Chapter 3

Structure of a C++ Program

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

OBJECTIVES

After studying this chapter you will be able to:

- Understand the concept of and use expressions.
- Identify the seven types of C++ expressions.
- Use basic expressions in a program.
- Assign expression values to a variable.
- Evaluate expressions using precedence and associativity.
- Understand and use expression side effects.
- Understand and use compound statements.
- Understand that good functions are simple and short (KISS).
- Use parentheses to clarify code.
- Communicate clearly with the user through well written prompts.

©Brooks/Cole, 2004

Expressions

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

the value of a modulus

in the following expressions:

```
3 / 5 evaluates to  0
3 % 5 evaluates to  3
```

The multiplicative expressions are summarized in Table 3-2.

*	Result is algebraic multiplication of two operands.
/	Result is algebraic division of first operand by second operand: • Integer quotient if both operands are integer. • Floating-point quotient if either operand is a floating-point number.
%	Result is integer remainder after first operand is divided by second operand; operands must be integer types.

Table 3-2 Multiplicative binary operators

Table 3-3 contains several examples of the multiplicative binary expressions

	Integer	Float
Multiplication:	3 * 5	Multipli
Value:	15	

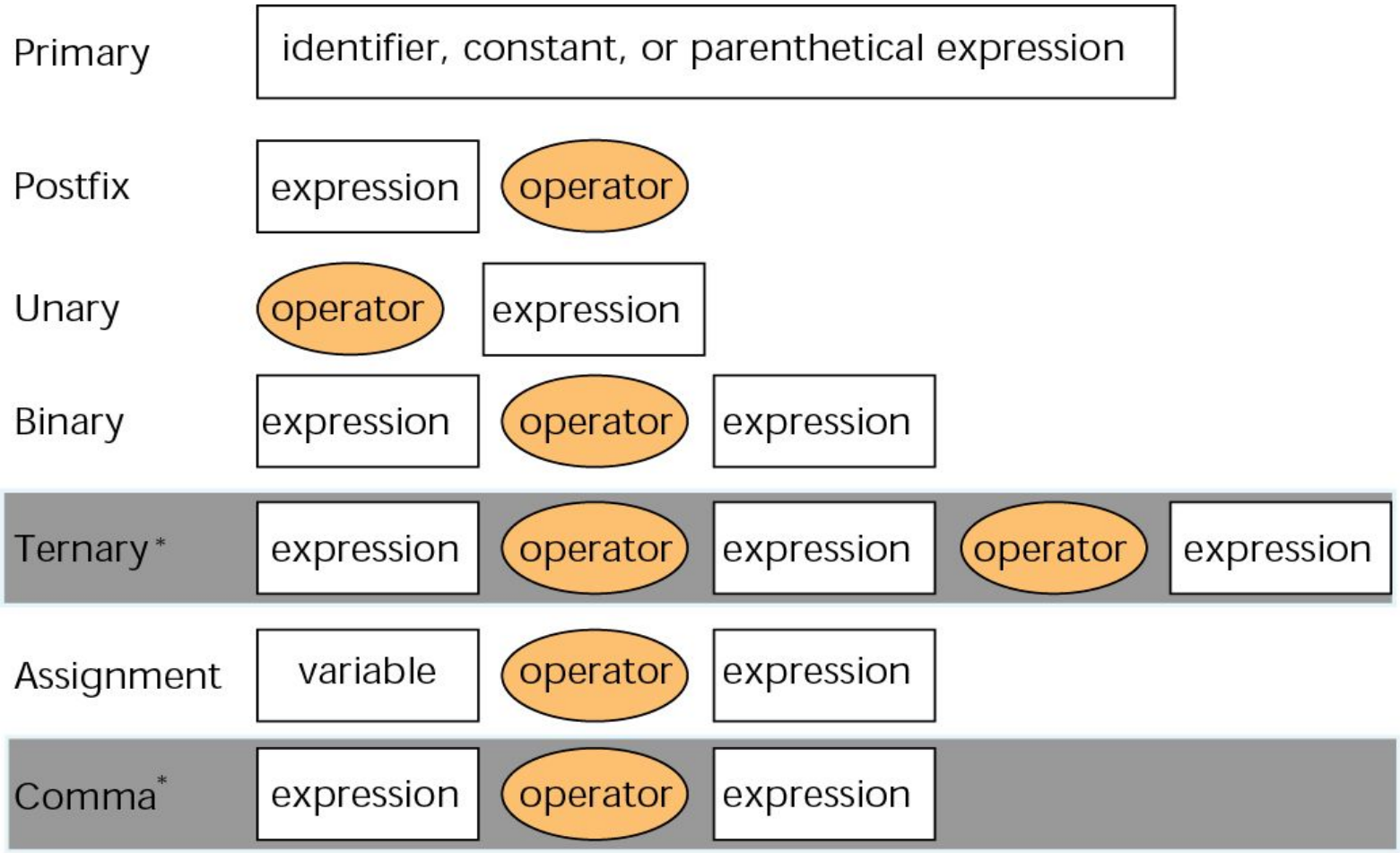
Note:

Expressions always reduce to a single value.

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Figure 3-1 C++ expression format

