OPERATORS IN CPP

Operators

C/C++ is rich in built-in operators. In fact, it places more significance on operators than do most other computer languages. There are four main classes of operators: arithmetic, relational, logical, and bitwise. In addition, there are some special operators for particular tasks.

The Assignment Operator

You can use the assignment operator within any valid expression. This is not the case with many computer languages (including Pascal, BASIC, and FORTRAN), which treat the assignment operator as a special case statement. The general form of the assignment operator is variable_name = expression;

where an expression may be as simple as a single constant or as complex as you require. C/C++ uses a single equal sign to indicate assignment (unlike Pascal or Modula-2, which use the := construct). The target, or left part, of the assignment must be a variable or a pointer, not a function or a constant. Frequently in literature on C/C++ and in compiler error messages you will see these two terms: lvalue and rvalue. Simply put, an lvalue is any object that can occur on the left side of an assignment statement. For all practical purposes, "lvalue" means "variable." The term rvalue refers to expressions on the right side of an assignment and simply means the value of an expression.

Type Conversion in Assignments

When variables of one type are mixed with variables of another type, a type conversion will occur. In an assignment statement, the type conversion rule is easy: The value of the right side
(expression side) of the assignment is converted to the type of the left side (target variable), as illustrated here:

```c
int x;
char ch;
float f;
void func(void)
{
  ch = x; /* line 1 */
x = f; /* line 2 */
f = ch; /* line 3 */
f = x; /* line 4 */
}
```

**Multiple Assignments**

C/C++ allows you to assign many variables the same value by using multiple assignments in a single statement. For example, this program fragment assigns `x`, `y`, and `z` the value 0: `x = y = z = 0`; In professional programs, variables are frequently assigned common values using this method.

**Arithmetic Operators**

Table 2-4 lists C/C++'s arithmetic operators. The operators `+`, `−`, `∗`, and `/` work as they do in most other computer languages. You can apply them to almost any built-in data type. When you apply `/` to an integer or character, any remainder will be truncated. For example, 5/2 will equal 2 in integer division. The modulus operator `%` also works in C/C++ as it does in other languages, yielding the remainder of an integer division. However, you cannot use it on floating-point types.

The following code fragment illustrates `%:

```c
int x, y;
x = 5;
y = 2;
printf("%d", x/y); /* will display 2 */
printf("%d", x%y); /* will display 1, the remainder of the integer division */
```
x = 1;
y = 2;
printf("%d %d", x/y, x%y); /* will display 0 1 */
The last line prints a 0 and a 1 because 1/2 in integer division is 0 with a remainder of 1.
The unary minus multiplies its operand by -1. That is, any number preceded by a minus sign
switches its sign.

**Increment and Decrement**

C/C++ includes two useful operators not found in some other computer languages. These are
the increment and decrement operators, ++ and --. The operator ++ adds 1 to its operand,
and -- subtracts 1.

In other words:

x = x+1;
is the same as

++x;
and

x = x-1;
is the same as

x--;

Both the increment and decrement operators may either precede (prefix) or follow (postfix) the operand.

For example, x = x+1; can be written

++x;
or

x++;

There is, however, a difference between the prefix and postfix forms when you use these operators in an expression. When an increment or decrement operator precedes its operand, the increment or decrement operation is performed before obtaining the value of the operand for use in the expression. If the operator follows its operand, the value of the operand is obtained before incrementing or decrementing it.

For instance,

x = 10;
y = ++x;
sets y to 11. However, if you write the code as
x = 10;
y = x++;
y is set to 10. Either way, x is set to 11; the difference is in when it happens. Most C/C++
compilers produce very fast, efficient object code for increment and decrement operations—
code that is better than that generated by using the equivalent assignment statement. For this
reason, you should use the increment and decrement operators when you can.
Here is the precedence of the arithmetic operators:

**highest** ++ --
-- (unary minus)
*/%*

**lowest** + –

Operators on the same level of precedence are evaluated by the compiler from left to right. Of
course, you can use parentheses to alter the order of evaluation. C/C++ treats parentheses in the
same way as virtually all other computer languages. Parentheses force an operation, or set of
operations, to have a higher level of precedence.

**Relational and Logical Operators**

In the term *relational operator*, relational refers to the relationships that values can have with
one another. In the term *logical operator*, logical refers to the ways these relationships can be
connected. Because the relational and logical operators often work together, they are discussed
together here. The idea of true and false underlies the concepts of relational and logical
operators.

