits, and `int` has 32

```
// 0233 is an octal literal (§ 2.1.3, p. 38)
unsigned char bits = 0233;
```

1	0	0	1	1	0	1	1

`bits << 8` *// `bits` promoted to `int` and then shifted left by 8 bits*

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0

`bits << 31` *// left shift 31 bits, left-most bits discarded*

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

`bits >> 3` *// right shift 3 bits, 3 right-most bits discarded*

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1

The left-shift operator (the **<< operator**) inserts 0-valued bits on the right. The behavior of the right-shift operator (the **>> operator**) depends on the type of the left-hand operand: If that operand is `unsigned`, then the operator inserts 0-valued

bits on the left; if it is a signed type, the result is implementation defined—either copies of the sign bit or 0-valued bits are inserted on the left.

Bitwise NOT Operator

The bitwise NOT operator (the **~operator**) generates a new value with the bits of its operand inverted. Each 1 bit is set to 0; each 0 bit is set to 1:

```
unsigned char bits = 0227;   1 0 0 1 0 1 1 1

~bits
1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1 | 0 1 1 0 1 0 0 0
```

Here, our `char` operand is first promoted to `int`. Promoting a `char` to `int` leaves the value unchanged but adds 0 bits to the high order positions. Thus, promoting `bits` to `int` adds 24 high order bits, all of which are 0-valued. The bits in the promoted value are inverted.

Bitwise AND, OR, and XOR Operators

The AND (&), OR (|), and XOR (^) operators generate new values with the bit pattern composed from its two operands:

```
unsigned char b1 = 0145;        0 1 1 0 0 1 0 1

unsigned char b2 = 0257;        1 0 1 0 1 1 1 1

   b1 & b2   24 high-order bits all 0   0 0 1 0 0 1 0 1

   b1 | b2   24 high-order bits all 0   1 1 1 0 1 1 1 1

   b1 ^ b2   24 high-order bits all 0   1 1 0 0 1 0 1 0
```

For each bit position in the result of the bitwise AND operator (the **& operator**) the bit is 1 if both operands contain 1; otherwise, the result is 0. For the OR (inclusive or) operator (the | **operator**), the bit is 1 if either or both operands contain 1; otherwise, the result is 0. For the XOR (exclusive or) operator (the ^ **operator**), the bit is 1 if either but not both operands contain 1; otherwise, the result is 0.

It is a common error to confuse the bitwise and logical operators (§ 4.3, p. 141). For example to confuse the bitwise `&` with the logical `&&`, the bitwise `|` with the logical `||`, and the bitwise `~`and the logical `!`).

Using Bitwise Operators

As an example of using the bitwise operators let's assume a teacher has 30 students in a class. Each week the class is given a pass/fail quiz. We'll track the results of each quiz using one bit per student to represent the pass or fail grade on a given test. We might represent each quiz in an `unsigned` integral value:

```
unsigned long quiz1 = 0;  // we'll use this value as a collection of bits
```

We define `quiz1` as an `unsigned long`. Thus, `quiz1` will have at least 32 bits on any machine. We explicitly initialize `quiz1` to ensure that the bits start out with well-defined values.

The teacher must be able to set and test individual bits. For example, we'd like to be able to set the bit corresponding to student number 27 to indicate that this student passed the quiz. We can indicate that student number 27 passed by creating a value that has only bit 27 turned on. If we then bitwise OR that value with `quiz1`, all the bits except bit 27 will remain unchanged.

For the purpose of this example, we will count the bits of `quiz1` by assigning 0 to the low-order bit, 1 to the next bit, and so on.

We can obtain a value indicating that student 27 passed by using the left-shift operator and an `unsigned long` integer literal 1 (§ 2.1.3, p. 38):

```
1UL << 27   // generate a value with only bit number 27 set
```

`1UL` has a 1 in the low-order bit and (at least) 31 zero bits. We specified `unsigned long` because `int`s are only guaranteed to have 16 bits, and we need at least 27. This expression shifts the 1 bit left 27 positions inserting 0 bits behind it.

Next we OR this value with `quiz1`. Because we want to update the value of `quiz1`, we use a compound assignment (§ 4.4, p. 147):

```
quiz1 |= 1UL << 27;   // indicate student number 27 passed
```

The `|=` operator executes analogously to how `+=` does. It is equivalent to

```
quiz1 = quiz1 | 1UL << 27;   // equivalent to quiz1 |= 1UL << 27;
```

Imagine that the teacher reexamined the quiz and discovered that student 27 actually had failed the test. The teacher must now turn off bit 27. This time we need an integer that has bit 27 turned off and all the other bits turned on. We'll bitwise AND this value with `quiz1` to turn off just that bit:

```
quiz1 &= ~(1UL << 27);   // student number 27 failed
```

We obtain a value with all but bit 27 turned on by inverting our previous value. That value had 0 bits in all but bit 27, which was a 1. Applying the bitwise NOT to that value will turn off bit 27 and turn on all the others. When we bitwise AND this value with `quiz1`, all except bit 27 will remain unchanged.