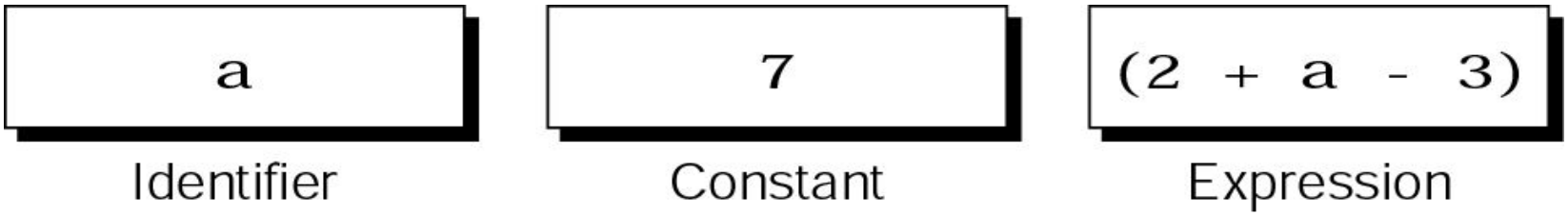


*These expression types are unique to C and C++

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

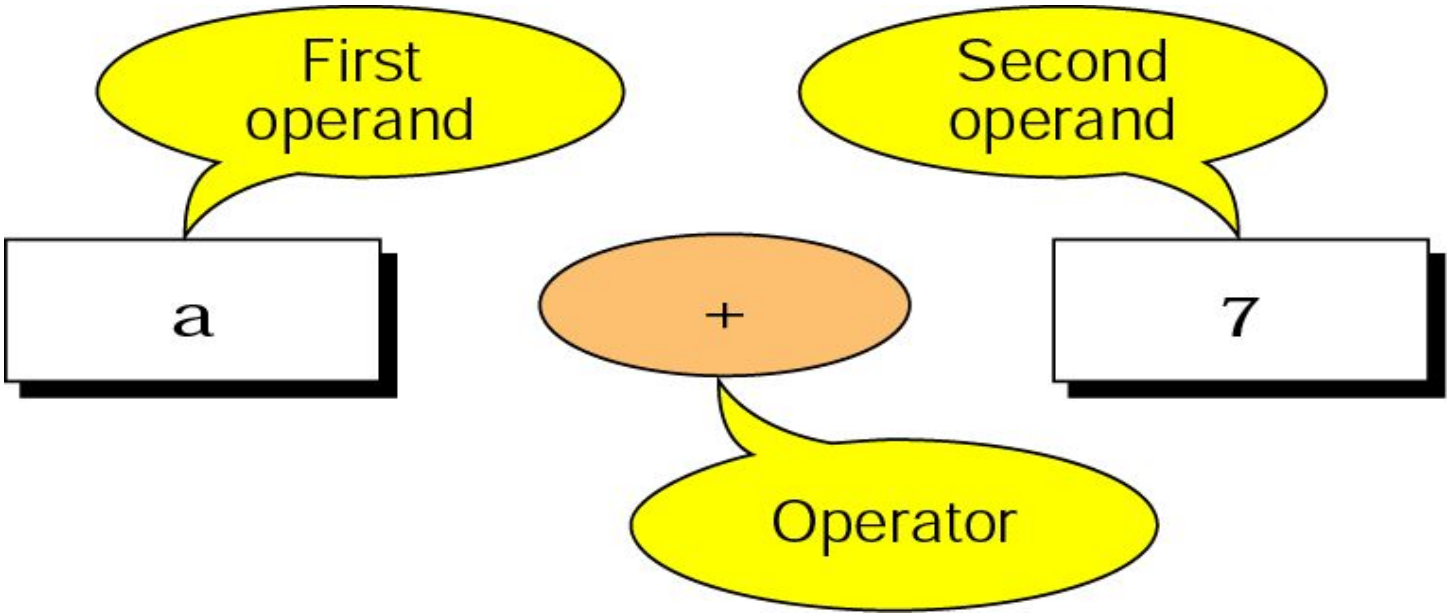
Figure 3-2 Primary expressions



Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

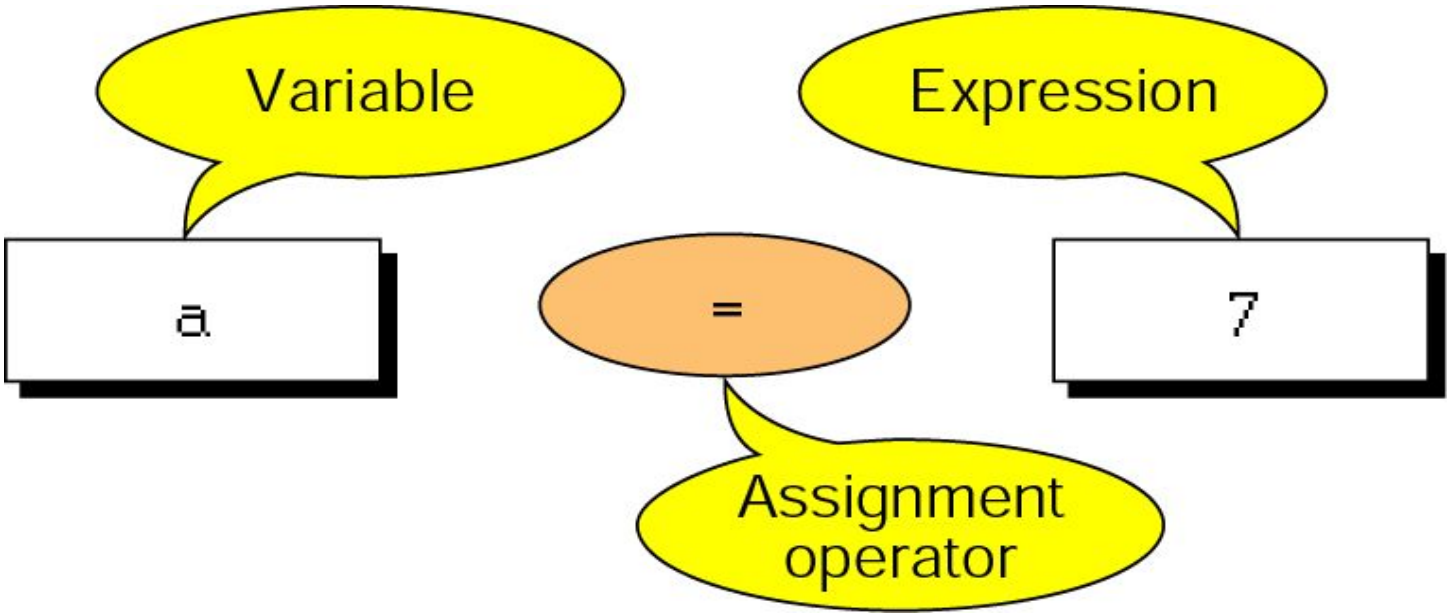
Figure 3-3 Binary expressions



Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Figure 3-4 Assignment expression



©Brooks/Cole, 2004

To evaluate a compound assignment expression, first change it to a simple assigr
as shown in Table 3-5. Then perform the operation to determine the value of the expres

Compound expression	Equivalent simple expression
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x += y	x = x + y
x -= y	x = x - y

able 3-5 Expansion of compound expressions

Examples of the five basic compound assignment expressions are seen in Table
u may find it helpful to convert each expression to its simple form to see hov
ult is determined.

Note:

The left operand in an assignment expression must be a single variable.

