4.3 Tokens, Identifiers, and Keywords

C++ program consists of many elements, which are identified by the compiler as *tokens*. Tokens supported in C++ can be categorized as keywords, variables, constants, special characters, and operators as shown in Figure 4.1.

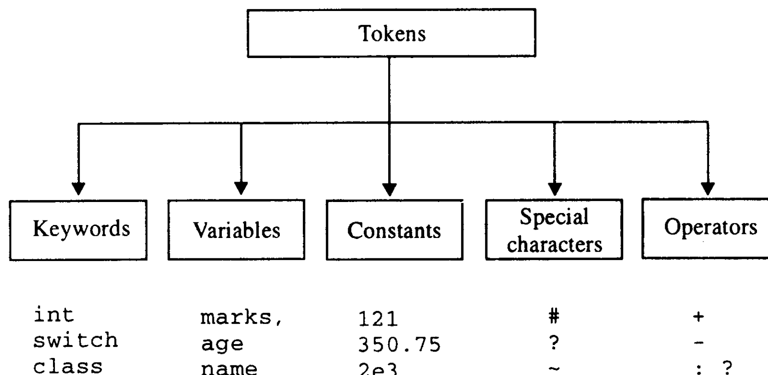


Figure 4.1: C++ tokens

In a C++ program, every word can be either classified as an identifier, or a keyword. As the name suggests, identifiers are used to identify or name variables, symbolic constants, functions, and so on. Keywords have predefined meanings and cannot be changed by the user. The following rules need to be followed while naming identifiers:

- Identifier name is formed by using alphabets, digits, or underscore characters.
- Identifier names must begin with an alphabet or underscore character.
- The maximum number of characters used in forming an identifier must not exceed 31 characters. Some compilers allow the identifier length to be more than 31 characters, however, only the first 31 characters are significant.
- C++ is case sensitive (since the upper and lower case letters are treated differently). For instance, names such as `rate`, `Rate`, and `RATE` are treated as different identifiers. It is a general practice to use lower or mixed case letters to name variables and functions, and upper case to name symbolic constants.
- C++ has standard identifiers called *keywords*. Keywords are declared by the C++ language and have a predefined meaning. Hence, they cannot be used for any other purpose other than that specified by the C++ language. The keywords supported by C language are shown in Table 4.2 and they are also available in C++ (C++ is a superset of C).

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table 4.2: Keywords common to C and C++

C++ Specific Keywords

There are several keywords specific to C++ which are listed in Table 4.3. These keywords primarily deal with classes, templates, and exception handling. For more details on keywords, refer to Appendix: *C++ Keywords and Operators.*

asm	new	template
catch	operator	this
class	private	throw
delete	protected	try
friend	public	virtual
inline		

Table 4.3: Keywords specific to C++

4.4 Variables

A variable is an entity whose value can be changed during program execution and is known to the program by a name. A variable definition associates a memory location to the variable name. A variable can hold only one value at a time during the program execution. Its value can be changed during the execution of the program. The various components associated with variables are the following:

- Data type - `char`, `int`, `float`, `date` (user defined), etc.
- Variable name - User view
- Binding address - Machine view
- Value - data stored in memory location

The relation among the above components is shown in Figure 4.2. In the statement

```
f = 1.8 * c + 32.
```

the symbols `f` and `c` are variables.

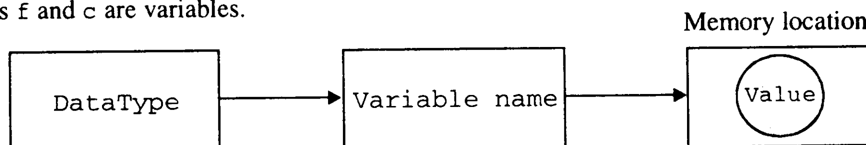


Figure 4.2: Components of variables

Variable Names

Variable names are identifiers used to name variables. They are the symbolic names assigned to the memory locations. A variable name consists of a sequence of letters and digits, the first one being a letter. The rules that apply to identifiers (given above), also apply to variable names. The following are some valid variable names:

```
i            sum           MAX        min
class_mark   student_name  emp_num    fact_recur
classMark    StudentName   rank1  _x1   _num
```

The following are some invalid variable names (with reasons given along side):

`a's`	illegal character(')
`fact recur`	blank not allowed
`class-mark`	illegal character(-)
`5root`	first character should be a letter
`student,rec`	comma not allowed

4.5 Data Types and Sizes

C++ supports a wide variety of data types and the programmer can select the type appropriate to the needs of the application. However, storage representation and machine instructions to manipulate each data type differ from machine to machine, although C++ instructions are identical on all machines. C++ supports the following classes of data types:

- Primary (fundamental) data types
- Derived data types
- User-defined data types

The primary data types and their extensions is the subject of this chapter. Derived data types such as arrays and pointers, and user defined data types such as structures and classes are discussed in the later chapters.

C++ language supports the following basic data types:

`char`	a single byte that can hold one character.
`int`	an integer.
`float`	a single precision floating point number.
`double`	a double precision floating point number.

Further, applying qualifiers to the above basic types yields additional types. A qualifier alters the characteristics such as the size or sign of the data types. The qualifiers that alter the size are `short` and `long`. These qualifiers are applicable to integers, and yield two more types:

`short int`	Integer represented by 16 bits irrespective of machine type.
`long int`	Integer represented 32 bits irrespective of machine type.

The exact sizes of these data types depend on the compiler as shown in Table 4.4.

Data Type	Data Size (bytes)	Minimum value	Maximum value
`char`	1	-128	127
`short`	2	-327868	327867
`int`	2 (16 bit compiler)	-327868	327867
	4 (32 bit compiler)	-2147483648	2147483647
`long`	4	-2147483648	2147483647
`float`	4	-3.4E-38	3.4E+38
`double`	8	-1.7E-308	1.7E+308
`long double`	10	-3.4E-4932	1.1E+4932

Table 4.4: Data types and their size

The qualifier `long` can also be used along with the double precision floating point type:

`long double` — an extended precision floating point number.

The sign qualifiers are `signed` and `unsigned`. The sign qualifiers are applicable to the integer data types (`int`, `short int`, and `long int`) resulting in six additional data types given below:

```
signed short int
unsigned short int
signed int
unsigned int
signed long int
unsigned long int
```

Qualifiers are also applicable to the char data type as follows:

```
signed char
unsigned char
```

Size qualifiers (short and long) cannot be applied to the char and float data types and sign qualifiers (signed and unsigned) cannot be applied to float, double, and long double.

4.6 Variable Definition

A variable must be defined before using it in a program. It reserves memory required for data storage and associates it with a symbolic name. The syntax for defining variables is shown in Figure 4.3. The variable name can be any valid C++ identifier except the reserved words. The data type can be any primitive or user-defined data type such as int, float, double, and so on.

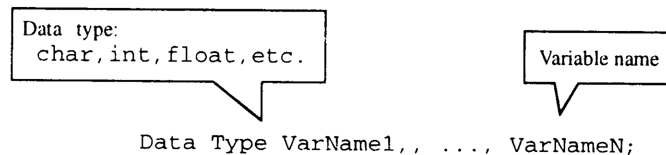


Figure 4.3: Syntax of variable definition

The following are some of the valid variable definition statements:

```
int a;                  // a is an integer variable
int b, c, d;            // b, c, and d are valid integer variables
float total, length;    // total and length are valid real variables
double sum1, product;   // sum1 and product are valid double variables
```

Variables can be defined at the point of their usage as follows:

```
int c;
cout << "Hello World";
int d = 10;
for( int i = 0; i < 10; i++ )
   cout << i;
```

Note that the variables d and i are defined at the point of their usage.

4.7 Variable Initialization

In C++, a variable can be assigned with a value during its definition, or during the execution of a program. The assignment operator (=) is used in both the cases. A variable can be initialized during its definition using any one of the following syntax:

data-type variable-name = constant-value;
data-type variable-name(constant-value);

This syntax is most commonly used since it avoids chances of using uninitialized variables leading to runtime errors. The following statements initialize variables during their definition:

```
int a = 20;                  // or int a(20);
float c = 1920.9, d = 4.5;   // or float c(1920.9), d(4.5);
double g = 123455.56;
```

The value to be initialized to a variable at the time of definition must be known while writing the program i.e., it must be a constant value or must have been assigned at runtime before its definition as follows:

```
int i = 3;
int k = i + 3;
```

A variable which is initialized at its definition is called *value-initialized variable*. However, its value can be modified during the program execution at a later point. When multiple variables are being declared in a single statement, initialization is carried out in the following way:

```
int i = 10, j = 5;
```

The right side of the assignment operator can be any valid expression as given below:

```
int k = i / j;
```

It assigns the value 2 to `k` if `i=10` and `j=5`.

The variables can be initialized by using any valid expression at runtime. The general format is as follows:

variable-name = expression;

The *expression* can be a constant, variable name, or variables and/or constants connected by using operators (mathematical expression). For example,

```
a = 10;
a = b;
a = c+d-5;
```

where the symbols + and - represent addition and subtraction operation respectively. The program `show1.cpp` illustrates the initialization of variables in the definition or during its execution.

```
// show1.cpp: variable definition and assignment
#include <iostream.h>
void main()
{
   int a, b;         // integer type variable definition
   int c = 100;      // variable definition and initialization
   float distance;   // floating-point type variable definition
   // initialization during execution time
   a = c;
   b = c + 100;
   distance = 55.9;
   // display contents of the variables
   cout << "a = " << a << "\n";
   cout << "b = " << b << "\n";
   cout << "c = " << c << "\n";
   cout << "distance = " << distance;
}
```

Run

```
a = 100
b = 200
c = 100
distance = 55.9
```

In `main()`, the statement

```
int c = 100;
```

defines a variable called c and initializes it with the constant integer value 100. The statement

```
a = c;
```

reads the contents of the variable c and assigns it to the variable a. The statement

```
b = c + 100;
```

adds the contents of the variable c with the numeric constant 100, and assigns the result to the variable b. The statement

```
distance = 55.9;
```

assigns the floating-point constant value 55.9 to the variable distance. The statement

```
cout << "a = " << a << "\n";
```

displays a message a = followed by the contents of the variable a and then a newline. Input and output operations in C++ have already been discussed in Chapter 2. For more information refer to the chapter, *Streams Computation with Console*.

4.8 Characters and Character Strings

A character variable can hold a single character. For instance, the statement

```
char code = 'R';
```

assigns the character constant R to the variable code. The value stored in the variable code is the ASCII equivalent of the character R. Note that the character constant is enclosed in a pair of single quotes and each character representation requires 8 bits (one byte).