Finally, we might want to know how the student at position 27 fared:

```
bool status = quiz1 & (1UL << 27);   // how did student number 27 do?
```

Here we AND a value that has bit 27 turned on with `quiz1`. The result is nonzero (i.e., true) if bit 27 of `quiz1` is also on; otherwise, it evaluates to zero.

Shift Operators (aka IO Operators) Are Left Associative

Although many programmers never use the bitwise operators directly, most programmers do use overloaded versions of these operators for IO. An overloaded operator has the same precedence and associativity as the built-in version of that operator. Therefore, programmers need to understand the precedence and associativity of the shift operators even if they never use them with their built-in meaning.

Because the shift operators are left associative, the expression

```
cout << "hi" << " there" << endl;
```

executes as

```
( (cout << "hi") << " there" ) << endl;
```

In this statement, the operand `"hi"` is grouped with the first `<<` symbol. Its result is grouped with the second, and then that result is grouped with the third.

The shift operators have midlevel precedence: lower than the arithmetic operators but higher than the relational, assignment, and conditional operators. These relative precedence levels mean we usually have to use parentheses to force the correct grouping of operators with lower precedence.

```
cout << 42 + 10;    // ok: + has higher precedence, so the sum is printed
cout << (10 < 42); // ok: parentheses force intended grouping; prints 1
cout << 10 < 42;    // error: attempt to compare cout to 42!
```

The last `cout` is interpreted as

```
(cout << 10) < 42;
```

which says to "write `10` onto `cout` and then compare the result of that operation (i.e., `cout`) to `42`."

Exercises Section 4.8

Exercise 4.25: What is the value of `~'q' << 6` on a machine with 32-bit `int`s and 8 bit `char`s, that uses Latin-1 character set in which `'q'` has the bit pattern `01110001`?

Exercise 4.26: In our grading example in this section, what would happen if we used `unsigned int` as the type for `quiz1`?

Exercise 4.27: What is the result of each of these expressions?

```
unsigned long ul1 = 3, ul2 = 7;
(a)  ul1 & ul2            (b)  ul1 | ul2
(c)  ul1 && ul2           (d)  ul1 || ul2
```

4.9 The `sizeof` Operator

The **`sizeof`** operator returns the size, in bytes, of an expression or a type name. The operator is right associative. The result of `sizeof` is a constant expression (§ 2.4.4, p. 65) of type `size_t` (§ 3.5.2, p. 116). The operator takes one of two forms:

```
sizeof (type)
sizeof expr
```

In the second form, `sizeof` returns the size of the type returned by the given expression. The `sizeof` operator is unusual in that it does not evaluate its operand:

```
Sales_data data, *p;
sizeof(Sales_data); //  size required to hold an object of type Sales_data
sizeof data; //  size of data's type, i.e., sizeof(Sales_data)
sizeof p;    //  size of a pointer
sizeof *p;   //  size of the type to which p points, i.e., sizeof(Sales_data)
sizeof data.revenue; //  size of the type of Sales_data's revenue member
sizeof Sales_data::revenue; //  alternative way to get the size of revenue
```

The most interesting of these examples is `sizeof *p`. First, because `sizeof` is right associative and has the same precedence as `*`, this expression groups right to left. That is, it is equivalent to `sizeof (*p)`. Second, because `sizeof` does not evaluate its operand, it doesn't matter that `p` is an invalid (i.e., uninitialized) pointer (§ 2.3.2, p. 52). Dereferencing an invalid pointer as the operand to `sizeof` is safe because the pointer is not actually used. `sizeof` doesn't need dereference the pointer to know what type it will return.

Under the new standard, we can use the scope operator to ask for the size of a member of a class type. Ordinarily we can only access the members of a class through an object of that type. We don't need to supply an object, because `sizeof` does not need to fetch the member to know its size.

C++ 11

The result of applying `sizeof` depends in part on the type involved:

- `sizeof char` or an expression of type `char` is guaranteed to be 1.
- `sizeof` a reference type returns the size of an object of the referenced type.
- `sizeof` a pointer returns the size needed hold a pointer.
- `sizeof` a dereferenced pointer returns the size of an object of the type to which the pointer points; the pointer need not be valid.
- `sizeof` an array is the size of the entire array. It is equivalent to taking the `sizeof` the element type times the number of elements in the array. Note that `sizeof` does not convert the array to a pointer.
- `sizeof` a `string` or a `vector` returns only the size of the fixed part of these types; it does not return the size used by the object's elements.

