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.