A sequence of characters is called a string. String constants are enclosed in double-quotes as follows:

```
"Hello World"
```

String constants are useful while conveying some messages to the user. For instance, the statement

```
cout << "I love C++ programming";
```

displays the message indicated by the string constant as follows:

```
I love C++ programming
```

In C++, characters can be treated like integers. A character variable holds one character such as a letter, a digit, or a punctuation mark. These characters are represented in memory by a number, called the code for the character. For example, the code for the letter A may be 65, that for letter B may be 66, and so on.

Actually, any number can represent the letter A, any other number can be used for B and so on, but these numbers should be fixed by a coding convention. For example, when the computer wants the printer to print the letter A, it actually sends the number 65 to the printer. The important point here is that the printer accepts the number 65 and prints the letter A. Hence, the printer must also use the same code to represent character as that is used by the computer. This requirement led to the establishment of a standard called ASCII (American Standard Code for Information Interchange). ASCII codes are widely used all over the world to represent various symbols in a computer.

The program ascii.cpp reads the ASCII code of a character and prints out the symbol associated with the code.

```
// ascii.cpp: ASCII code example
#include <iostream.h>
void main()
{
   int code;
   char symbol;
   cout << "Enter an ASCII code (0 to 127): ";
   cin >> code;    // reads integer value
   symbol = code; // store into character variable
   cout << "The symbol corresponding to " << code  << " is " << symbol;
}
```

Run1

```
Enter an ASCII code (0 to 127): 65
The symbol corresponding to 65 is A
```

Run2

```
Enter an ASCII code (0 to 127): 67
The symbol corresponding to 67 is C
```

In `main()`, the statement

```
symbol = code;
```

assigns the value of the integer variable code to the character variable `symbol`. In the output statement

```
cout << "The symbol corresponding to " << code  << " is " << symbol;
```

the character variable `code` forces `cout` to display the ASCII symbol corresponding to the value stored in it.

A string in C++ is just a sequence of consecutive characters in memory, the last one being the null character. A null character has an ASCII code 0 and is called the *end-of-string* marker in C++. For instance, consider the following string constant:

```
"I love C++ programming"
```

In memory, it is stored as a sequence of bytes as shown in Figure 4.4. Each location holds ASCII equivalent of the respective character. The null character (a byte with value zero) is placed at the end of the string. It serves to terminate the string. i.e., to mark the end of the string.

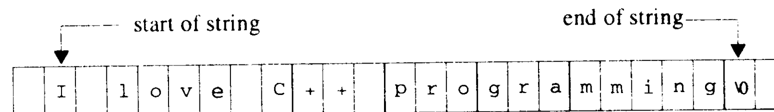


Figure 4.4: String representation in memory

4.9 Operators and Expressions

C++ operators are special characters which instruct the compiler to perform *operation* on some *operands*. Operation instructions are specified by operators, while operands can be variables, expressions, or literal values. Some operators operate on a single operand and they are called *unary operators*. Some operators are indicated before operands and they are called *prefix operators*. Others, indicated after the

operand are called *postfix operators*. For instance, expressions ++i or i++ use unary prefix and postfix operators respectively. Most operators are embedded between the two operands, and they are called *infix binary operators*. An expression a+b uses the binary plus operator. C++ has even an operator that takes three operands, called a *ternary operator*. Unification of the operands and the operators results in the formation of *expressions*.

Types of Operators

In C++, operators can be classified into various categories based on their utility and action as follows:

- Arithmetic operators
- Relational operators
- Logical operators
- Assignment operators
- Increment and decrement operators
- Conditional operator
- Bitwise operators
- Special operators

An expression is a combination of variables, constants and operators written according to the syntax of the language. In C++, every expression evaluates to a value. i.e., every expression results in some value of a valid data type, that can be assigned to a variable. The following are some of the valid expressions:

```
a+b
a+200+40
c+b*z
z+20
total+20+c/3
```

Expressions having operands of different data types are called *mixed-mode expressions*. Consider the following statements:

```
int a, c;
float d, e;
```

The expression

```
(a+d+e+c)
```

is called *mixed-mode expression*, since it contains variables of types; integer and floating-point.

Assignment Operator =

As in most other languages, the equal (=) sign is used for assigning a value to a variable. It has the following syntax:

variable = expression;

The left hand side has to be a variable (often called *lvalue*) and the right hand side has to be a valid expression (often called *rvalue*). The following are some valid assignment statements:

```
a = 32000;              // rvalue is constant
b = z + 10 * a;         // rvalue is expression
c = sqrt( 20.2 );       // rvalue is function
```

The program temper.cpp illustrates the conversion of temperature value in fahrenheit to centigrade and vice-versa using the following relation:

*fahrenheit = 1.8 * centigrade + 32*

```
// temper.cpp: conversion of centigrade to fahrenheit and vice-versa
#include <iostream.h>
void main()
{
   float c, f;
   cout << "Enter temperature in celsius: ";
   cin >> c;
   f = 1.8 * c + 32;
   cout << "Equivalent fahrenheit = " << f << endl;
   cout << "Enter temperature in fahrenheit: ";
   cin >> f;
   c = (f - 32) / 1.8;
   cout << "Equivalent celsius = " << c;
}
```

Run

```
Enter temperature in celsius: 5
Equivalent fahrenheit = 41
Enter temperature in fahrenheit: 40
Equivalent celsius = 4.444445
```

4.10 Qualifiers

Qualifiers modify the behavior of the variable type, to which they are applied. Qualifiers can be classified into two types:

- Size qualifiers
- Sign qualifiers

Consider the variable definition statement:

```
int i;
```

It specifies that `i` is an integer, which takes both positive and negative values. That is, `i` is a signed integer by default. The above definition could also be written as:

```
signed int i;
```

Prefixing of the qualifier `signed` explicitly, is unnecessary, since `int` data type definitions are `signed` by default. If the variable `i` is used to hold only positive values (for example, if it is used to hold the number of students in a class), it can be defined as follows:

```
unsigned int i;
```

Here, the qualifier `unsigned` is applied to the data type `int`. This qualifier modifies the behavior of the integer so that a variable of this type always contains a positive value.

`int` with Size Qualifiers

Size qualifiers alter the size of the basic type. There are two size qualifiers that can be applied to integers (i.e., to the basic type int): `short` and `long`. In any ANSI C++ compiler, the sizes of `short int`, `int` and `long int` have the following specification:

- The size of a `short int` is 16 bits.
- The size of an `int` must be greater than or equal to that of a `short int`.
- The size of a `long int` must be greater than or equal to that of an `int`.
- The size of a `long int` is 32 bits.

In most compilers available on DOS, the size of a `short int` and an `int` are the same (16 bits). A `long int` occupies 32 bits. But in 32-bit compilers such as GNU C/C++, an `int` and a `long int` are of the same size (32 bits), while a `short int` is 16 bits. On the other hand, almost all UNIX compilers have the size of `int` as 16 bits, `int` and `long int` being 32 bits.

sizeof operator

The operator `sizeof` returns the number of bytes required to represent a data type or variable. It has the following forms:

sizeof(data-type)
sizeof(variable)

The data type can be standard or user defined data type. The following statements illustrate the usage of `sizeof` operator:

```
int i, j;
float c;
sizeof( int )        returns 2 or 4 depending on compiler implementation
sizeof( float )      returns 4
sizeof( long )       returns 4
sizeof( i )          returns 2 or 4 depending on compiler implementation
sizeof( c )          returns 4
```

The program `size.cpp` determines the size of an integer and its variants. It uses the function `sizeof()`, which gives the size of any data type in bytes.

```
// size.cpp: size qualifiers and sizeof operator
#include <iostream.h>
void main()
{
   cout << "sizeof( char ) = " << sizeof(char) << endl;
   cout << "sizeof( short )  = " << sizeof(short) << endl;
   cout << "sizeof( short int ) = " << sizeof(short int) << endl;
   cout << "sizeof( int ) = " << sizeof(int) << endl;
   cout << "sizeof( long ) = " << sizeof(long) << endl;
   cout << "sizeof( long int ) = " << sizeof(long int) << endl;
   cout << "sizeof( float ) = " << sizeof(float) << endl;
   cout << "sizeof( double ) = " << sizeof(double) << endl;
   cout << "sizeof( long double ) = " << sizeof(long double) << endl;
}
```

Run1

```
sizeof( char ) = 1
sizeof( short )  = 2
sizeof( short int ) = 2
sizeof( int ) = 2
sizeof( long ) = 4
sizeof( long int ) = 4
sizeof( float ) = 4
sizeof( double ) = 8
sizeof( long double ) = 10
```

Run2

```
sizeof( char ) = 1
sizeof( short )  = 2
sizeof( short int ) = 2
sizeof( int ) = 4
sizeof( long ) = 4
sizeof( long int ) = 4
sizeof( float ) = 4
sizeof( double ) = 8
sizeof( long double ) = 10
```

Note: The output displayed in *Run1* is generated by executing the program on the DOS system compiled with Borland C++ compiler. *Run2* is generated by executing the program on the UNIX system.

The name of the variable type is passed to the `sizeof` operator in parentheses. The number of bytes occupied by a variable of that type is given by the `sizeof` operator. The `sizeof` operator is very useful in developing portable programs. It is a bad practice to assume the size of a particular type. since its size can vary from compiler to compiler.

The `sizeof` operator also takes variable names and returns the size of the variable given, in bytes. For example, the statements

```
int i;
cout << "The size of i is " << sizeof(i);
```

will output

```
The size of i is 2
```

The result of `sizeof(int)` or `sizeof(i)` will be the same, since `i` is an `int`. However, it is a better practice to pass a variable name to the `sizeof` operator; if the data-type of `i` has to be changed later, then rest of the program need not be modified.

Size qualifiers, as applied to `double`

The type qualifier `long` can also be applied on the data type `double`. A variable of type `long double` has more precision when compared to a variable of type `double`. C++ provides three data types for real numbers: `float`, `double` and `long double`. In any C++ compiler, the precision of a `double` is greater than or equal to that of a `float`, and the precision of a `long double` is greater than or equal to that of a `double`. i.e., precision-wise,

precision of long double >= precision of double >= precision of float

To find out the actual range that a compiler offers, a program(mer) can refer to the constants defined in the header file `float.h`.

Sign Qualifiers

The keywords `signed` and `unsigned` are the two sign qualifiers which inform the compiler whether a variable can hold both negative and positive numbers, or only positive numbers. These qualifiers can be applied to the data types `int` and `char` only. For example, an unsigned integer can be declared as

```
unsigned int i;
```

As mentioned earlier, `signed int` is the same as `int` (i.e., `int` is signed by default). The `char` data type can be treated as either `signed char`, or `unsigned char`, and the exact representation is compiler dependent.