Because `sizeof` returns the size of the entire array, we can determine the number of elements in an array by dividing the array size by the element size:

```
//  sizeof(ia)/sizeof(*ia) returns the number of elements in ia
constexpr size_t sz = sizeof(ia)/sizeof(*ia);
int arr2[sz];  //  ok sizeof returns a constant expression § 2.4.4 (p. 65)
```

Because `sizeof` returns a constant expression, we can use the result of a `sizeof` expression to specify the dimension of an array.

4.10 Comma Operator

The **comma operator** takes two operands, which it evaluates from left to right. Like the logical AND and logical OR and the conditional operator, the comma operator guarantees the order in which its operands are evaluated.

Exercises Section 4.9

Exercise 4.28: Write a program to print the size of each of the built-in types.

Exercise 4.29: Predict the output of the following code and explain your reasoning. Now run the program. Is the output what you expected? If not, figure out why.

```
int x[10];   int *p = x;
cout << sizeof(x)/sizeof(*x) << endl;
cout << sizeof(p)/sizeof(*p) << endl;
```

Exercise 4.30: Using Table 4.12 (p. 166), parenthesize the following expressions to match the default evaluation:

(a) `sizeof x + y` (b) `sizeof p->mem[i]`
(c) `sizeof a < b` (d) `sizeof f()`

The left-hand expression is evaluated and its result is discarded. The result of a comma expression is the value of its right-hand expression. The result is an lvalue if the right-hand operand is an lvalue.

One common use for the comma operator is in a `for` loop:

```
vector<int>::size_type cnt = ivec.size();
// assign values from size... 1 to the elements in ivec
for(vector<int>::size_type ix = 0;
                ix != ivec.size(); ++ix, --cnt)
    ivec[ix] = cnt;
```

This loop increments `ix` and decrements `cnt` in the expression in the `for` header. Both `ix` and `cnt` are changed on each trip through the loop. As long as the test of `ix` succeeds, we reset the current element to the current value of `cnt`.

Exercises Section 4.10

Exercise 4.31: The program in this section used the prefix increment and decrement operators. Explain why we used prefix and not postfix. What changes would have to be made to use the postfix versions? Rewrite the program using postfix operators.

Exercise 4.32: Explain the following loop.

```
constexpr int size = 5;
int ia[size] = {1,2,3,4,5};
for (int *ptr = ia, ix = 0;
     ix != size && ptr != ia+size;
     ++ix, ++ptr)   { /* ... */ }
```

Exercise 4.33: Using Table 4.12 (p. 166) explain what the following expression does:

```
someValue ? ++x, ++y : --x, --y
```

4.11 Type Conversions

In C++ some types are related to each other. When two types are related, we can use an object or value of one type where an operand of the related type is expected. Two types are related if there is a **conversion** between them.

As an example, consider the following expression, which initializes `ival` to 6:

```
int ival = 3.541 + 3; // the compiler might warn about loss of precision
```

The operands of the addition are values of two different types: `3.541` has type `double`, and `3` is an `int`. Rather than attempt to add values of the two different types, C++ defines a set of conversions to transform the operands to a common type. These conversions are carried out automatically without programmer intervention—and sometimes without programmer knowledge. For that reason, they are referred to as **implicit conversions**.

The implicit conversions among the arithmetic types are defined to preserve precision, if possible. Most often, if an expression has both integral and floating-point operands, the integer is converted to floating-point. In this case, `3` is converted to `double`, floating-point addition is done, and the result is a `double`.

The initialization happens next. In an initialization, the type of the object we are initializing dominates. The initializer is converted to the object's type. In this case, the `double` result of the addition is converted to `int` and used to initialize `ival`. Converting a `double` to an `int` truncates the `double`'s value, discarding the decimal portion. In this expression, the value `6` is assigned to `ival`.

When Implicit Conversions Occur

The compiler automatically converts operands in the following circumstances:

- In most expressions, values of integral types smaller than `int` are first promoted to an appropriate larger integral type.
- In conditions, nonbool expressions are converted to `bool`.
- In initializations, the initializer is converted to the type of the variable; in assignments, the right-hand operand is converted to the type of the left-hand.
- In arithmetic and relational expressions with operands of mixed types, the types are converted to a common type.
- As we'll see in Chapter 6, conversions also happen during function calls.

4.11.1 The Arithmetic Conversions

The **arithmetic conversions**, which we introduced in § 2.1.2 (p. 35), convert one arithmetic type to another. The rules define a hierarchy of type conversions in which operands to an operator are converted to the widest type. For example, if one operand is of type `long double`, then the other operand is converted to type `long double` regardless of what the second type is. More generally, in expressions that mix floating-point and integral values, the integral value is converted to an appropriate floating-point type.