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Note:

Assignment Expression

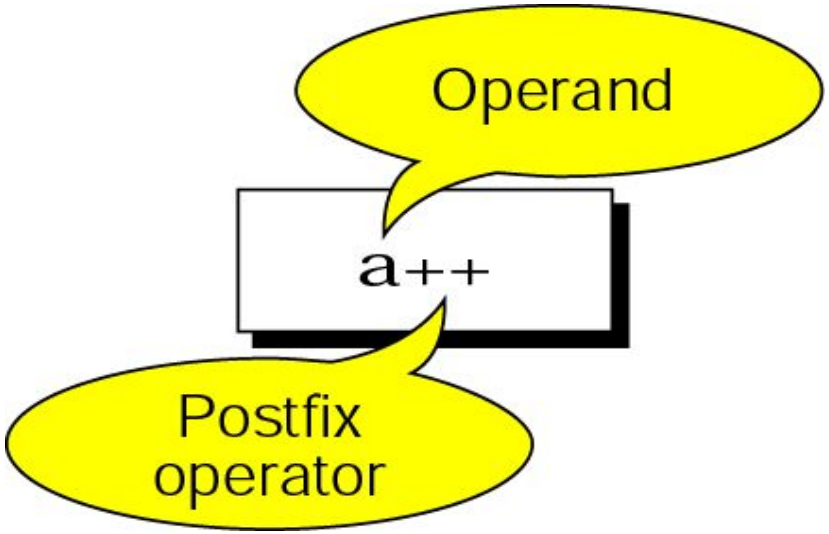
The assignment expression has a value and a result.

- ***The value of the total expression is the value of the expression on the right of the assignment operator (=).***
- ***The result places the expression value in the operator on the left of the assignment operator.***