4.11 Arithmetic Operators

The C++ language has both unary and binary arithmetic operators. Unary operators are those, which operate on a single operand whereas, binary operators operate on two operands. The arithmetic operators can operate on any built-in data type. Arithmetic operators and their meaning are shown in Table 4.5. Note that, C++ has no operator for exponentiation. However, a function `pow(x, y)` exists in `math.h` which returns x^y.

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo Division

Table 4.5: Arithmetic operators

Unary Minus Operator (Negation)

The unary minus operator can be used to negate the value of a variable. It is also used to specify a negative number; here a minus (-) sign is prefixed to the number. Consider the following examples

1.
```
int x = 5;
y = -x;
```

The value of x after negation is assigned to y i.e., y becomes -5.

2.
```
int x = -5;
sum = -x;
```

The value of sum is +5. The unary minus operator has the effect of multiplying its operand by -1.

3. The use of unary + operator does not serve any purpose. However, it can be used as follows:

```
a = +100;
```

By default, numeric constants are assumed to be positive.

Binary Operators

Binary arithmetic operators such as +, -, *, etc., require two operands of standard data types. Depending on the data types of the operands, these operators perform either integer or floating-point arithmetic operation.

Integer arithmetic: When the two operands say x and y are defined as integers, any arithmetic operation performed on these operands is called integer arithmetic, which always yields an integer result.

Example:

Let x and y be defined by the statement:

```
int x = 16, y = 5;
```

Then the integer arithmetic operations yield the following results:

```
x + y = 21
x - y = 11
x * y = 80
```

`x / y = 3` The result is truncated, the decimal part is discarded.
`x % y = 1` The result is the remainder of the integer division. The sign of the result is always the sign of the first operand.

In integer division operation, the result is truncated towards the lower value if both the operands are of the same sign, and is dependent on the machine if one of the operands is negative.

Example:

```
6 / 8 = 0
-6 / -8 = 0
-6 / 8 = 0 or -1
```
The result is machine dependent.

Floating-point arithmetic: Floating-point arithmetic involves operands of real type in decimal or exponential notation. The floating point results are rounded off to the number of significant digits specified, and hence the final value is only an approximation of the correct result. The remainder operator `%` is not applicable to floating point operands.

Example:

Let `a` and `b` be defined by the statement

```
float a = 14.0, b = 4.0;
```

and `p`, `q` and `r` be floating point variables; then the floating point arithmetic operations will yield the following results

```
p = a / b = 3.500000
q = b / a = 0.285714
r = a + b = 18.00000
```

Mixed mode arithmetic: In mixed mode arithmetic, if either one of the operands is real, the resultant value is always a real value. For example, 35 / 5.0 = 7.0. Here, since 5.0 is a double constant, 35 is converted to a double and the result is also a double

The expression

```
x % y
```

produces the remainder when `x` is divided by `y` (it returns 0 when `y` divides `x` exactly). The program `modulus.cpp` illustrates the use of the *modulus operator.*

```
// modules.cpp: computation of remainder of division operation
#include <iostream.h>
void main()
{
   int numerator, denominator;
   float result, remainder;
   cout << "Enter numerator: ";
   cin >> numerator;
   cout << "Enter denominator: ";
   cin >> denominator;
   result = numerator / denominator;
   remainder = numerator % denominator;
   cout << numerator << "/" << denominator << " = " << result << endl;
   cout << numerator << "%" << denominator << " = " << remainder;
}
```

Run

```
Enter numerator: 12
Enter denominator: 5
12/5 = 2
12%5 = 2
```

An arithmetic expression without parentheses will evaluate from left to right using the following rules of precedence for operators:

High priority: `* / %`
Low priority: `+ -`

The basic evaluation process requires two passes. During the first pass, the highest priority operators are applied as they are encountered and in the next pass, the low priority operators are applied. Consider the following statement:

```
a = b + c * 5 + d / 2 - 3;
```

When `b = 5, c = 2, d = 10`, the statement becomes

```
a = 5 + 2 * 5 + 10 / 2 - 3;
```

It is evaluated as follows:

First pass:

```
step 1: a = 5 + 10 + 10 / 2 - 3;
step 2: a = 5 + 10 + 5 - 3;
```

Second pass:

```
step 3: a = 15 + 5 - 3;
step 4: a = 20 - 3;
step 5: a = 17;
```

These evaluation steps are shown in Figure 4.5, which illustrates the hierarchy of operators. When parentheses are used, the expression within the innermost parentheses gains highest priority.

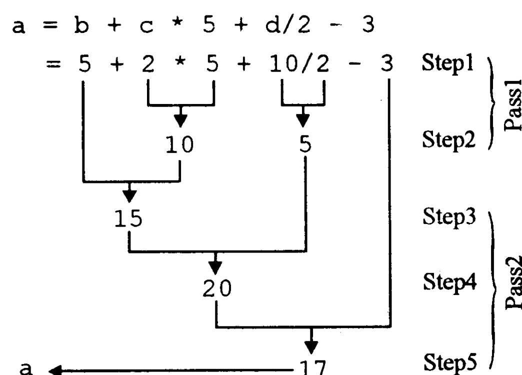
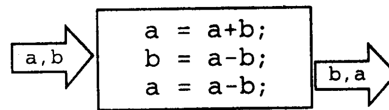
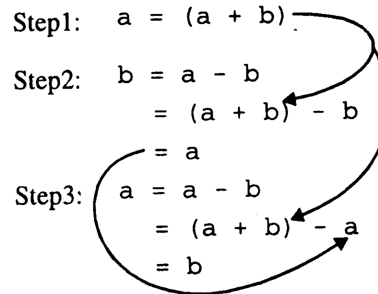


Figure 4.5: Hierarchy of operations

A program for swapping two integer numbers without using a temporary variable, is listed in `notemp.cpp`. The steps involved are illustrated in Figure 4.6.



(a) Steps for swapping two numbers



(b) Swapping steps derivations

Figure 4.6: Swapping without using temporary variable

```
// notemp.cpp: swapping two numbers without using temporary variable
#include <iostream.h>
void main()
{
   int a, b;
   cout << "Enter two integers <a, b>: ";
   cin >> a >> b;
   a = a + b;
   b = a - b;
   a = a - b;
   // logic for swapping a and b ends here
   cout << "Value of a and b on swapping in main(): " << a << " " << b;
}
```

Run

```
Enter two integers <a, b>: 10 20
Value of a and b on swapping in main(): 20 10
```

4.12 Relational Operators

A relational operator is used to make comparisons between two expressions. All these operators are binary and require two operands. Logically similar quantities are often compared for taking decisions. These comparisons can be done with the help of relational operators as shown in Table 4.6. Each one of these operators compares its left hand side operand with its right hand side operand. The whole expression involving the relational operator then evaluates to an integer. It evaluates to zero if the condition is false, and non-zero value if it is true.

Operator	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

Table 4.6: Relational operators

In order to understand the relational operators, it is necessary to know the basics of an `if` statement. (The `if` statement is elaborately discussed in the next chapter.) If condition-expression is true, it executes the *then-part* only, otherwise, it evaluates the *else-part*, as shown below:

```
if( condition )
   statement1;        // executed when condition is true
else
   statement2;        // executed when condition is false
```

The program `relation.cpp` illustrates the use of the relational operators in taking decisions.

```
// relation.cpp: relational operator usage
#include <iostream.h>
void main()
{
   int my_age, your_age;
   cout << "Enter my age: ";
   cin >> my_age;
   cout << "Enter your age: ";
   cin >> your_age;
   if( my_age == your_age )
      cout << "We are born in the same year.";
   else
      cout << "We are born in different years";
}
```

Run1

```
Enter my age: 25
Enter your age: 25
We are born in the same year.
```

Run2

```
Enter my age: 25
Enter your age: 21
We are born in different years
```

In `main()`, the statement

```
if( my_age == your_age )
```

has the expression `my_age == your_age` as a conditional expression. It returns *true* if `my_age`

and `your_age` are equal, otherwise it returns *false*. Note that 0 is treated as *false*, whereas any non-zero value is treated as *true*.

Note that in C++, the operator for testing equality is == (two = signs placed together). One of the most common mistakes is to use a single = sign, to test for equality. For example, consider the statement

```
if( my_age = your_age )
```

The conditional expression evaluates to *true* even if `my_age` and `your_age` are unequal (except when `your_age` is equal to zero). This happens because the result of an assignment operator is the assigned value itself. (Consider `my_age` is 25 and `your_age` is 21.) Here, the value of `your_age` (25) is assigned to `my_age`, and the assignment expression evaluates to 25, which is non-zero. Since any non-zero value is considered to be *true*, the statements following the if (then-part) are executed.

While using the relational operators, the fact whether the numbers being compared are signed or not becomes important. Neglecting this fact can lead to hard-to-find errors. The program `char1.cpp` illustrates the use of `char` type variables as 8-bit integers.

```
// char1.cpp: Using char as an 8-bit integer
#include <iostream.h>
void main()
{
   // Integer value being assigned to a char
   char c = 255;
   char d = -1;
   if( c < 0 )
      cout << "c is less than 0\n";
   else
      cout << "c is not less than 0\n";
   if( d < 0 )
      cout << "d is less than 0\n";
   else
      cout << "d is not less than 0\n";
   if( c == d )
      cout << "c and d are equal";
   else
      cout << "c and d are not equal";
}
```

Run

```
c is less than 0
d is less than 0
c and d are equal
```

In `main()`, the statement

```
if( c == d )
```

treats `c` and `d` as equal, although `c` is assigned with 255 and `d` is assigned with -1. It is because both of them are treated as signed numbers by default. This can be overcome by explicitly defining variables of type `char` as signed or unsigned while using them as 8-bit integers, as illustrated in the program `char2.cpp`.

```
// char2.cpp: Using char as an 8-bit integer
#include <iostream.h>
void main()
{
   // Integer value being assigned to a char
   unsigned char c = 255;
   char d = -1;
   if( c < 0 )
      cout << "c is less than 0\n";
   else
      cout << "c is not less than 0\n";
   if( d < 0 )
      cout << "d is less than 0\n";
   else
      cout << "d is not less than 0\n";
   if( c == d )
      cout << "c and d are equal";
   else
      cout << "c and d are not equal";
}
```

Run

```
c is not less than 0
d is less than 0
c and d are not equal
```

4.13 Logical Operators

Any expression that evaluates to zero denotes a FALSE logical condition, and that evaluating to non-zero value denotes a TRUE logical condition. Logical operators are useful in combining one or more conditions. C++ has three logical operators shown in Table 4.7.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Table 4.7: Logical operators

The first two operators && and || are binary, whereas the exclamation (!) is a unary operator and is used to negate a condition. The result of logical operations when applied to operands with all possible values, is shown in Table 4.8.