Integral Promotions

The **integral promotions** convert the small integral types to a larger integral type. The types `bool`, `char`, `signed char`, `unsigned char`, `short`, and `unsigned short` are promoted to `int` if all possible values of that type fit in an `int`. Otherwise, the value is promoted to `unsigned int`. As we've seen many times, a `bool` that is `false` promotes to 0 and `true` to 1.

The larger `char` types (`wchar_t`, `char16_t`, and `char32_t`) are promoted to the smallest type of `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long` in which all possible values of that character type fit.

Operands of Unsigned Type

If the operands of an operator have differing types, those operands are ordinarily converted to a common type. If any operand is an `unsigned` type, the type to which the operands are converted depends on the relative sizes of the integral types on the machine.

As usual, integral promotions happen first. If the resulting type(s) match, no further conversion is needed. If both (possibly promoted) operands have the same signedness, then the operand with the smaller type is converted to the larger type.

When the signedness differs and the type of the unsigned operand is the same as or larger than that of the signed operand, the signed operand is converted to unsigned. For example, given an `unsigned int` and an `int`, the `int` is converted to `unsigned int`. It is worth noting that if the `int` has a negative value, the result will be converted as described in § 2.1.2 (p. 35), with the same results.

The remaining case is when the signed operand has a larger type than the unsigned operand. In this case, the result is machine dependent. If all values in the unsigned type fit in the larger type, then the unsigned operand is converted to the signed type. If the values don't fit, then the signed operand is converted to the unsigned type. For example, if the operands are `long` and `unsigned int`, and `int` and `long` have the same size, the `long` will be converted to `unsigned int`. If the `long` type has more bits, then the `unsigned int` will be converted to `long`.

Understanding the Arithmetic Conversions

One way to understand the arithmetic conversions is to study lots of examples:

```
bool      flag;        char           cval;
short     sval;        unsigned short usval;
int       ival;        unsigned int   uival;
long      lval;        unsigned long  ulval;
float     fval;        double         dval;

3.14159L + 'a'; // 'a' promoted to int, then that int converted to long double
dval + ival;    // ival converted to double
dval + fval;    // fval converted to double
ival = dval;    // dval converted (by truncation) to int
flag = dval;    // if dval is 0, then flag is false, otherwise true
cval + fval;    // cval promoted to int, then that int converted to float
sval + cval;    // sval and cval promoted to int
```

```
cval + lval;     // cval converted to long
ival + ulval;    // ival converted to unsigned long
usval + ival;    // promotion depends on the size of unsigned short and int
uival + lval;    // conversion depends on the size of unsigned int and long
```

In the first addition, the character constant lowercase `'a'` has type `char`, which is a numeric value (§ 2.1.1, p. 32). What that value is depends on the machine's character set. On our machine, `'a'` has the numeric value 97. When we add `'a'` to a `long double`, the `char` value is promoted to `int`, and then that `int` value is converted to a `long double`. The converted value is added to the literal. The other interesting cases are the last two expressions involving unsigned values. The type of the result in these expressions is machine dependent.

Exercises Section 4.11.1

Exercise 4.34: Given the variable definitions in this section, explain what conversions take place in the following expressions:

(a) `if (fval)` (b) `dval = fval + ival;` (c) `dval + ival * cval;`

Remember that you may need to consider the associativity of the operators.

Exercise 4.35: Given the following definitions,

```
char cval;      int ival;     unsigned int ui;
float fval;     double dval;
```

identify the implicit type conversions, if any, taking place:

```
(a) cval = 'a' + 3;       (b) fval = ui - ival * 1.0;
(c) dval = ui * fval;     (d) cval = ival + fval + dval;
```

4.11.2 Other Implicit Conversions

In addition to the arithmetic conversions, there are several additional kinds of implicit conversions. These include:

Array to Pointer Conversions: In most expressions, when we use an array, the array is automatically converted to a pointer to the first element in that array:

```
int ia[10];     // array of ten ints
int* ip = ia;   // convert ia to a pointer to the first element
```

This conversion is not performed when an array is used with `decltype` or as the operand of the address-of (`&`), `sizeof`, or `typeid` (which we'll cover in § 19.2.2 (p. 826)) operators. The conversion is also omitted when we initialize a reference to an array (§ 3.5.1, p. 114). As we'll see in § 6.7 (p. 247), a similar pointer conversion happens when we use a function type in an expression.