Brooks/Cole
Thomson Learning™

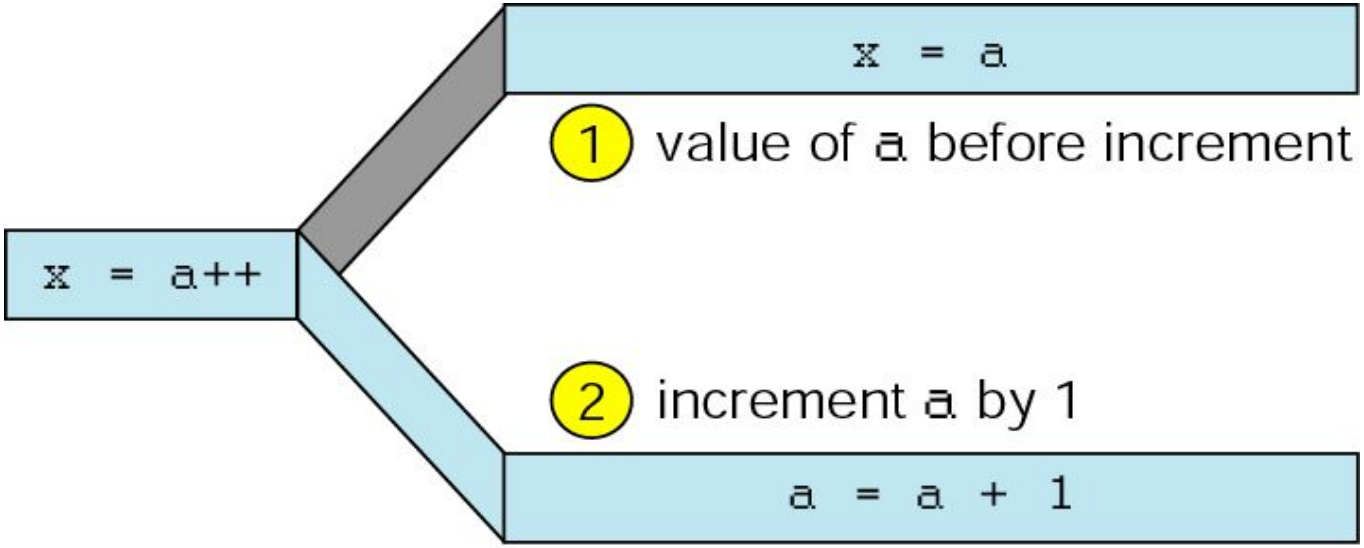
©Brooks/Cole, 2004

Figure 3-5 Postfix expressions



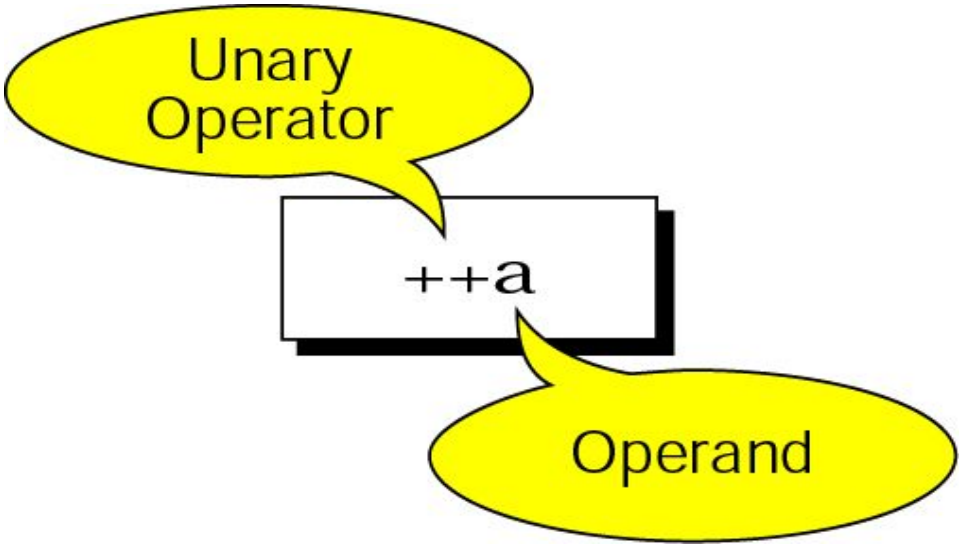
©Brooks/Cole, 2004

Figure 3-6 Result of postfix a++



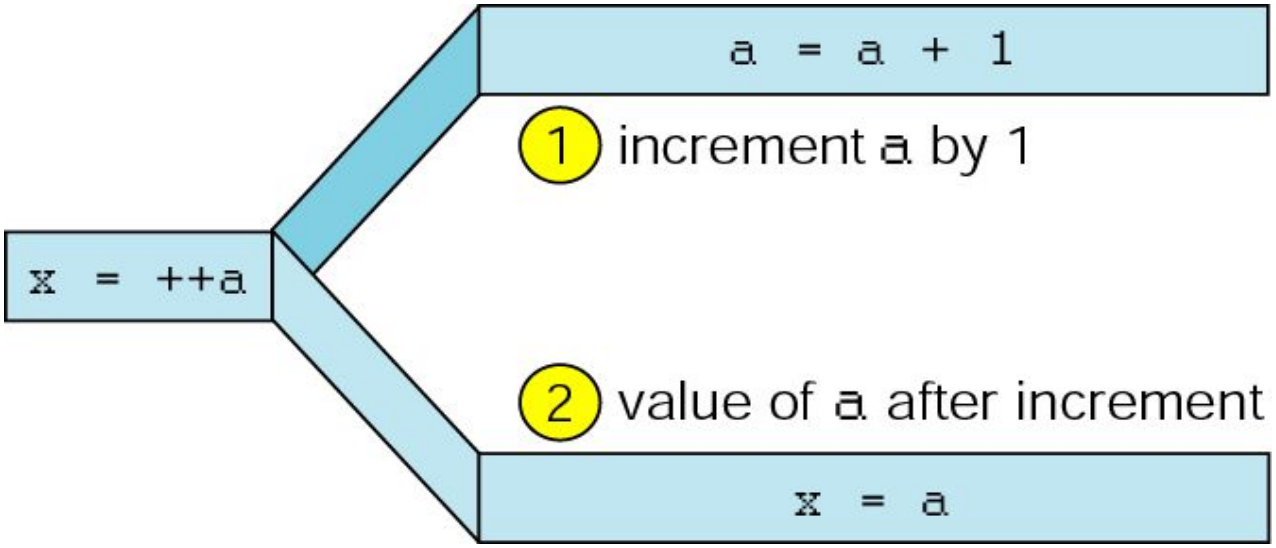
©Brooks/Cole, 2004

Figure 3-7 Unary expressions



©Brooks/Cole, 2004

Figure 3-8 Result of prefix ++a



Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Example 1	Example 2

x = 3;
y = ++x;
// x contains 4, y contains 4
	x = 3;
	y = x++;
	// x contains 4, y contains 3

©Brooks/Cole, 2004

operator	description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Note:

(++a) has the same effect as

(a = a + 1)

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Boolean conditions

..are built using

- Comparison operators

  == equal

  != not equal

  < less than

  > greater than

  <= less than or equal

  >= greater than or equal

- Boolean operators

  && and

  || or

  ! not

Examples

Assume we declared the following variables:

```
int a = 2, b=5, c=10;
```

Here are some examples of boolean conditions we can use:

- `if (a == b)`
- `if (a != b)`
- `if (a <= b+c)`
- `if(a <= b) && (b <= c)`
- `if !((a < b) && (b<c))`

Precedence and Associativity

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

1.22 Arithmetic

- Arithmetic operators:

C++ operation	Arithmetic operator	Algebraic expression	C++ expression
Addition	+	*f + 7*	`f + 7`
Subtraction	-	*p – c*	`p - c`
Multiplication	*	*bm*	`b * m`
Division	/	*x / y*	`x / y`
Modulus	%	*r mod s*	`r % s`

- Rules of operator precedence:

Operator(s)	Operation(s)	Order of evaluation (precedence)
`()`	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
`*`, `/`, or `%`	Multiplication Division Modulus	Evaluated second. If there are several, they re evaluated left to right.
`+` or `-`	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

© 2000 Prentice Hall, Inc. All rights reserved.

is found inside the cover. The table groups expressions by the seven formats shown in Table 3-1. By the time you finish this book, you will be quite familiar with all the different expressions, but it will take the whole book to get to that point.

Type	Description		‡	Pr	Assoc
Primary	Identifier Constant Parenthetical Expression		N	18	
Postfix	Function Postfix Increment • Decrement	(...) ++ – –	Y Y	17 16	Left
Unary	Prefix Increment • Decrement Size in Bytes Plus • Minus	++ – – sizeof + –	Y N N	15	Right
Binary	Multiply • Divide • Modulus Addition • Subtraction	* / % + –	N N	13 12	Left
Assignment	Assignment	= += –= *= /= %=	Y	2	Right

‡ Side Effects (Yes/No)

© 2000 Prentice Hall, Inc. All rights reserved.

Table 3-1 Partial precedence table for C++ expressions

Although the table is easy to use, a brief explanation of its contents is in order. The first column indicates the **expression type**. The expressions are listed in priority order (Pr, column 4) from the highest (18) to the lowest (2). The numbers are important because they indicate which expressions are evaluated first; that is, they indicate **precedence**. The higher the number, the earlier the evaluation. In examining the priority column, you will see that many expressions are missing (the priorities run consecutively from 18 to 1). We will fill in the missing expressions as we progress through the text.

Associativity (column 5) determines how operators with the same precedence are grouped together to form an expression. In other words, associativity determines how C++ would use parentheses to group the different expressions at the same level. "Left" indicates that the expression is evaluated from the left; "right" indicates that it is evaluated from the right.

© 2000 Prentice Hall, Inc. All rights reserved.

1.22 Arithmetic

- Arithmetic calculations
  - Use `*` for multiplication and `/` for division
  - Integer division truncates remainder
    - `7 / 5` evaluates to 1
  - Modulus operator returns the remainder
    - `7 % 5` evaluates to 2
- Operator precedence
  - Some arithmetic operators act before others (i.e., multiplication before addition)
    - Be sure to use parenthesis when needed
  - Example: Find the average of three variables a, b and c
    - Do not use: `a + b + c / 3`
    - Use: `(a + b + c ) / 3`

© 2000 Prentice Hall, Inc. All rights reserved.