Logical AND: For example, consider the following expression

```
a > b && x == 10
```

The expression on the left is `a > b` and that on the right is `x == 10`. The whole expression evaluates to true only if both expressions are true (if a is greater than b as well as x is equal to 10)

T- True, F- False

operand1 a	operand2 b	~a	~b	a && b	a || b
F	F	T	T	F	F
F	T	T	F	F	T
T	F	F	T	F	T
T	T	F	F	T	T

Table 4.8: Truth table for logical operator

Logical OR: Consider the following example involving the | | operator.

```
a < m || a < n
```

The expression is *true* if one of them is *true*, or if both of them are *true* i.e., if the value of a is less than that of m, or if it is less than n. Needless to say, it evaluates to *true* when a is less than m and n.

Logical NOT: The ! (NOT) operator takes a single expression and evaluates to *true* if the expression is *false*, and evaluates to *false* if the expression is *true*. In other words, it just reverses the value of the expression. For example, consider:

```
!( x >= y )
```

It has the same meaning as

```
x < y
```

The ! operator can be conveniently used to replace a statement such as

```
if( a == 0 )
```

by the statement

```
if( !a )
```

The expression !a evaluates to *true* if the variable a holds zero, *false* otherwise.

The unary negation operator (!) has a higher precedence amongst these, followed by the logical AND (&&) operator and then the logical OR (| |) operator, and are evaluated from left to right.

The logical operator is used to connect various conditions to determine whether a given year is a leap year or not. A year is a leap year if it is divisible by 4 but not by 100, or that is divisible by 400. The program leap.cpp illustrates the use of the modulus operator.

```
// leap.cpp: detects whether year is leap or not
#include <iostream.h>
void main()
{
   int year;
   cout << "Enter any year: ";
   cin >> year;
   if( (year % 4 == 0 && year % 100 != 0 ) || (year % 400 == 0) )
      cout << year << " is a leap year";
   else
      cout << year << " is not a leap year";
}
```

Run1

```
Enter any year: 1996
1996 is a leap year
```

Run2

```
Enter any year: 1997
1997 is not a leap year
```

In `main()`, the statement

```
if( (year % 4 == 0 && year % 100 != 0 ) || (year % 400 == 0) )
```

can be replaced by

```
if( (!(year % 4) && year % 100 != 0 ) || !(year % 400) )
```

4.14 Bit-wise Operators

The support of bit-wise manipulation on integer operands is useful in various applications. Table 4.9 shows the bit-wise operators supported by C++. To illustrate these operators with examples, assume that a, b and c are defined as integer variables as follows:

```
int a = 13, b = 7, c;
```

Consider the variables a, b, and c as 16-bit integers, and the value stored in a and b have the following representation in the binary form:

The binary representation of a is `0000 0000 0000 1101`
The binary representation of b is `0000 0000 0000 0111`

(The spaces appearing after every 4 bits are only for clarity. Actually, the integers are merely 16 continuous bits.)

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise EX-OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Table 4.9: Bit-wise operators

(i) Logical Bit-wise Operators

Logical bit-wise operators perform logical operations such as AND, OR, EX-OR, NOT between corresponding bits of operands (if binary) and negation of bits (if unary).

Unary Operator : One's Complement Operator (~)

The complement operator causes the bits of its operand to be inverted, i.e., 1 becomes 0 and 0 becomes 1. For instance, the largest possible number, which can be stored in an unsigned integer can be found as follows. When one's complement operator is applied on this word holding zero, all the bits will be

inverted to ones and a new value becomes the largest possible number. The program `large.cpp` illustrates this conversion process.

```
// large.cpp: detects largest possible unsigned integer
#include <iostream.h>
int main()
{
   unsigned u = 0;
   cout << "Value before conversion: " << u << endl;
   u = ~u;
   cout << "Value after conversion : " << u << endl;
   return 0;
}
```

Run1

```
Value before conversion: 0
Value after conversion : 65535
```

Run2

```
Value before conversion: 0
Value after conversion : 4294967295
```

☞ *Run1* is executed on the MS-DOS based machine using a 16-bit compiler.
Run2 is executed on the UNIX based machine using a 32-bit compiler.

Binary Logical Bit-wise Operators

There are three binary logical bit-wise operators: & (and), | (or) and ^ (exclusive or). The operations are carried out independently on each pair of the corresponding bits in the operands, i.e. the bit 1 of operand1 is logically operated with the bit 1 of operand 2. The operations using these operators are discussed in the following sections.

Bitwise AND: The statement

```
c = a & b;
```

makes use of the bitwise AND operator. After this statement is executed, each bit in `c` will be 1 only if the corresponding bits in both `a` and `b` are 1. For example, the rightmost bit of both integers is 1, and hence the rightmost bit in `c` is 1. The next bit is 0 in `a` and 1 in `b`. Hence the second bit (from the right) in `c` is 0. Applying the same reasoning for all the bits in each one of the integers, the value of `c` after the above statement is executed will be `0000 0000 0000 0101` which, in decimal is 5 and is illustrated below:

Bitwise AND operator: a & b

```
    a   0000 0000 0000 1101
    b   0000 0000 0000 0111
a & b   0000 0000 0000 0101
```

Bitwise OR: The statement

```
c = a | b;
```

makes use of the bitwise OR operator. After this statement is executed, a bit in `c` will be 1 whenever at least one of the corresponding bits in either `a` or `b` is 1. In the example given below, the value of `c` will be `0000 0000 0000 1111` i.e., decimal 15 and is illustrated below:

Bitwise OR operator: a | b

```
    a   0000 0000 0000 1101
    b   0000 0000 0000 0111
a | b   0000 0000 0000 1111
```

Bitwise XOR: The statement

```
c = a ^ b;
```

makes use of the bitwise XOR operator. After this statement is executed, a bit in `c` will be 1 whenever the corresponding bits in `a` and `b` differ. So in the example given below, the value of `c` will be `0000 0000 0000 1010` which, in decimal is 10 and is illustrated below:

Bitwise EX-OR operator: a ^ b

```
    a   0000 0000 0000 1101
    b   0000 0000 0000 0111
a ^ b   0000 0000 0000 1010
```

(ii) Shift Operators

There are two shift operators in C++: left shift (<<) and right shift (>>). These are binary operators and have the following syntax:

```
operand << count        for left shift
operand >> count        for right shift
```

The first operand is the value which is to be shifted. The second is the number of bits by which it is shifted. The left shift operator moves the `count` number of bits to the left, whereas the right shift operator moves the `count` number of bits to the right. The leftmost or the rightmost bits are shifted out and are lost.

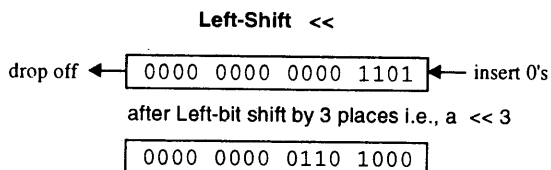
Left Shift Operator

Consider the statement

```
c = a << 3;
```

The value in the integer `a` is shifted left by three bit positions. The result is assigned to the integer `c`. Since the value of `a` is `0000 0000 0000 1101` the value of `c` after the execution of the above statement is `0000 0000 0110 1000` (104 in decimal), and is illustrated below:

Left-Shift <<



The three leftmost bits drop off due to the left shift (i.e., they are not present in the result). Three zeros are inserted in the right. The effect of shifting a variable to the left by one bit position is equivalent to multiplying the value by 2. If the initial value of `a` is 13, shifting left by 3 bit positions yields `13*8=104`.

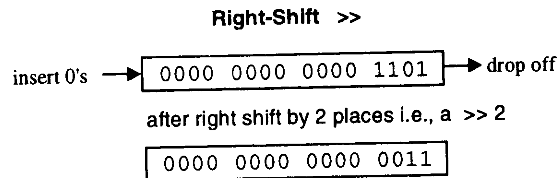
While multiplying a number with a power of 2, considerable savings in execution time can be achieved by using the left bit-shift operator instead of the multiplication operator, since a shift is faster than multiplication.

Right Shift Operator

Consider the statement

```
c = a >> 2;
```

The value of a is shifted right by 2 positions. Since the value of a is 0000 0000 0000 1101 the value of c after the execution of the above statement is 0000 0000 0000 0011 (3 in decimal) and is illustrated below:



The 2 rightmost bits drop off (are not present in the result), and zeros are inserted in the left. The effect of shifting a variable to the right by one bit position is equivalent to dividing the value by 2 (i.e., divide by 2 and truncate the result). As the initial value of a is 13, shifting it right by 2 bit positions yields the value 3 (the result of dividing 13 by 4 and truncating the result). Note that if the negative number is shifted right, then 1 is inserted at the left for every bit shifted to the right.

The program `extract.cpp` illustrates the binary operators. It reads an integer and prints the value of a specified bit in the integer. The position of bits are numbered starting with 0 from right to left. For example, to find the value of the second bit of an integer `i`, it is necessary to shift `i` to the right by two bits, and take the least significant digit.

```
// extract.cpp: Fishing the nth bit
#include <iostream.h>
void main()
{
   // a is the input integer and n, the bit position to extract
   int a, n, bit;
   cout << "Enter an integer: ";
   cin >> a;
   cout << "Enter bit position to extract: ";
   cin >> n;
   bit = (a >> n) & 1;  // bit is the value of the bit extracted (0 or 1)
   cout << "The bit is " << bit;
}
```

Run

```
Enter an integer: 10
Enter bit position to extract: 2
The bit is 1
```

In `main()`, the statement

```
bit = (a >> n) & 1;
```

first shifts a to the right by n bits and then masks (clears) all the bits of a except the least significant bit (rightmost bit), retaining the bit which is required. Parentheses are not required in the above statement, since the operator >> has more precedence than & (i.e., in an expression, if both >> and & are present, the >> operator is executed first). Since this fact is not obvious, it is always better to use parentheses in such situations, for the sake of readability.

4.15 Compound Assignment Operators

As discussed earlier, the assignment operator = (equal sign) evaluates the expression on the right and assigns the resulting value to the variable on the left. Other forms of assignment operators exist, which are obtained by combining operators such as +, -, *, etc., with the = sign as follows:

variable operator= expression/constant/function;

For example, expressions such as

```
i = i + 10;
```

in which the variable `i` on the left hand side is repeated immediately after = sign, and can be rewritten in the compact form as follows:

```
i += 10;
```

The operator += is known as compound assignment operator. Various possible compound assignment operators are shown in Table 4.10. These operators evaluate the expression on their right, and use the result to perform the corresponding operation on the variable on the left. Note that, only the binary operators can be combined with the assignment operator.