In C, true is any value other than zero. False is zero. Expressions that use relational or logical
operators return 0 for false and 1 for true. C++ fully supports the zero/non-zero concept of true
and false. However, it also defines the **bool** data type and the Boolean constants **true** and **false**.
In C++, a 0 value is automatically converted into **false**, and a non-zero value is automatically
converted into **true**. The reverse also applies: **true** converts to 1 and **false** converts to 0. In
C++, the outcome of a relational or logical operation is **true** or **false**. But since this
automatically converts into 1 or 0, the distinction between C and C++ on this issue is mostly
academic.
The following program contains the function `xor()`, which returns the outcome of an exclusive OR operation performed on its two arguments:

```c
#include <stdio.h>
int xor(int a, int b);
int main(void)
{
    printf("%d", xor(1, 0));
    printf("%d", xor(1, 1));
    printf("%d", xor(0, 1));
    printf("%d", xor(0, 0));
    return 0;
}
```

**Bitwise Operators**

Unlike many other languages, C/C++ supports a full complement of bitwise operators. Since C was designed to take the place of assembly language for most programming including operations on bits. *Bitwise operation* refers to testing, setting, or shifting the actual bits in a byte or word, which correspond to the `char` and `int` data types and variants. You cannot use bitwise operations on `float`, `double`, `long double`, `void`, `bool`, or other, more complex types. These operations are applied to the individual bits of the operands. The bitwise AND, OR, and NOT (one's complement) are governed by the same truth table as their logical equivalents, except that they work bit by bit.

**The ? Operator**

C/C++ contains a very powerful and convenient operator that replaces certain statements of the if-then-else form. The ternary operator `?` takes the general form `Exp1 ? Exp2 : Exp3`; where `Exp1`, `Exp2`, and `Exp3` are expressions. Notice the use and placement of the colon. The `?` operator works like this: `Exp1` is evaluated. If it is true, `Exp2` is evaluated and becomes the value of the expression. If `Exp1` is false, `Exp3` is evaluated and its value becomes the value of the expression.

For example, in
x = 10;
y = x＞9 ? 100 : 200;
y is assigned the value 100. If x had been less than 9, y would have received the value 200. The
same code written using the if-else statement is  x = 10;
if(x＞9) y = 100;
else y = 200;
The ? operator will be discussed more fully in Chapter 3 in relationship to the other conditional
statements.

The & and * Pointer Operators
A pointer is the memory address of some object. A pointer variable is a variable that is
specifically declared to hold a pointer to an object of its specified type. Knowing a variable's
address can be of great help in certain types of routines. However, pointers have three main
functions in C/C++. They can provide a fast means of referencing array elements. They allow
functions to modify their calling parameters. Lastly, they support linked lists and other
dynamic data structures. Chapter 5 is devoted exclusively to pointers. However, this chapter
briefly covers the two operators that are used to manipulate pointers. The first pointer operator
is &, a unary operator that returns the memory address of its operand. (Remember, a unary
operator only requires one operand.)
For example, m = &count;
places into m the memory address of the variable count. This address is the computer's internal
location of the variable. It has nothing to do with the value of count. You can think of & as
meaning "the address of." Therefore, the preceding assignment statement means "m receives
the address of count."

The Compile-Time Operator sizeof
sizeof is a unary compile-time operator that returns the length, in bytes, of the variable or
parenthesized type-specifier that it precedes. For example, assuming that integers are 4 bytes
and doubles are 8 bytes, double f;
printf("%d", sizeof f);
printf("%d", sizeof(int));
will display 8 4.
Remember, to compute the size of a type, you must enclose the typename in parentheses.
This is not necessary for variable names, although there is no harm done if you do so. C/C++
defines (using **typedef**) a special type called **size_t**, which corresponds loosely to an unsigned
integer. Technically, the value returned by **sizeof** is of type **size_t**. For all practical purposes,
however, you can think of it (and use it) as if it were an unsigned integer value.

**The Comma Operator**
The comma operator strings together several expressions. The left side of the comma operator
is always evaluated as **void**. This means that the expression on the right side becomes the value
of the total comma-separated expression. For example, \( x = (y=3, y+1) \); first assigns \( y \) the value
3 and then assigns \( x \) the value 4. The parentheses are necessary because the comma operator
has a lower precedence than the assignment operator. Essentially, the comma causes a
sequence of operations. When you use it on the right side of an assignment statement, the value
assigned is the value of the last expression of the comma-separated list. The comma operator
has somewhat the same meaning as the word "and" in normal English as used in the phrase "do
this and this and this."

**The Dot (.) and Arrow (>) Operators**
In C, the . (dot) and the >(arrow) operators access individual elements of structures and unions.
**Structures** and **unions** are compound (also called **aggregate**) data types that may be referenced
under a single name (see Chapter 7). In C++, the dot and arrow operators are also used to
access the members of a class. The dot operator is used when working with a structure or union
directly. The arrow operator is used when a pointer to a structure or union is used.
For example, given the fragment

```
struct employee
{
    char name[80];
    int age;
    float wage;
} emp;
```
struct employee *p = &emp; /* address of emp into p */
you would write the following code to assign the value 123.23 to the wage member of
structure variable emp:
emp.wage = 123.23;
However, the same assignment using a pointer to emp would be p->wage = 123.23.

The [ ] and () Operators
Parentheses are operators that increase the precedence of the operations inside them. Square
brackets perform array indexing (arrays are discussed fully in Chapter 4). Given an array, the
expression within square brackets provides an index into that array.
For example,
#include <stdio.h>
char s[80];
int main(void)
{
    s[3] = 'X';
    printf("%c", s[3]);
    return 0;
}
first assigns the value 'X' to the fourth element (remember, all arrays begin at 0) of array s, and
then prints that element.