Pointer Conversions: There are several other pointer conversions: A constant integral value of 0 and the literal `nullptr` can be converted to any pointer type; a pointer to any nonconst type can be converted to `void*`, and a pointer to any

type can be converted to a `const void*`. We'll see in § 15.2.2 (p. 597) that there is an additional pointer conversion that applies to types related by inheritance.

Conversions to `bool`: There is an automatic conversion from arithmetic or pointer types to `bool`. If the pointer or arithmetic value is zero, the conversion yields `false`; any other value yields `true`:

```
char *cp = get_string();
if (cp) /* ... */      // true if the pointer cp is not zero
while (*cp) /* ... */ // true if *cp is not the null character
```

Conversion to `const`: We can convert a pointer to a nonconst type to a pointer to the corresponding `const` type, and similarly for references. That is, if `T` is a type, we can convert a pointer or a reference to `T` into a pointer or reference to `const T`, respectively (§ 2.4.1, p. 61, and § 2.4.2, p. 62):

```
int i;
const int &j = i;  // convert a nonconst to a reference to const int
const int *p = &i; // convert address of a nonconst to the address of a const
int &r = j, *q = p; // error: conversion from const to nonconst not allowed
```

The reverse conversion—removing a low-level `const`—does not exist.

Conversions Defined by Class Types: Class types can define conversions that the compiler will apply automatically. The compiler will apply only one class-type conversion at a time. In § 7.5.4 (p. 295) we'll see an example of when multiple conversions might be required, and will be rejected.

Our programs have already used class-type conversions: We use a class-type conversion when we use a C-style character string where a library `string` is expected (§ 3.5.5, p. 124) and when we read from an `istream` in a condition:

```
string s, t = "a value";  // character string literal converted to type string
while (cin >> s)          // while condition converts cin to bool
```

The condition (`cin >> s`) reads `cin` and yields `cin` as its result. Conditions expect a value of type `bool`, but this condition tests a value of type `istream`. The IO library defines a conversion from `istream` to `bool`. That conversion is used (automatically) to convert `cin` to `bool`. The resulting `bool` value depends on the state of the stream. If the last read succeeded, then the conversion yields `true`. If the last attempt failed, then the conversion to `bool` yields `false`.

4.11.3 Explicit Conversions

Sometimes we want to explicitly force an object to be converted to a different type. For example, we might want to use floating-point division in the following code:

```
int i, j;
double slope = i/j;
```

To do so, we'd need a way to explicitly convert `i` and/or `j` to `double`. We use a **cast** to request an explicit conversion.

Although necessary at times, casts are inherently dangerous constructs.

Named Casts

A named cast has the following form:

```
cast-name<type>(expression);
```

where *type* is the target type of the conversion, and *expression* is the value to be cast. If *type* is a reference, then the result is an lvalue. The *cast-name* may be one of **`static_cast`**, **`dynamic_cast`**, **`const_cast`**, and **`reinterpret_cast`**. We'll cover `dynamic_cast`, which supports the run-time type identification, in § 19.2 (p. 825). The *cast-name* determines what kind of conversion is performed.

`static_cast`

Any well-defined type conversion, other than those involving low-level `const`, can be requested using a `static_cast`. For example, we can force our expression to use floating-point division by casting one of the operands to `double`:

```
// cast used to force floating-point division
double slope = static_cast<double>(j) / i;
```

A `static_cast` is often useful when a larger arithmetic type is assigned to a smaller type. The cast informs both the reader of the program and the compiler that we are aware of and are not concerned about the potential loss of precision. Compilers often generate a warning for assignments of a larger arithmetic type to a smaller type. When we do an explicit cast, the warning message is turned off.

A `static_cast` is also useful to perform a conversion that the compiler will not generate automatically. For example, we can use a `static_cast` to retrieve a pointer value that was stored in a `void*` pointer (§ 2.3.2, p. 56):

```
void* p = &d;    // ok: address of any nonconst object can be stored in a void*
// ok: converts void* back to the original pointer type
double *dp = static_cast<double*>(p);
```

When we store a pointer in a `void*` and then use a `static_cast` to cast the pointer back to its original type, we are guaranteed that the pointer value is preserved. That is, the result of the cast will be equal to the original address value. However, we must be certain that the type to which we cast the pointer is the actual type of that pointer; if the types do not match, the result is undefined.

`const_cast`

A `const_cast` changes only a low-level (§ 2.4.3, p. 63) `const` in its operand:

```
const char *pc;
char *p = const_cast<char*>(pc); // ok: but writing through p is undefined
```

Conventionally we say that a cast that converts a `const` object to a nonconst type "casts away the `const`." Once we have cast away the `const` of an object, the compiler will no longer prevent us from writing to that object. If the object was originally not a `const`, using a cast to obtain write access is legal. However, using a `const_cast` in order to write to a `const` object is undefined.