Operator	Usage	Effect
`+=`	`a += exp;`	`a = a + (exp);`
`-=`	`a -= exp;`	`a = a - (exp);`
`*=`	`a *= exp;`	`a = a * (exp);`
`/=`	`a /= exp;`	`a = a / (exp);`
`%=`	`a %= exp;`	`a = a % (exp);`
`&=`	`a &= exp;`	`a = a & (exp);`
`|=`	`a |= exp;`	`a = a | (exp);`
`^=`	`a ^= exp;`	`a = a ^ (exp);`
`<<=`	`a <<= exp;`	`a = a << (exp);`
`>>=`	`a >>= exp;`	`a = a >> (exp);`

Table 4.10: Compound assignment operators

The statement

variable operator= expression;

is equivalent to

variable = variable operator (expression);

Hence, a statement such as

```
x *= y + 2;
```

is equivalent to

```
x = x * (y + 2);
```

rather than

```
x = x * y + 2;
```

4.16 Increment and Decrement Operators

The C++ language offers two unusual unary operators for incrementing and decrementing variables. These are ++ and -- operators and are known as increment and decrement operators respectively. These operators increase or decrease the value of a variable on which they operate by one. The speciality about them is that they can be used as prefix or postfix and their meaning changes accordingly. When used as a prefix, the value of the variable is incremented/decremented before being used in the expression. But when used as a postfix, it's value is first used in the expression and then the value is incremented/decremented. The syntax of the operators is given below:

++VariableName
VariableName++
--VariableName
VariableName--

The operator ++ adds 1 to the operand and -- subtracts 1 from the operand. The prefix and postfix for increment expressions are shown below:

`++m` and `m++`

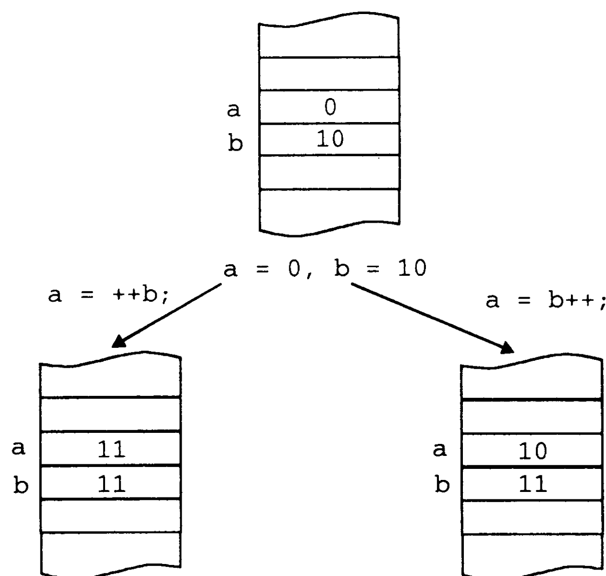


Figure 4.7: Prefix and postfix increment

Consider the following statements

```
++m;
m++;
```

In the above statements, it does not matter whether the increment operator is prefixed or suffixed, it will produce the same result. However, in the following examples, it does make a difference:

```
int a = 0, b = 10;
```

The statement

```
a = ++b;
```

is different from

```
a = b++;
```

In the first case, the value of a after the execution of this statement will be 11, since b is incremented first and then assigned. In the second case, the value of a will be 10, since it is assigned first and then incremented (see Figure 4.7). The value of b in both the cases will be 11. These unary operators have a higher precedence than the binary arithmetic operators. The increment and decrement operators can only be applied to variables; an expression such as `(i+j)++` is illegal.

4.17 Conditional Operator (Ternary Operator)

An alternate method to using a simple if-else construct is the conditional expression operator `?:`. It is called the ternary operator, which operates on three operands. It has the following syntax:

expression1 ? expression2 : expression3

Here the *expression1* is evaluated first; if it is true, then the value of *expression2* is the result; otherwise, the *expression3* is the result. The `if-else` construct

```
if(a > b)
   z = a;
else
   z = b;
```

which finds the maximum of a and b; it can be alternatively realized by using

```
z = (a > b) ? a : b;
```

It is illustrated in Figure 4.8.

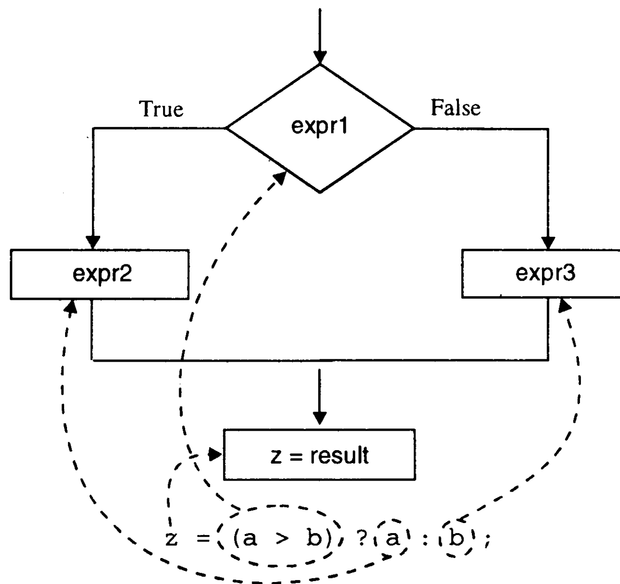


Figure 4.8: Ternary operation evaluation

The program max.cpp reads two integers and displays the value of the larger of the two numbers computed using the ternary operator. If they are equal, then naturally, either of them can be printed.

```
// max.cpp: finding the maximum using the conditional operator
#include <iostream.h>
void main()
{
   int a, b, larger;
   cout << "Enter two integers: ";
   cin >> a >> b;
   larger =  a > b ? a : b;
   cout << "The larger of the two is " << larger;
}
```

Run

```
Enter two integers: 10 20
The larger of the two is 20
```

In main(), the statement

```
larger =  a > b ? a : b;
```

has three components. The conditional expression (a > b ? a : b) returns a if a > b, otherwise it returns b. It can be equivalently coded using the if statement as follows:

```
if( a > b )
  larger = a;
else
  larger = b;
```

The expressions in the ternary operator can be any valid variable, constant, or an expression. The program oddeven.cpp checks whether the number is odd or even using the ternary operator.

```
// oddeven.cpp: checks whether the number is odd or even
#include <iostream.h>
void main()
{
   int num;
   char *str;
   cout << "Enter the number: ";
   cin >> num;
   cout << "The number " << num << " is ";
   cout << ((num % 2) ? "Odd" : "Even");
   cout << endl << "Enter the number: ";
   cin >> num;
   cout << "The number " << num << " is ";
   (num % 2) ? cout << "Odd" : cout << "Even";
}
```

Run

```
Enter the number: 10
The number 10 is Even
Enter the number: 25
The number 25 is Odd
```

In `main()`, the statements

```
cout << ((num % 2) ? "Odd" : "Even");
(num % 2) ? cout << "Odd" : cout << "Even";
```

produce the same result. In the first statement, when the input value is 10, it returns the string `Even`, which is passed to `cout` for display. The second statement executes

```
cout << "Even"
```

when the input is a even number, otherwise, it executes the first expression

```
cout << "Odd"
```

4.18 Special Operators

Some of the special operators supported by C++ include `sizeof`, indirection, comma, etc. The `sizeof()` operator returns the size of the data type or the variable in terms of bytes occupied in memory, as illustrated earlier. Another class of operators is the member selection operators (. and ->) which are used with structures and unions. The indirection and address operators * and & respectively are explained in detail in the later chapters.

Comma Operator

A set of expressions separated by commas is a valid construct in the C++ language. It links the related expressions together. Expressions linked using *comma operator* are evaluated from left to right and the value of the rightmost expression is the result. For example, consider the following statement that makes use of the comma operator.

```
i = (j = 3, j + 2);
```

The right hand side consists of two expressions separated by commas. The first expression is `j=3` and the second one is `j+2`. These expressions are evaluated from left to right. i.e., first the value 3 is assigned to `j` and then the expression `j+2` is evaluated, giving 5. *The value of the entire comma-separated expression is the value of the right-most expression.* In the above example, the value assigned to `i` would be 5.

Some other typical situations where the comma operator can be used are the following:

```
1. for( int i = 2, j = 10; ..; .. )
2. t = x, x = y, y = t;      // exchanges x and y values
```

4.19 `typedef` Statement

The `typedef` statement is used to give new names to existing data types. It allows the user to declare an identifier to represent an existing data type (with enhancement) as shown in the following syntax:

typedef type identifier;

where *type* refers to an existing data type and `identifier` refers to the new name given to the data type. For example, the statement,

typedef unsigned long ulong;

declares `ulong` to be a new type, equivalent to `unsigned long`. It can be used just like any standard data type in the program. For example, the statement

```
ulong u;
```

defines `u` to be of type `ulong`. Also `sizeof(ulong)` returns the size of the new variable type in bytes.

4.20 Promotion and Type Conversion

A mixed mode expression is one in which the operands are not of the same type. In this case, the operands are converted before evaluation, to maintain compatibility between data types. It can be carried out by the compiler automatically or by the programmer explicitly.

Implicit Type Conversion

The compiler performs type conversion of data items when an expression consists of data items of different types. This is called implicit or automatic type conversion. The rules followed by the compiler for implicit type conversion is shown in Table 4.11.

Operand1	Operand2	Result
char	int	int
int	long	long
int	float	float
int	double	double
int	unsigned	unsigned
long	double	double
double	float	double

Table 4.11: Automatic type conversion rule table

Consider the following statements to illustrate automatic type conversion

```
float f = 10.0;
int i = 0;
i = f / 3;
```

In this expression, the constant 3 will be converted to a `float` and then the floating point division will take place, resulting in 3.33333. This (integer to float) type of conversion, where the variable of a lower data type (which can hold lower range of values or has lower precision) is converted to a higher type (which can hold higher range of values or has higher precision) is called *promotion*. But the `lvalue` is an integer variable, hence, the result of `f/3` will be automatically truncated to 3 and the fractional part will be lost. This (float to integer) type of conversion, where the variable of higher type is converted to a lower type is called *demotion*.

The implicit conversions thus occurring are also called *silent conversions* since the programmer is not aware of these conversions. The flexibility of the C++ language, to allow mixed type conversions implicitly, saves a lot of effort on the part of the programmer, but at times, it can give rise to bugs in the program.