- Let b=5
- K=(-(b++))

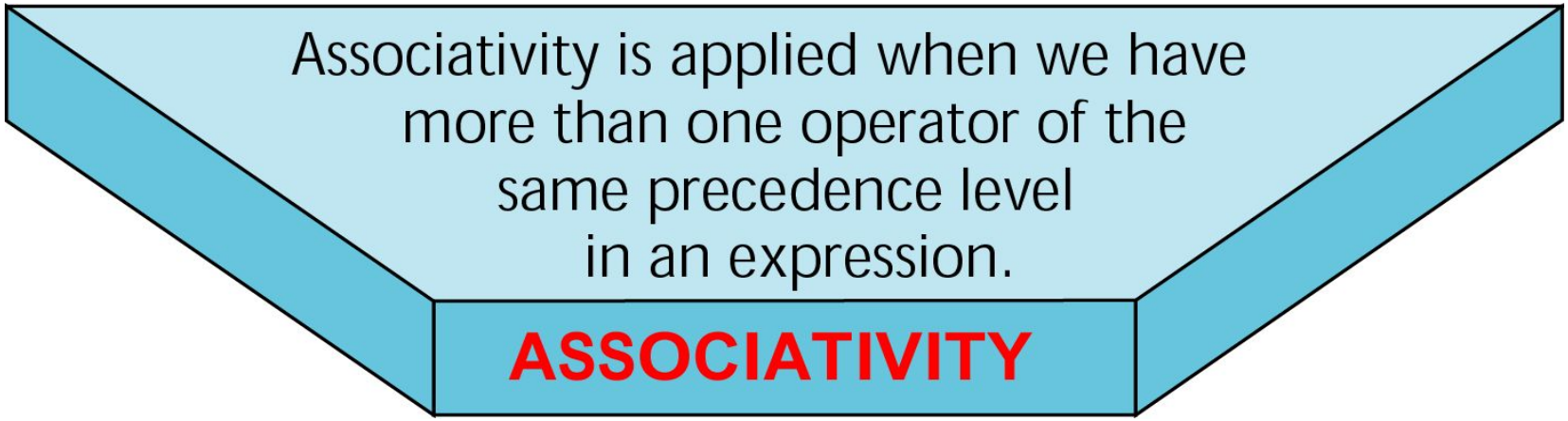
K =-5
b will be 6 as a side effect

- K=(-b++)

- K=-b++

©Brooks/Cole, 2004

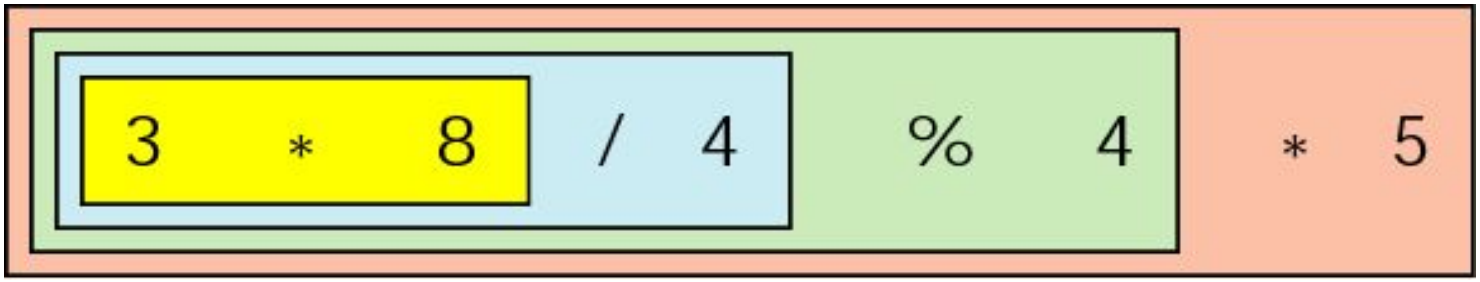
Figure 3-9 Associativity



Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Figure 3-10 Left associativity



Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

	Postfix Increment • Decrement	++ – –	Y	16	Left
Unary	Prefix Increment • Decrement Size in Bytes Plus • Minus	++ – – sizeof + –	Y N N	15	Right
Binary	Multiply • Divide • Modulus Addition • Subtraction	* / % + –	N N	13 12	Left
Assignment	Assignment = += –= *= /= %=		Y	2	Right

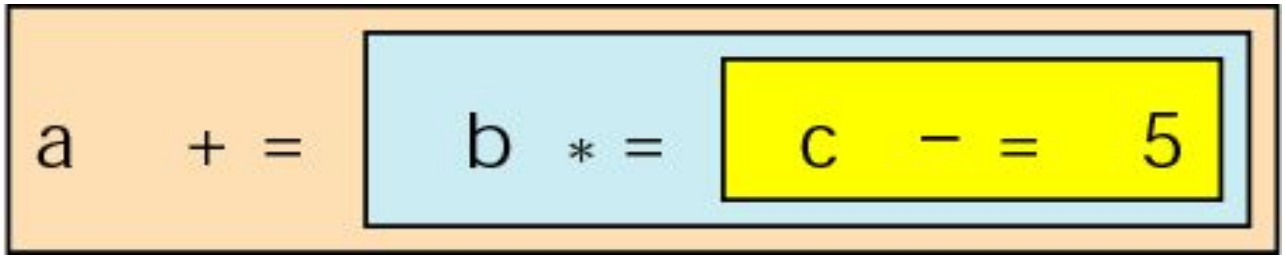
‡ Side Effects (Yes/No)

Right associativity

- A+=b*=c-=5

©Brooks/Cole, 2004

Figure 3-11 Right associativity



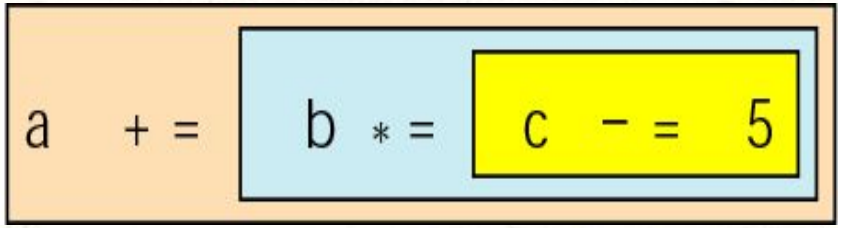
Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

There are only three types of expressions that associate from the right: the unary expressions, the conditional ternary expression, and the assignment expressions.

When there is more than one assignment operator in an assignment expression, the assignment operators must be interpreted from right to left. This means that the rightmost expression will be evaluated first; then its value will be assigned to the operand on the left of the assignment operator, and the next expression will be evaluated. Under these rules, the expression

```
a += b *= c -= 5
```



is evaluated as

```
(a += (b *= (c -= 5)))
```

which is expanded to

```
(a = a + (b = b * (c = c - 5)))
```

If a has an initial value of 3, b has an initial value of 5, and c has an initial value of 8, these expressions become

```
(a = 3 + (b = (5 * (c = 8 - 5))))
```

which results in c being assigned a value of 3, b being assigned a value of 15, and a being assigned a value of 18. The value of the complete expression is also 18. A diagram of this expression is seen in Figure 3-11.