Only a `const_cast` may be used to change the `const`ness of an expression. Trying to change whether an expression is `const` with any of the other forms of named cast is a compile-time error. Similarly, we cannot use a `const_cast` to change the type of an expression:

```
const char *cp;
//  error: static_cast can't cast away const
char *q = static_cast<char*>(cp);
static_cast<string>(cp); //  ok: converts string literal to string
const_cast<string>(cp);  //  error: const_cast only changes constness
```

A `const_cast` is most useful in the context of overloaded functions, which we'll describe in § 6.4 (p. 232).

`reinterpret_cast`

A `reinterpret_cast` generally performs a low-level reinterpretation of the bit pattern of its operands. As an example, given the following cast

```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
```

we must never forget that the actual object addressed by `pc` is an `int`, not a character. Any use of `pc` that assumes it's an ordinary character pointer is likely to fail at run time. For example:

```
string str(pc);
```

is likely to result in bizarre run-time behavior.

The use of `pc` to initialize `str` is a good example of why `reinterpret_cast` is dangerous. The problem is that types are changed, yet there are no warnings or errors from the compiler. When we initialized `pc` with the address of an `int`, there is no error or warning from the compiler because we explicitly said the conversion was okay. Any subsequent use of `pc` will assume that the value it holds is a `char*`. The compiler has no way of knowing that it actually holds a pointer to an `int`. Thus, the initialization of `str` with `pc` is absolutely correct—albeit in this case meaningless or worse! Tracking down the cause of this sort of problem can prove extremely difficult, especially if the cast of `ip` to `pc` occurs in a file separate from the one in which `pc` is used to initialize a `string`.

A `reinterpret_cast` is inherently machine dependent. Safely using `reinterpret_cast` requires completely understanding the types involved as well as the details of how the compiler implements the cast.

Old-Style Casts

In early versions of C++, an explicit cast took one of the following two forms:

```
type (expr); // function-style cast notation
(type) expr; // C-language-style cast notation
```

ADVICE: AVOID CASTS

Casts interfere with normal type checking (§ 2.2.2, p. 46). As a result, we strongly recommend that programmers avoid casts. This advice is particularly applicable to `reinterpret_cast`s. Such casts are always hazardous. A `const_cast` can be useful in the context of overloaded functions, which we'll cover in § 6.4 (p. 232). Other uses of `const_cast` often indicate a design flaw. The other casts, `static_cast` and `dynamic_cast`, should be needed infrequently. Every time you write a cast, you should think hard about whether you can achieve the same result in a different way. If the cast is unavoidable, errors can be mitigated by limiting the scope in which the cast value is used and by documenting all assumptions about the types involved.

Depending on the types involved, an old-style cast has the same behavior as a `const_cast`, a `static_cast`, or a `reinterpret_cast`. When we use an old-style cast where a `static_cast` or a `const_cast` would be legal, the old-style cast does the same conversion as the respective named cast. If neither cast is legal, then an old-style cast performs a `reinterpret_cast`. For example:

```
char *pc = (char*) ip;  // ip is a pointer to int
```

has the same effect as using a `reinterpret_cast`.

Old-style casts are less visible than are named casts. Because they are easily overlooked, it is more difficult to track down a rogue cast.

EXERCISES SECTION 4.11.3

Exercise 4.36: Assuming `i` is an `int` and `d` is a `double` write the expression `i *= d` so that it does integral, rather than floating-point, multiplication.

Exercise 4.37: Rewrite each of the following old-style casts to use a named cast:

```
int i;  double d;  const string *ps;  char *pc;  void *pv;
(a) pv = (void*)ps;    (b) i = int(*pc);
(c) pv = &d;           (d) pc = (char*) pv;
```

Exercise 4.38: Explain the following expression:

```
double slope = static_cast<double>(j/i);
```