The following statement illustrates the process of type conversion:

```
int a, c;
long l;
```

```
float f;
double d;
l = l / a + f * d - d;
```

The variables `a` and `f` are type converted to `long` and `double` respectively. The process of type conversion leading to data *promotion* or *demotion* while assigning the computed result (if necessary), is shown in Figure 4.9.

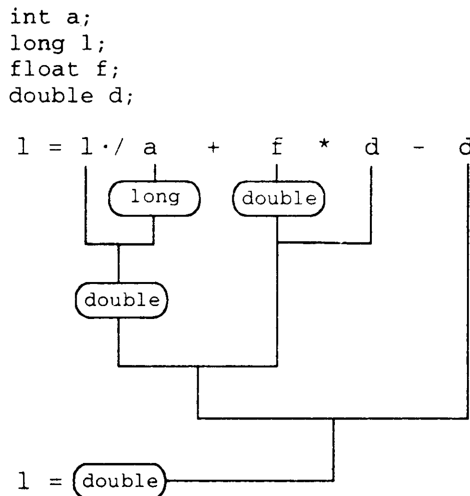


Figure 4.9: Automatic type conversion

Explicit Type Conversion

Implicit type conversions, as allowed by the C++ language, can lead to errors creeping into the program, if adequate care is not taken. Therefore, the use of explicit type conversion is recommended in mixed mode expressions. It is achieved by typecasting a value of a particular type, into the desired type as follows:

(*type*) *expression*
(*type*) *variable_name*

The expression/variable is converted to the given type. Consider the expression:

```
(float)i+f
```

It *type casts* the variable `i` of type integer to `float`. Another syntax for type conversion, which is specific to C++ is as follows:

type(*expression*)
type(*variable_name*)

Typecasting can also be used to convert from a higher type to a lower type. For example, if `f` is a float whose value is 2.7, the expression

```
int(f)
```

evaluates to 2. The program `coerce.cpp` illustrates the different ways of achieving type conversion.

```
// coerce.cpp: type conversion
#include <iostream.h>
void main()
```

```
{
   int i, j;
   float f;
   i = 12;
   j = 5;
   cout << "when i = " << i << " j = " << j << endl;
   f = i/j;
   cout << "i/j = " << f << endl;
   f = (float)i/j;
   cout << "(float)i/j = " << f << endl;
   f = float(i)/j;
   cout << "float(i)/j = " << f << endl;
   f = i/float(j);
   cout << "i/float(j) = " << f << endl;
}
```

Run

```
when i = 12 j = 5
i/j = 2
(float)i/j = 2.4
float(i)/j = 2.4
i/float(j) = 2.4
```

4.21 Constants

A constant does not change its value during the entire execution of the program. They can be classified as integer, floating point, character, and enumeration constants.

(i) Integer Constants

C++ allows to represent the integer constants in three forms. They are octal, decimal, and hexadecimal.

Octal System (Base 8): Octal numbers are specified with a leading zero, rest of the digits being between 0 and 7. For instance, 0175 is an integer constant specified in octal whose base-10 (decimal) value is 125.

Decimal System (Base 10): It is the most commonly used system. A number in this system is represented by using digits 0-10. For instance, 175 is an integer constant with base 10.

Hexadecimal System (Base 16): Hexadecimal numbers are specified with `0x` or `0x` in the beginning. The digits that follow `0x` must be numbers in the range 0-9 or one of the letters `a-f` or `A-F`. For example, `0xa1` is an integer constant specified in hexadecimal whose base-10 or decimal value is 161. `0Xa1` is the same as `0xa1`, or `0xA1`. i.e., either a lower case or an upper case `x` can be used.

A size or sign qualifier can be appended at the end of the constant. The suffix `u` is used for `unsigned int` constants, `l` for `long int` constants and `s` for `signed int` constants. It can be represented either in upper case or lower case.

Examples:

1. Unsigned integer constants

```
56789U
56789u
```

2. Long integer constants

```
7689909L
76899091
```

`0675434L` (A long integer constant specified in octal).
`0x34ADL` (A long integer constant specified in hexadecimal).
`0xf4A3L` (A long integer constant in hexadecimal with upper and lower case letters).

3. The suffixes can be combined, as illustrated in the following unsigned long integer constants. The suffixes can be specified in any order.

```
6578890994Ul
6578890994ul
```

(ii) Floating Point Constants

Floating point constants have a decimal point, or an exponent sign, or both.

Decimal notation: Here the number is represented as a whole number, followed by a decimal point and a fractional part. It is possible to omit digits before and after the decimal point.

Examples of valid floating point constants:

```
125.45  241.   .976  -.71   +.5
```

Exponential notation: Exponential notation is useful in representing numbers whose magnitudes are very large or very small. The exponential notation consist of a mantissa and an exponent. The exponent is positive unless preceded by a minus sign. The number 231.78 can also be written as 0.23178e3, representing the number $0.23178*10^3$. The sequence of digits 23178 in this case after the decimal point is called the mantissa, and 3 is called the exponent.

For example, the number 75000000000 can be written as 75e9 or 0.75e11. Similarly , the number 0.00000000045 can be written as 0.45e-9.

(i) The following examples are valid constants

```
2000.0434
3.4e4
3E8
```

(ii) The following are some invalid constants.

2,000.0434	- comma not allowed.
3.4E.4	- exponent must be an integer.
3e 8	- blank not allowed.

Normalized exponential representation is one in which the value of the mantissa is adjusted to a value between 0.1 and 0.99, for example, the number 75000000000 is written as 0.75e11

The rules governing exponential representation of the real constants are given below:

- The mantissa is either a real number expressed in decimal notation or an integer.
- The mantissa can be preceded by a sign.
- The exponent is an integer preceded by an optional sign.
- The letter e can be written in lowercase or uppercase.
- Embedded white space is not allowed.

By default, real constants are assumed to be double. Suffixes f or F can be used to specify the float values. For example, 0.257 is assumed to be a double constant, while 0.257f is a float constant.

The character l or L can be used to specify long double values. For example, 0.257L is a long double constant.

(iii) Character Constants

A character constant is enclosed in single quotes.

Examples: Valid character constants: 'a' '5' '\n'.
Invalid character constants: 'ab' '54'.

Note that multiple characters can also exist within single quotes. The compiler will not report any error. The value of the constant however, depends upon the compiler used. This notation of having multiple characters in single quotes is practically never used.

Inside the single quotes, a backslash character starts an escape sequence. \xhh specifies a character constant in hexadecimal, where h is any hexadecimal digit. The hexadecimal digits hh gives the ASCII value of the character. For example, the character constant \x07 represents the BELL character. The complete list of escape sequences is shown in Table 4.12.

Operator	Meaning
\a	Beep
\b	Backspace
\f	Formfeed
\n	Newline
\t	Horizontal tab
\\	Backslash
\'	Single quote
\"	Double quote
\v	Vertical tab
\?	Question mark
\0	Null
\0ooo	Code specified in octal
\xhh	Code specified in hexadecimal

Table 4.12: Escape sequences

The escape sequence \OXXX specifies a character constant in octal, where 0 denotes any octal digit. As before, XXX is the ASCII value of the number specified in octal. For example, the ASCII code of K is 75. The character constant \0113 specifies this character in octal. The program beep.cpp generates the beep sound using the escape sequence. The escape sequence \x07 (can be \7) in the cout can be replaced by \a.

```
// beep.cpp: generating beep sound
#include <iostream.h>
void main()
{
   cout << '\x07';   // computer generates sound
}
```

Run

Note: You will hear a beep sound.

Examples:

(i) `cout << "\\ is a backslash.";` will print as follows:

```
\ is a backslash.
```

(ii) `cout << "This \" is a double quote.";` will print

```
This " is a double quote.
```

(iv) String Literals

A string literal is a sequence of characters enclosed in double quotes.The characters may be letters, numbers, escape sequences, or blank space. To make it easier, string constants are concatenated at compile time. For example, the strings:

`"C++ is the best"` and,

`"C++ is "  "the best"` are the same.

An important difference: 'A' and "A"

The notations 'A' and "A" have an important difference. The first one ('A') is a character constant, while the second ("A") is a string constant. The notation 'A' is a constant occupying a single byte containing the ASCII code of the character A. The notation "A" on the other hand, is a constant that occupies two bytes, one for the ASCII code of A and the other for the null character with value 0, that terminates all strings. The statement

```
char ch = 'R';
```

assigns ASCII code of the character R to the variable ch, whereas the statement

```
char *str = "Hello OOPs!";
```

assigns the starting address of the string `Hello OOPs!` to the variable `str`.

4.22 Declaring Symbolic Constants–Literals

Literals are constants to which symbolic names are associated for the purpose of readability and ease of handling. C++ provides the following three ways of defining constants:

- # define preprocessor directive
- enumerated data types
- const keyword

The keyword `const` is already discussed in the chapter 2. The following section discusses macros and enumerated data types.

#define Preprocessor Directive

The preprocessor directive `#define`, associates a constant value to a symbol and is visible throughout the module in which it is defined. The symbols defined using `#define` are called *macros*. The syntax of `#define` directive is

```
#define SymbolName ConstantValue
```

Examples:

```
# define MAX_VAR 100
```

```
# define PI 3.1452
# define NAME "Rajkumar"
```

The preprocessor will replace all the macro symbols used in the program by their values before starting the compilation operation. For instance, the statement,

```
area = PI * radius * radius;
```

is translated as,

```
area = 3.1452 * radius * radius;
```

by the processor if there exist a preprocessor directive,

```
#define PI 3.1452
```

in the program and before the statement referencing to it. The definition of macros can be superseded by a new definition. For instance, the symbol `PI` can be redeclared as,

```
#define PI (22/7)
```

The program `city.cpp` illustrates the superseding of the value of old macro symbol by a new declaration.

```
// city.cpp: superseding of macros
#include <iostream.h>
#define CITY "Bidar"
void which_city();
void main()
{
   cout << "Earlier City: ";
   cout << CITY << endl;
#define CITY "Bangalore"
   cout << "New City: ";
   cout << CITY << endl;
   which_city();
}
void which_city()
{
   cout << "City in Function: ";
   cout << CITY;
}
```

Run

```
Earlier City: Bidar
New City: Bangalore
City in Function: Bangalore
```

In the above program, initially the macro constant `CITY` is declared with the value `"Bidar"`. The statement in the beginning of the `main()` function

```
cout << CITY << endl;
```

will print the message

```
Bidar
```

as seen in the output of the program. However, the same statement at the end of `main()` and in the function `which_city()` prints the message

```
Bangalore
```

Thus, the most recent declaration of the macro constant will supersede the earlier one. Macro constants

behave similar to global variables except that they are visible from the point of their declaration.