Side Effects

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Now let's look at the rules for an expression that ... parenthetical expressions. For this example, consider the expression

```
--a * (3 + b) / 2 - c++ * b
```

Assume that the variables have the values used above, a = 3, b = 4, c = 5. To evaluate this expression, use the following steps:

1. Calculate the value of the parenthetical expression (3 + b) first. The expression now reads

```
--a * 7 / 2 - c++ * b
```

2. Evaluate the c++ *postfix* expression next. Remember that as a *postfix* expression, the value of c++ is the same as the value of c; the increment takes place after the evaluation. The expression now reads

```
--a * 7 / 2 - 5 * b
```

3. Evaluate the --a prefix expression next. Remember that as a *prefix* expression, the value of --a is the value after the side effect, which means that we first decrement a and then use its decremented value. The expression now reads

```
2 * 7 / 2 - 5 * b
```

4. The multiplication and division are now evaluated using the associativity rules, left to right, as shown below.

```
14 / 2 - 5 * b ⇒ 7 - 5 * 4 ⇒ 7
```

- (b++ - b++)


```
((4++) - (b++))
( 4    - (5++))
       (-1)
b is 6
```

Two other interpretations are

```
((b++) - (4++))
((5++) -   4  )
    (5 - 4)
      (+1)
b is 6
```

```
((4++) - (4++))
  ((4) - (4))
    (4 - 4)
      (0)
b is 6
```

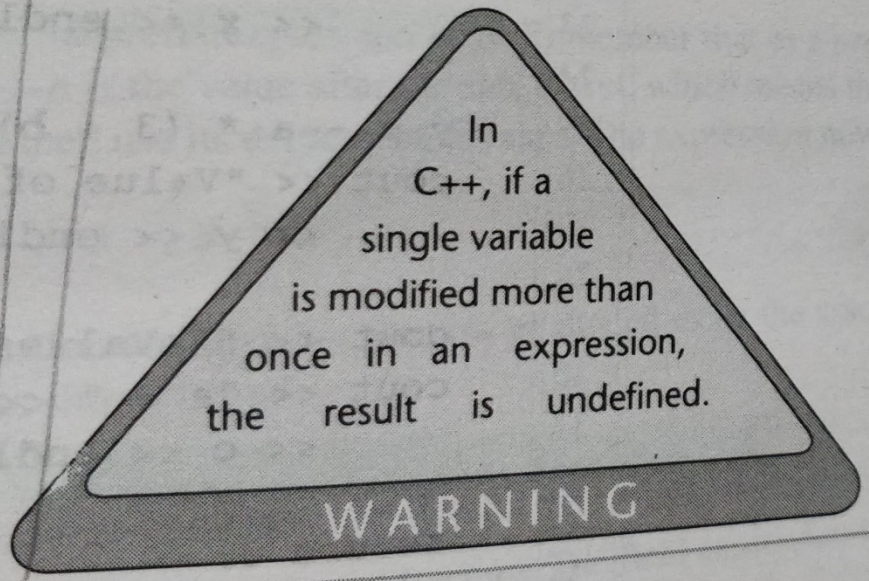
Although the side effect is the same in all cases—b is 6—the value of the expression

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

```
b is 6                                  (0)
                                  b is 6
```

Although the side effect is the same in all cases—b is 6—the value of the expression differs. In the first case, the value is –1; in the second case, the value is +1; and, in the last case, the value is 0. *Never use a variable affected by a side effect more than once in an expression.*



Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

3.4

Evaluating Expressions

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Mixed Type Expressions

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

The value and type of a mixed expression is determined by the following rules.

1. If the two values are both floating-point types—*float*, *double*, or *long double*—but of different precision, such as *float* and *double*, the type of the value with the lower precision is converted to the type of the value with the higher precision. For example, in an expression containing a *float* plus a *double*, the *float* is implicitly converted to a *double*. Note, however, that only the value is converted; the value and type of the memory variable are unchanged.

float_+double

2. If one of the values is a floating-point type and the other is an integral type—such as *char*, *short*, *int*, or *long*—the integral type is implicitly converted to the floating-point type. For example, in an expression containing an *int* plus a *double*, the value of the *int* is converted to a *double*.

int+double

3. If the two values are both integral types, then the value of the one with the more limited range is converted to the other. For example, in an expression containing a *short* plus an *int*, the *short* is converted to an *int*. Similarly, in an expression containing an *int* plus an *unsigned int*, the *int* is converted to an *unsigned int*.

short+int

4. An enumerated type (see Section 11-10, "Enumerated Types" on page 617) is promoted to an appropriate integral type based on the range of the enumerated type.

In all cases, **implicit type conversion** is done according to the precedence and associativity of the operators. For example, in an expression with

```
float + int * double
```

the *int* in the multiply is first converted to a *double*. After the multiply is evaluated, we have

```
float + double
```

which requires that the *float* value be converted to a *double* for the addition. The type of the expression value is therefore a *double*.

le, 2004

Conversion in an assignment expression

ple, in an expression with

```
float + int * double
```

the *int* in the multiply is first converted to a *double*. After the multiply is evaluated, we have

```
float + double
```

which requires that the *float* value be converted to a *double* for the addition. The type of the expression value is therefore a *double*.

When an assignment expression contains mixed types, the evaluation of the expression on the right of the assignment operator follows the rules for evaluating expressions covered in the previous section. If, after evaluating the expression, the type of the expression value is not the same as the type on the left of the assignment operator, then the expression value must be implicitly converted to the type of the left-side variable. For example, if a floating-point expression is assigned to an integer variable as shown in the following example, the floating-point expression is converted to an integer before being stored.

```
int x = float-type expression;
```

Assignment of expressions containing mixed types are inherently dangerous. As programmers, it is our responsibility to ensure that the results are always valid. As long as the size of the variable on the left of the assignment operator is the same or larger than

Thomson Learning™
©Brooks/Cole, 2004

previous section. If, after evaluating the expression, the type of the expression value is not the same as the type on the left of the assignment operator, then the expression value must be implicitly converted to the type of the left-side variable. For example, if a floating-point expression is assigned to an integer variable as shown in the following example, the floating-point expression is converted to an integer before being stored.

```
int x = float-type expression;
```

Assignment of expressions containing mixed types are inherently dangerous. As programmers, it is our responsibility to ensure that the results are always valid. As long as the size of the variable on the left of the assignment operator is the same or larger than the size of the expression value, the conversion is safe. However, if the expression value is larger than can be stored in the variable on the left of the assignment operator, then an invalid result is stored. Similarly, if a floating-point expression value is stored in an integral type, the fractional part is lost even if the integral is large enough to hold the integral part of the floating-point number. Some compilers issue warning messages when the size of the receiving variable is smaller than the size of the expression value type.