4.12 Operator Precedence Table

Associativity and Operator		Function	Use	See Page
L	`::`	global scope	`::`name	286
L	`::`	class scope	class`::`name	88
L	`::`	namespace scope	namespace`::`name	82
L	`.`	member selectors	object`.`member	23
L	`->`	member selectors	pointer`->`member	110
L	`[]`	subscript	expr`[` expr `]`	116
L	`()`	function call	name`(`expr_list`)`	23
L	`()`	type construction	type`(`expr_list`)`	164
R	`++`	postfix increment	lvalue`++`	147
R	`--`	postfix decrement	lvalue`--`	147
R	`typeid`	type ID	`typeid(`type`)`	826
R	`typeid`	run-time type ID	`typeid(`expr`)`	826
R	explicit cast	type conversion	*cast_name*<type>(expr)	162
R	`++`	prefix increment	`++`lvalue	147
R	`--`	prefix decrement	`--`lvalue	147
R	`~`	bitwise NOT	`~`expr	152
R	`!`	logical NOT	`!`expr	141
R	`-`	unary minus	`-`expr	140
R	`+`	unary plus	`+`expr	140
R	`*`	dereference	`*`expr	53
R	`&`	address-of	`&`lvalue	52
R	`()`	type conversion	(type) expr	164
R	`sizeof`	size of object	`sizeof` expr	156
R	`sizeof`	size of type	`sizeof(` type `)`	156
R	`sizeof...`	size of parameter pack	`sizeof...(` name `)`	700
R	`new`	allocate object	`new` type	458
R	`new[]`	allocate array	`new` type[size]	458
R	`delete`	deallocate object	`delete` expr	460
R	`delete[]`	deallocate array	`delete[]` expr	460
R	`noexcept`	can expr throw	`noexcept (` expr `)`	780
L	`->*`	ptr to member select	ptr`->*`ptr_to_member	837
L	`.*`	ptr to member select	obj`.*`ptr_to_member	837
L	`*`	multiply	expr `*` expr	139
L	`/`	divide	expr `/` expr	139
L	`%`	modulo (remainder)	expr `%` expr	139
L	`+`	add	expr `+` expr	139
L	`-`	subtract	expr `-` expr	139
L	`<<`	bitwise shift left	expr `<<` expr	152
L	`>>`	bitwise shift right	expr `>>` expr	152
L	`<`	less than	expr `<` expr	141
L	`<=`	less than or equal	expr `<=` expr	141
L	`>`	greater than	expr `>` expr	141

Continued on next page

Table 4.4: Operator Precedence (continued)

Associativity and Operator		Function	Use	See Page
L	`>=`	greater than or equal	expr `>=` expr	141
L	`==`	equality	expr `==` expr	141
L	`!=`	inequality	expr `!=` expr	141
L	`&`	bitwise AND	expr `&` expr	152
L	`^`	bitwise XOR	expr `^` expr	152
L	`|`	bitwise OR	expr `|` expr	152
L	`&&`	logical AND	expr `&&` expr	141
L	`||`	logical OR	expr `||` expr	141
R	`?:`	conditional	expr `?` expr `:` expr	151
R	`=`	assignment	lvalue `=` expr	144
R	`*=, /=, %=,`	compound assign	lvalue `+=` expr, etc.	144
R	`+=, -=,`			144
R	`<<=, >>=,`			144
R	`&=, |=, ^=`			144
R	`throw`	throw exception	throw expr	193
L	`,`	comma	expr `,` expr	157

Chapter Summary

C++ provides a rich set of operators and defines their meaning when applied to values of the built-in types. Additionally, the language supports operator overloading, which allows us to define the meaning of the operators for class types. We'll see in Chapter 14 how to define operators for our own types.

To understand expressions involving more than one operator it is necessary to understand precedence, associativity, and order of operand evaluation. Each operator has a precedence level and associativity. Precedence determines how operators are grouped in a compound expression. Associativity determines how operators at the same precedence level are grouped.

Most operators do not specify the order in which operands are evaluated: The compiler is free to evaluate either the left- or right-hand operand first. Often, the order of operand evaluation has no impact on the result of the expression. However, if both operands refer to the same object and one of the operands *changes* that object, then the program has a serious bug—and a bug that may be hard to find.

Finally, operands are often converted automatically from their initial type to another related type. For example, small integral types are promoted to a larger integral type in every expression. Conversions exist for both built-in and class types. Conversions can also be done explicitly through a cast.

Defined Terms

arithmetic conversion A conversion from one arithmetic type to another. In the context of the binary arithmetic operators, arithmetic conversions usually attempt to preserve precision by converting a smaller type to a larger type (e.g., integral types are converted to floating point).

associativity Determines how operators with the same precedence are grouped. Operators can be either right associative (operators are grouped from right to left) or left associative (operators are grouped from left to right).

binary operators Operators that take two operands.

cast An explicit conversion.

compound expression An expression involving more than one operator.

const_cast A cast that converts a low-level `const` object to the corresponding non`const` type or vice versa.

conversion Process whereby a value of one type is transformed into a value of another type. The language defines conversions among the built-in types. Conversions to and from class types are also possible.

dynamic_cast Used in combination with inheritance and run-time type identification. See § 19.2 (p. 825).

expression The lowest level of computation in a C++ program. Expressions generally apply an operator to one or more operands. Each expression yields a result. Expressions can be used as operands, so we can write compound expressions requiring the evaluation of multiple operators.

implicit conversion A conversion that is automatically generated by the compiler. Given an expression that needs a particular type but has an operand of a differing type, the compiler will automatically convert the operand to the desired type if an appropriate conversion exists.