The important advantages of using macro symbols include the following:

- Program coding is easier
- Enhances program readability
- Program maintenance is easier

The disadvantage of macro constants is that, they do not support the specification of the data-type in the declaration; any type of value can be assigned (either integer, float, or string).

4.23 Enumerated Data Types

An enumerated data type is a user defined type, with values ranging over a finite set of identifiers called enumeration constants. For example,

```
enum color {red, blue, green};
```

This defines `color` to be of a new data type which can assume the value, `red`, `blue`,or `green`. Each of these is an enumeration constant. In the program, `color` can be used as a new type. A variable of type `color` can have any one of the three values: red, blue or green. For example, the statement

```
color c;
```

defines `c` to be of type `color`. Internally, the C++ compiler treats an enum type (such as color) as an integer itself. The above identifiers `red`, `blue`, and `green` represent the integer values of 0, 1, and 2 respectively. So, the statements

```
c = blue;
cout << "As an int, c has the value " << c;
```

will print

```
As an int, c has the value 1
```

Constant values can be explicitly specified for the identifiers. When the value for one identifier is specified in this manner, the value of the next element is incremented by one (next higher integer). For example, if the definition of `color` is

```
enum color {red = 10, blue, green = 34};
```

then the statement `c = red` will assign the value `10` to `c`. Thereafter, the statement

```
c = blue;
```

assigns the value 11 to `c`, and the statement

```
c = green;
```

assigns the value `34` to `c`. (If no value is specified for `green` in the declaration, it would assume the value 12).

Enumeration is a convenient way to associate constant integers with meaningful names. They have the advantage of generating the values automatically. Use of enumeration constants, in general makes the program easier to read and change at a later date.

Names of different enumeration constants must be distinct. The following example is invalid.

```
enum emotion  {happy, hot, cool};
enum weather  {hot, cold, wet};
```

It is not difficult to see why the above declarations are invalid; the name `hot` has the value 1 in the `enum emotion` and the value 0 in `weather`. In the program, if the name `hot` is used, there is

ambiguity as to which value to use. On the other hand, values need not be distinct in the same enumeration. For example, the following declaration is perfectly valid:

```
enum weather {hot, warm = 0, cold, wet);
```

The names hot and warm can be interchangeably used, since both represent the value 0.

Consider the following enumeration statement

```
enum flag { false, true };
```

It declares the identifier flag as an enumerated data type. It can be further used in the definition of enumerated variables as follows:

```
flag flag1;   // holds either false or true
```

In this case, the variable flag1 is defined as an enumerated variable of type flag and always holds the value either true or false as follows:

```
flag1 = true;
```

If an attempt is made to assign any value other than true or false, the compiler generates a warning.

```
flag1 = 3;    // warning: trying to assign integer to flag1
```

Use only enumerated constants with enumerated variables. The multimodule programs color1.cpp and color2.cpp illustrate some critical points on enumerated data types.

```
// color1.cpp: main having enum typedef and calling function from color2.cpp
#include <iostream.h>
typedef enum Color { red, green, blue }; // red = 0, green = 1, and blue = 2
void PrintColor( Color c );
void main()
{
   cout << "Your color choice in color1.cpp module: green" << endl;
   PrintColor( green );    // calls module in color2.cpp
}
```

```
// color2.cpp: prints color name based on color code
#include <iostream.h>
typedef enum Color { red, blue, green }; // red = 0, blue = 1, and green = 2
void PrintColor( Color c )
{
   char *color;
   switch( c )
   {
      case red:   // case 0
         color = "red";
         break;
      case blue:  // case 1
         color = "blue";
         break;
      case green: // case 2
         color = "green";
         break;
   }
   cout << "Your color choice as per color2.cpp module: " << color;
}
```

Run

```
Your color choice in color1.cpp module: green
Your color choice as per color2.cpp module: blue
```

The modules `color1.cpp` and `color2.cpp` must be compiled and linked together in order to create an executable code. The command to create an executable version of these modules, in the Borland C++ environment is:

```
bcc color1.cpp color2.cpp
```

It creates the executable file `color1.exe`.

The enumeration declaration statement in `color1.cpp`

```
typedef enum Color { red, green, blue };
```

creates three constant symbols red, green, blue with 0, 1, and 2 respectively. It can be written without the use of `typedef` keyword as follows:

```
enum Color { red, green, blue };
```

An enumerated variable can be defined using the statement

```
Color c1;
```

although, the `typedef` keyword is missing. The enumeration declaration statement in `color2.cpp`

```
typedef enum Color { red, blue, green };
```

creates three constant symbols red, green, blue with 0, 1, and 2 respectively. Note that, the enumerated symbol green has the value 1 in the first module `color1.cpp` whereas, it has the value 2 in the module `color2.cpp`. The statement in `color1.cpp`

```
PrintColor( green );     // calls module in color2.cpp
```

invokes the `PrintColor()` defined in the `color2.cpp` module with the enumerated symbol green (whose value is 1 in `color1.cpp`) to print the message `green`. Instead it prints the message `blue`; the enumeration declaration in `color2.cpp` declares the symbol green having the value 2 and blue as 1. The value of symbol green in `color1.cpp` is the same as that of the symbol blue in `color2.cpp`. This can be observed from the switch statement with the enumerated variable `c` in the `color2.cpp` module. Such inconsistent enumeration declaration must be avoided, and they must have the same declaration in all the modules constituting a program. Thus, enumeration variables can be defined in any module, but it is defined according to the enumeration declaration in its own module. Enumerated constants will have the same value as declared in the current module. In the above program, the module `color1.cpp` has enumeration declaration:

```
typedef enum Color { red, green, blue };
```

and the module `color2.cpp` has the enumeration declaration:

```
typedef enum Color { red, blue, green };
```

Note that, in the above declarations, enumeration constants `green` and `blue` will have different value in different modules. Such mismatch in declaration will generate wrong results. Therefore, the call

```
PrintColor( green );
```

in the module `color1.cpp` prints `blue` instead of `green`.

4.24 Macro Functions

The preprocessor will replace all the macro functions used in the program by their function body before the compilation. The distinguishing feature of macro functions are that there will be no explicit function

call during execution, since the function body is substituted at the point of macro call during compilation. Thereby the runtime overhead for function linking or context-switch time is reduced. The directive `#define`, indicates the start of a macro function as shown in Figure 4.10. The macro function specification spans for a maximum of one line only. However, macro function body can spread to multiple lines if each new.line is followed by '\' character.

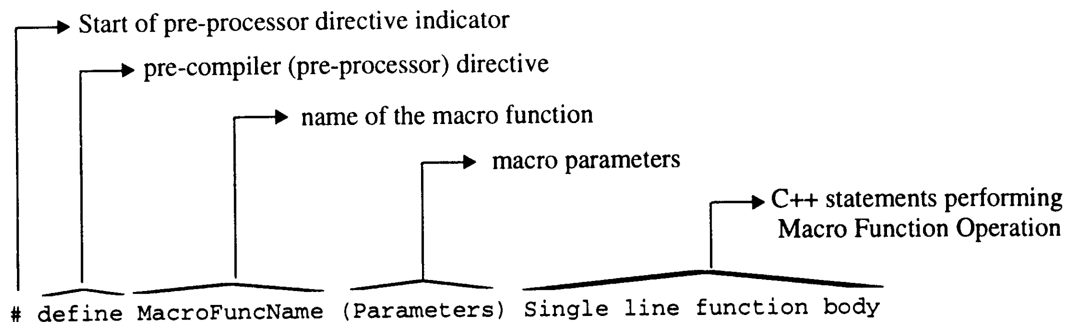


Figure 4.10: Syntax for declaring macro function

Examples:

```
#define inc( a ) a+1
#define add( a, b ) (a+b)
```

The program `maxmacro.cpp` illustrates the use of macro function in the computation of the maximum of two numbers.

```
// maxmacro.cpp: maximum of two numbers using macros
#include <iostream.h>
#define max( a, b ) ( a > b ? a : b )
void main()
{
   cout << "max( 2, 3 ) = " << max( 2, 3 ) << endl;
   cout << "max( 10.2, 4.5 ) = " << max( 10.2, 4.5 ) << endl;
   int i = 5, j = 10;
   cout << "i = " << i << endl;
   cout << "j = " << j << endl;
   cout << "On execution of k = max( ++i, ++j );..." << endl;
   int k = max( ++i, ++j );
   cout << "i = " << i << endl;
   cout << "j = " << j << endl;
   cout << "k = " << k << endl;
}
```

Run

```
max( 2, 3 ) = 3
max( 10.2, 4.5 ) = 10.2
i = 5
j = 10
On execution of k = max( ++i, ++j );...
i = 6
j = 12
k = 12
```

In `main()`, the expressions

```
max( 2, 3 )
max( 10.2, 4.5 )
```

invoke the macro function `max()`. Unlike normal functions, macro functions can take parameters of any data type. If arguments are in the form of expressions, they are not evaluated at the point of call, but at the time of their usage. Thus, the statement

```
int k = max( ++i, ++j );
```

is processed by the preprocessor as follows:

```
int k = ( ++i > ++j ? ++i : ++j );
```

It can be observed that, the variable with greater value, will be incremented twice. The macro function body can spread across multiple lines as follows:

```
#define print( n ) \
   for( int i = 0; i < n; i++ ) \
      cout << i;
```

4.25 Operator Precedence and Associativity

Every operator in C++ has a precedence associated with it. Precedence rules help in removing the ambiguity about the order of performing operations while evaluating an expression. The associativity of the operators is also important. Associativity specifies the direction in which the expression is evaluated, while using a particular operator. The precedence and associativity of all the operators including those introduced by C++ are shown in Table 4.13. It is important to note the order of precedence and evaluation (associativity) of operators. Consider the following two statements:

```
int a = 10, b = 15, c = 3, d;
d = a + b * c;
```

In the second statement, first `b` is multiplied by `c`, and then the result is added to `a` and the sum is assigned to `d`. Multiplication is done before addition, since it has higher precedence than the addition operator. In order to override the precedence, braces can be used. For example, the statement

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

would add `a` to `b` first, multiply the result by `c` and assign the product to `d`. Associativity of an operator can be from left-to-right or right-to-left. For example, in the expression

```
d = a - b - c;
```

the leftmost minus is evaluated first and then the second minus is evaluated, causing `c` to be subtracted from the result. Thus, in case where several operators of the same type appear in an expression without braces, the operators are normally evaluated starting from the leftmost operator, proceeding rightward, hence the minus operator associates from left to right. On the other hand, the assignment operator associates from right to left. For example, in the statement

```
d = a = c;
```

the second (right-most) assignment operator is evaluated first. The variable `c` is assigned to `a` and then this value is assigned to `d`.