EXPLICIT TYPE CONVERSION (CAST)

Rather than let the compiler implicitly convert data, you can convert data from one type to another yourself using **explicit type conversion**. C++ provides two types of explicit **casts**: static and dynamic. Explicit type conversions use the cast expression operator found at Priority 14 in the Precedence Table.

Static Cast

The syntax for the static cast is

```
static_cast<type> (expression)
```

In a static cast, the compiler converts the types at compilation time. Once the program is compiled, they cannot be changed. For example, given two integer variables, we can statically cast them to a float as shown in the following code.

```
average = static_cast<float> (totalScores) / numScores;
```

In this statement, there is an explicit conversion of `totalScores` to *float*, and then an implicit conversion of `numScores` so that it will match. The result of the divide is then a floating-point number to be assigned to `average`.

But beware! What would be the result of the following floating-point expression when the value of `a` is 3?

```
static_cast<float> (a / 10)
```

Are you surprised to find that the result is 0.0? Since there is no need to do any conversions to divide integer 3 by integer 10, C++ simply divides with an integer result, 0. The integer 0 is then explicitly converted to the floating-point 0.0.

Brooks/Cole Thomson Learning™
©Brooks/Cole, 2004

Are you surprised to find that the result is 0.0? Since there is no need to do any conversions to divide integer 3 by integer 10, C++ simply divides with an integer result, 0. The integer 0 is then explicitly converted to the floating-point 0.0.

Dynamic Cast

In a dynamic cast, the conversion is not determined until the program is run. To fully understand dynamic casts requires that we understand classes and dynamic memory allocation. Therefore, we defer its discussion until after we cover these subjects.

C-Style Cast

For compatibility with earlier versions of the language, the C++ compiler still accepts the C-style cast. Its format is shown in the next example.

```
(type) expression
```

The C-style cast should be avoided for two reasons. First, it is not clear whether the cast is static or dynamic. Because of the ambiguity, the determination of the cast type is left to discretion of the compiler. Second, we anticipate that the next release of the C++ standard could deprecate this format, which would require programs using C-style casts to be upgraded before they could be compiled.

One final thought about casts: Even when the compiler could correctly cast for you, it is sometimes better to code the cast explicitly to remind yourself that the cast is taking place. Program 3-8 demonstrates the use of explicit casts. In this program, we divide several mixed types. While the results are nonsense, they demonstrate the effect of casting.

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Explicit type casting operator

Type casting operators allow to convert a value of a given type to another type. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the floating-point number `3.14` to an integer value (`3`); the remainder is lost. Here, the typecasting operator was `(int)`. Another way to do the same thing in C++ is to use the functional notation preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int (f);
```

Both ways of casting types are valid in C++.

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

defined constant (#define) with
connection. This is an especially difficult error to decipher because you will not see it in your code—you see the code you wrote, not the code that the preprocessor substituted.

7. It is a compile error when the operand on the left of the assignment operator is not a variable. For example, `a + 3` is not a variable and cannot receive the value of `b * c`.

```
( a + 3 ) = b * c;
```

8. It is a compile error to use the increment or decrement operators with any expression other than a variable identifier. For example, the following code is an error:

```
( a + 3 )++
```

9. It is a compile error to use the modulus operator (%) with anything other than integers.
10. It is a logic error to use a variable before it has been assigned a value.
11. It is a logic error to modify a variable in an expression when the variable appears more than once. For example, the following code compiles and runs but will give inconsistent results:

```
a++ * (a + b)
a = a++ + b
```

Brooks/Cole
Thomson Learning™

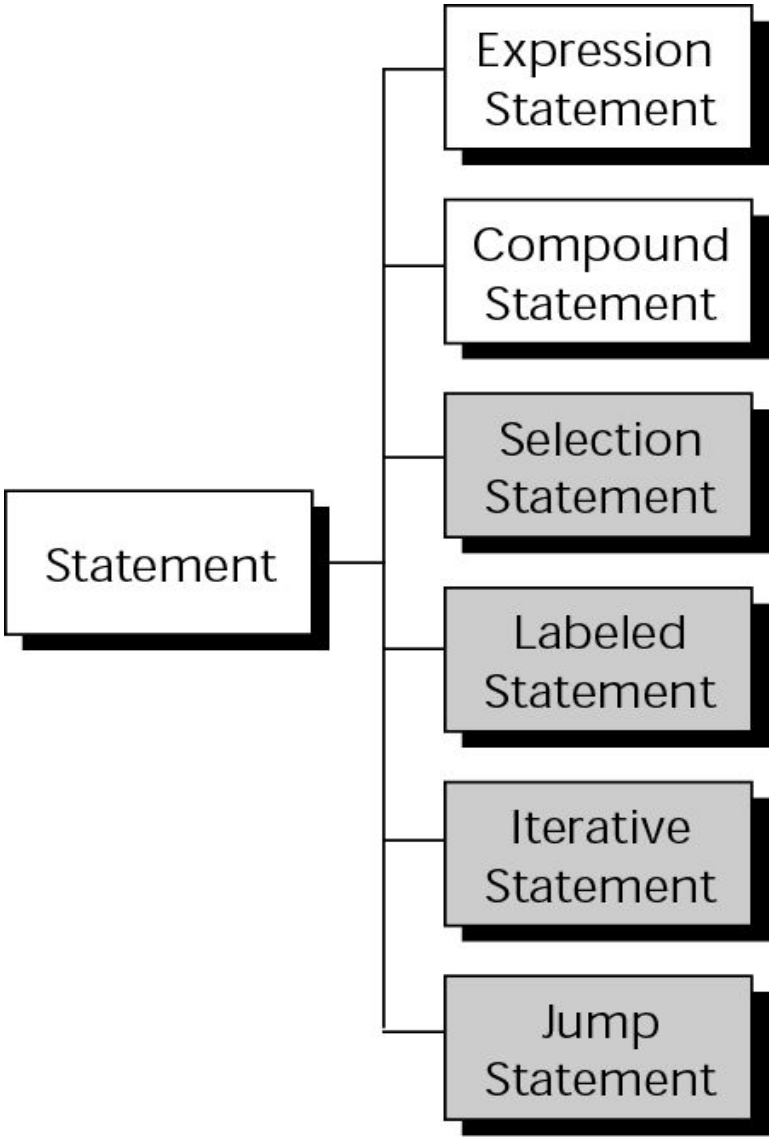
©Brooks/Cole, 2004

Statements

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Figure 3-12 Types of statements



Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

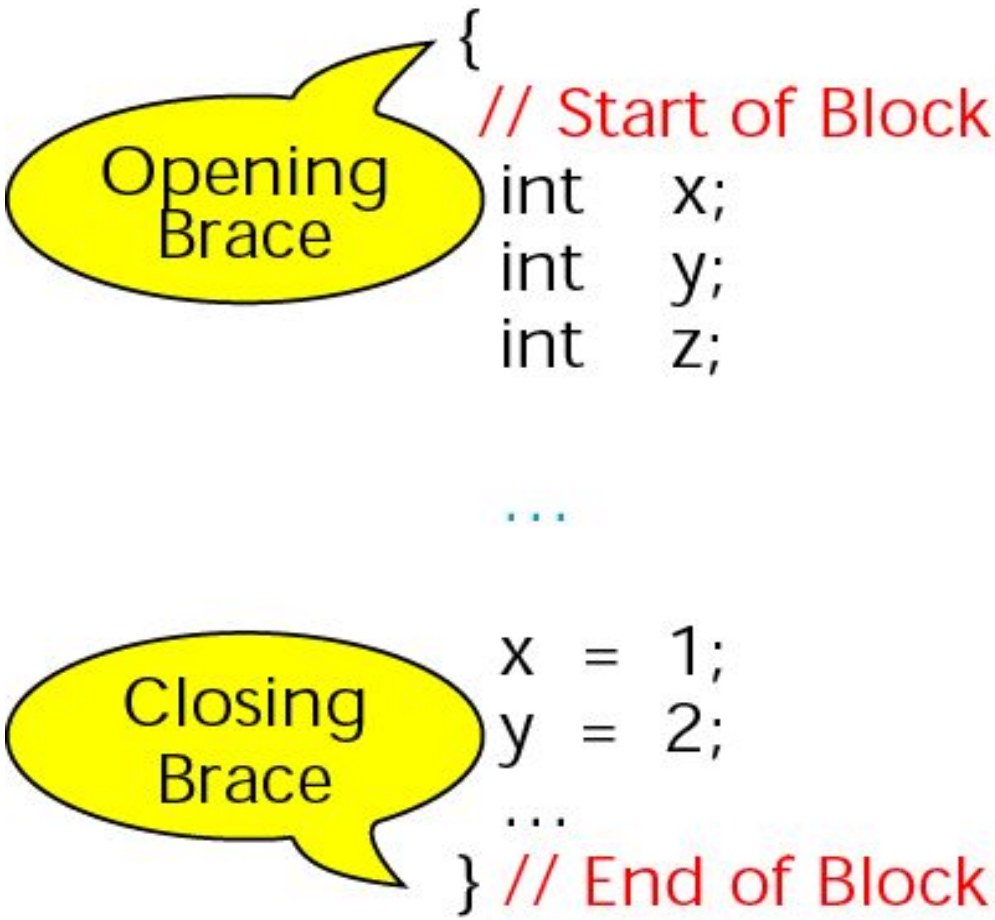
Note:

An expression statement is terminated with a semicolon. The semicolon is a terminator, and it tells the compiler that the statement is finished.

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Figure 3-13 Compound statement



©Brooks/Cole, 2004

Sample Programs

©Brooks/Cole, 2004

Introduction to strings

Fundamental types represent the most basic types handled by the machines where the code may run. But one of the major strengths of the C++ language is its rich set of compound types, of which the fundamental types are mere building blocks.

An example of compound type is the `string` class. Variables of this type are able to store sequences of characters, such as words or sentences. A very useful feature!

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type, the program needs to include the header where the type is defined within the standard library (header `<string>`):

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is a string";
  cout << mystring;
  return 0;
}
```

```
This is a string
```

As you can see in the previous example, strings can be initialized with any valid string literal, just like numerical type variables can be initialized to any valid numerical literal. As with fundamental types, all initialization formats are valid with strings:

```
string mystring = "This is a string";
string mystring ("This is a string");
string mystring {"This is a string"};
```

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and change its value during execution:

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is the initial string content";
  cout << mystring << endl;
  mystring = "This is a different string content";
  cout << mystring << endl;
  return 0;
}
```

```
This is the initial string content
This is a different string content
```

Note: inserting the `endl` manipulator **end**s the **l**ine (printing a newline character and flushing the stream).

The `string` class is a *compound type*. As you can see in the example above, *compound types* are used in the same way as *fundamental types*: the same syntax is used to declare variables and to initialize them.

For more details on standard C++ strings, see the `string` class reference.

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004

A C++ program

```
//include headers; these are modules that include
   functions that you may use in your
//program; we will almost always need to include the
   header that
// defines cin and cout; the header is called iostream.h
#include <iostream.h>

int main() {

//variable declaration
//read values input from user
//computation  and print output to user
return 0;
}
```

After you write a C++ program you compile it; that is, you run a program called **compiler** that checks whether the program follows the C++ syntax

- if it finds errors, it lists them
- If there are no errors, it translates the C++ program into a program in machine

Notes

- what follows after // on the same line is considered comment
- indentation is for the convenience of the reader; compiler ignores all spaces and new line ; the delimiter for the compiler is the semicolon
- all statements ended by semicolon
- **Lower vs. upper case matters!!**
  - Void is different than void
  - Main is different that main

The infamous **Hello world** program

When learning a new language, the first program people usually write is one that salutes the world :)

Here is the Hello world program in C++.

```
#include <iostream.h>
int main() {
   cout << “Hello world!”;

    return 0;
}
```

Input statements

cin >> variable-name;

Meaning: read the value of the variable called <variable-name> from the user

Example:

```
cin >> a;
cin >> b >> c;
cin >> x;
cin >> my-character;
```

Output statements

cout << variable-name;

Meaning: print the value of variable <variable-name> to the user

cout << "any message ";

Meaning: print the message within quotes to the user

cout << endl;

Meaning: print a new line

Example:

```
cout << a;
cout << b << c;
cout << "This is my character: " << my-character
<< " he he he"
 << endl;
```

Software Engineering and Programming Style

©Brooks/Cole, 2004

Note:

Blocks of code should be no longer than one screen.

Brooks/Cole
Thomson Learning™

©Brooks/Cole, 2004