integral promotions conversions that take a smaller integral type to its most closely related larger integral type. Operands of small integral types (e.g., `short`, `char`, etc.) are always promoted, even in contexts where such conversions might not seem to be required.

lvalue An expression that yields an object or function. A nonconst lvalue that denotes an object may be the left-hand operand of assignment.

operands Values on which an expression operates. Each operator has one or more operands associated with it.

operator Symbol that determines what action an expression performs. The language defines a set of operators and what those operators mean when applied to values of built-in type. The language also defines the precedence and associativity of each operator and specifies how many operands each operator takes. Operators may be overloaded and applied to values of class type.

order of evaluation Order, if any, in which the operands to an operator are evaluated. In most cases, the compiler is free to evaluate operands in any order. However, the operands are always evaluated before the operator itself is evaluated. Only the `&&`, `||`, `?:`, and comma operators specify the order in which their operands are evaluated.

overloaded operator Version of an operator that is defined for use with a class type. We'll see in Chapter 14 how to define overloaded versions of operators.

precedence Defines the order in which different operators in a compound expression are grouped. Operators with higher precedence are grouped more tightly than operators with lower precedence.

promoted See integral promotions.

reinterpret_cast Interprets the contents of the operand as a different type. Inherently machine dependent and dangerous.

result Value or object obtained by evaluating an expression.

rvalue Expression that yields a value but not the associated location, if any, of that value.

short-circuit evaluation Term used to describe how the logical AND and logical OR operators execute. If the first operand to these operators is sufficient to determine the overall result, evaluation stops. We are guaranteed that the second operand is not evaluated.

sizeof Operator that returns the size, in bytes, to store an object of a given type name or of the type of a given expression.

static_cast An explicit request for a well-defined type conversion. Often used to override an implicit conversion that the compiler would otherwise perform.

unary operators Operators that take a single operand.

`,` operator Comma operator. Binary operator that is evaluated left to right. The result of a comma expression is the value of the right-hand operand. The result is an lvalue if and only if that operand is an lvalue.

`?:` operator Conditional operator. Provides an if-then-else expression of the form

```
cond ? expr1 : expr2;
```

If the condition *cond* is true, then *expr1* is evaluated. Otherwise, *expr2* is evaluated. The type *expr1* and *expr2* must be the same type or be convertible to a common type. Only one of *expr1* or *expr2* is evaluated.

`&&` operator Logical AND operator. Result is `true` if both operands are `true`. The right-hand operand is evaluated *only* if the left-hand operand is `true`.

`&` operator Bitwise AND operator. Generates a new integral value in which each bit position is 1 if both operands have a 1 in that position; otherwise the bit is 0.

^ operator Bitwise exclusive or operator. Generates a new integral value in which each bit position is 1 if either but not both operands contain a 1 in that bit position; otherwise, the bit is 0.

|| operator Logical OR operator. Yields `true` if either operand is `true`. The right-hand operand is evaluated *only* if the left-hand operand is `false`.

| operator Bitwise OR operator. Generates a new integral value in which each bit position is 1 if either operand has a 1 in that position; otherwise the bit is 0.

++ operator The increment operator. The increment operator has two forms, prefix and postfix. Prefix increment yields an lvalue. It adds 1 to the operand and returns the changed value of the operand. Postfix increment yields an rvalue. It adds 1 to the operand and returns a copy of the original, unchanged value of the operand. Note: Iterators have ++ even if they do not have the + operator.

-- operator The decrement operator has two forms, prefix and postfix. Prefix decrement yields an lvalue. It subtracts 1 from the operand and returns the changed value of the operand. Postfix decrement yields an rvalue. It subtracts 1 from the operand and returns a copy of the original, unchanged value of the operand. Note: Iterators have `--` even if they do not have the `-`.

<< operator The left-shift operator. Shifts bits in a (possibly promoted) copy of the value of the left-hand operand to the left. Shifts as many bits as indicated by the right-hand operand. The right-hand operand must be zero or positive and strictly less than the number of bits in the result. Left-hand operand should be `unsigned`; if the left-hand operand is `signed`, it is undefined if a shift causes a different bit to shift into the sign bit.

>> operator The right-shift operator. Like the left-shift operator except that bits are shifted to the right. If the left-hand operand is `signed`, it is implementation defined whether bits shifted into the result are 0 or a copy of the sign bit.

~ operator Bitwise NOT operator. Generates a new integral value in which each bit is an inverted copy of the corresponding bit in the (possibly promoted) operand.

! operator Logical NOT operator. Returns the inverse of the `bool` value of its operand. Result is `true` if operand is `false` and vice versa.