Like most programming languages, C++ does not specify the order in which the operands of an operator are evaluated. Such operators are `&&`, `||`, `?:`, and ',') For example, in the statement such as

```
x = g() + h();
```

`g()` may be evaluated before `h()` or vice versa; thus if `g()` or `h()` alters a variable (global) on which the other depends, then the resultant value of `x` is dependent on the order of evaluation. Similarly, the order in which function arguments are evaluated is not specified, so the statement

Category	Operator	Operation	Precedence	Associativity
Highest precedence	() [] -> :: .	Function call Array subscript C++ indirect component selector C++ scope access/resolution C++ direct component selector	1	L→R (left to right)
Unary	! ~ + - ++ -- & * sizeof new delete	Logical negation (NOT) Bitwise (1's) component Unary plus Unary minus Preincrement or postincrement Predecrement or postdecrement Address Indirection (returns size of operand, in bytes) dynamically allocates C++ storage dynamically deallocates C++ storage	2	R→L (right to left)
Member access	.* ->*	dereference dereference	3	L→R
Multiplication	* / %	Multiply Divide Remainder (modulus)	4	L→R
Additive	+ -	Binary plus Binary minus	5	L→R
Shift	<< >>	Shift left Shift right	6	L→R
Relational	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	7	L→R
Equality	== !=	Equal to Not equal to	8	L→R
Bitwise AND	&	Bitwise AND	9	L→R
Bitwise XOR	^	Bitwise XOR	10	L→R
Bitwise OR	|	Bitwise OR	11	L→R
Logical AND	&&	Logical AND	12	L→R
Logical OR	||	Logical OR	13	L→R
Conditional	?:	(a?x:y means "if a then x, else y")	14	L→R
Assignment	= *= /= %= += -= &= ^= |= <<= >>=	Simple assignment Assign product Assign quotient Assign remainder (modulus) Assign sum Assign difference Assign bitwise AND Assign bitwise XOR Assign bitwise OR Assign left shift Assign right shift	15	R→L
Comma	,	Evaluate	16	L→R

Table 4.13: Operator precedence and associativity

```
add( ++n, pow( 2, n ) );
```

can produce different results with different compilers. However, *most C++ compilers evaluate function arguments from right to left*. There are cases such as in function calls, nested assignments, increment and decrement operators cause *side effects*—some variable is changed as a by-product of the evaluation of the expression. The (C language and hence, C++) standard intentionally leaves such matters unspecified. When side effects (variable modification) take place within an expression, it is left to the discretion of the compiler, since the best order depends strongly on the machine architecture.

The moral is that developing a code that depends on the order of evaluation, is not a good programming practice in any language. Hence, it is necessary to know what to avoid, but if it is not known how they are treated, the programmer should not be tempted to take advantage of a particular implementation.

Review Questions

4.1 What are variables ? List C++ rules for variable naming.

4.2 Why `output` and `Output` are considered as different identifiers ?

4.3 What are keywords ? List keywords specific to C++. Can these keywords be used as variables ?

4.4 What is a data type ? What are the different data types supported by C++ ?

4.6 What is new about C++ in terms of the variable definition ?

4.7 What is the difference between a character and a character string representation ?

4.8 What is an expression ? Is this different from a statement ? Give reasons.

4.9 List categories of operators supported by C++.

4.10 What are qualifiers ? Illustrate them with examples.

4.11 Develop an interactive program to compute simple and compound interest.

4.12 List evaluation steps for the expression `(a+(b*c))*c+d/2`.

4.13 Write an interactive program to find the largest of two numbers.

4.14 How C++ represents true and false values ? Are the expressions `!a` and `a==0` have the same meaning ? Give reasons.

4.15 Write a program to determine the type of compiler (whether 16 or 32-bit) used to compile it.

4.16 Write a program to multiply and divide a given number by 2 without using * and / operators.

4.17 Illustrate how compound assignment operators allow to write compact expressions ?.

4.18 What is the effect of the following expressions if `i=1` and `j=4` ?

a) `i++`   b) `j = j++;`   c) `j = ++j;`   d) `i+++j`   e) `i = i+++++j;`

4.19 Write an interactive program to find elder among *you* and *me* using the ternary operator.

4.20 What is the outcome of the statement: `a=(a=10,a++,a--);` if a holds the value 5 initially.

4.21 What is type conversion ? What are the differences between silent and explicit type conversion? Write type conversion steps required for evaluating the statement:

`z=i+b+j-k/4;` (where i and j are ints, b is float, and k is double, and z is long type).

4.22 What are escape sequences ? Write a program to output messages in double quotes.

4.23 What are macros ? Write a program to find the minimum of two numbers using macros. What is the output of the statement: `a = min( ++a, ++b);` (if a = 2 and b = 4).

4.24 What is operator precedence ? Arrange the following operators in the order of their precedence:

`-`, `*`, `+`, `()`, `^`, `!`, `++`, `--`, `|`, `||`, `&`, `/`, and `&&`

4.25 What is the significance of the associativity of operators ? What is the order of evaluation of the operator `?:` in the statement

```
a = i > j ? i : j;
```

5

Control Flow

5.1 Introduction

In real-world, several activities are initiated (sequenced), or repeated based on some decisions. Such activities can be programmed by specifying the order in which computations are carried out. Flow control is the way a program causes the flow of execution to advance and branch based on changes in the data state. Branching, iteration, dispatch, and function calls are all different forms of *flow control*. Flow control in C++ is nearly identical to those in C. Many C programs can be converted quite easily to C++ because of this similarity. The C++ language offers a number of control flow statements: `for`, `while`, `do-while`, `if-else`, `else-if`, `switch`, `goto`. Although all of them can perform operations such as looping or branching, each one of them is convenient for a particular requirement. The control flow statements can be broadly categorized as, branching and looping statements.

Branching Statements

Branching statements alter sequential execution of program statements. Following are the branching statements supported by C++:

(a) `if` statement
(b) `if-else` statement
(c) `switch` statement
(d) `goto` statement

Among all the above statements, `goto` is the only unconditional branching statement.

Looping Statements

Loops cause a section of code to be executed repeatedly until a termination condition is met. The following are the looping statements supported in C++:

(a) `for` statement
(b) `while` statement
(c) `do-while` statement

The `goto` statement can be used for looping, but its use is generally avoided as it leads to haphazard code and also increases the chances of error.

5.2 Statements and Block

An expression such as `a = 1000`, `x++`, or `cout << "Hi"`, when followed by the semicolon, becomes a statement. For example, the following

```
a = 1000;
x++;
cout << "Hi";
```

are treated as C++ statements. In C++, the semicolon is a statement terminator, rather than a separator as in Pascal.

C++ allows grouping of statements, which have to be treated as an entity and the resulting group is called *compound statement* or *block*. It consists of declarations, definitions, and statements enclosed within braces { and } as follows:

```
{
    int a;
    int b = 10;
    a = b + 100;
    ....
}
```

Note that, there is no semicolon after the right brace that ends a block. A block is syntactically equivalent to a single statement. Any variable defined within a block is local to the block and it is not visible outside the block. Blocks are very useful when branching or looping action is to be applied on a set of statements depending on a particular decision. Examples illustrating the use of a block will be discussed later.

5.3 `if` Statement

The `if` construct is a powerful decision making statement which is used to control the sequence of the execution of statements. It alters the sequential execution using the following syntax:

if(test-expression)
  statement;

The test-expression should always be enclosed in parentheses. If test-expression is true (nonzero), then the statement immediately following it is executed. Otherwise, control passes to the next statement following the `if` construct. The control flow in the `if` statement is shown in Figure 5.1.

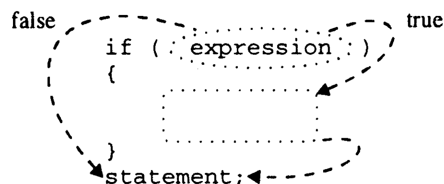


Figure 5.1: Control flow in if statement

Notice that there is no `then` keyword following the test expression, as there is in BASIC and Pascal. The program `age1.cpp` illustrates the use of `if` statement for making a decision.

```
// age1.cpp: use of if statement
#include <iostream.h>
void main()
{
   int age;
   cout << "Enter your age: ";
   cin >> age;
   if( age > 12 && age < 20 )
      cout << "you are a teen-aged person. good!";
}
```

Run1

```
Enter your age: 15
you are a teen-aged person. good!
```

Run2

```
Enter your age: 20
```

In `main()`, the statement

```
if( age > 12 && age < 20 )
```

first evaluates the test expression and executes the if-part only when it is true. In *Run1*, the input data entered is 15 which lies between 13 and 19 and hence, the statement

```
cout << "you are a teen-aged person. good!";
```

gets executed. Whereas, in *Run2*, the input data is 20 which does not lie within this range and hence, the control proceeds to the next statement.

Compound Statement with `if`

In the `if` construct, the if-part can be represented by a compound statement as follows:

if(test-expression)
{
  statement 1;
  statement 2;
}

In this case, when test-expression is true, the statements enclosed within the curly braces, representing a compound statement, are executed. The program `age2.cpp` illustrates the use of the compound-if statement.

```
// age2.cpp: use of if statement and data validation
#include <iostream.h>
void main()
{
   int age;
   cout << "Enter your age: ";
   cin >> age;
   if( age < 0 )
   {
      cout << "I am sorry!" << endl;
      cout << "age can never be negative";
      return;    // terminate program
   }
   if( age > 12 && age < 20 )
      cout << "you are a teen-aged person. good!";
}
```

Run

```
Enter your age: -10
I am sorry!
age can never be negative
```

In `main()`, the statement

```
if( age < 0 )
```

validates the input data and accordingly takes action. It terminates the program after issuing the warning message, when the input data is negative.

The program `large.cpp` illustrates the use of multiple decision statements to compute the maximum of three numbers.

```
// large.cpp: find the largest of three numbers
#include <iostream.h>
void main()
{
   float  a, b, c, big;
   cout << "Enter three floating-point numbers: ";
   cin >> a >> b >> c;
   // computing the largest of three numbers
   big = a;
   if(b > big)
      big = b;
   if(c > big)
      big = c;
   cout << "Largest of the three numbers = " << big;
}
```

Run

```
Enter three floating-point numbers: 10.2 15.6 12.8
Largest of the three numbers = 15.6
```

5.4 `if-else` Statement

The `if-else` statement will execute a single statement or a group of statements, when the test expression is true. It does nothing when the test expression fails. C++ provides the `if-else` construct to perform some action even when the test expression fails. The control flow in the `if-else` statement is shown in Figure 5.2.

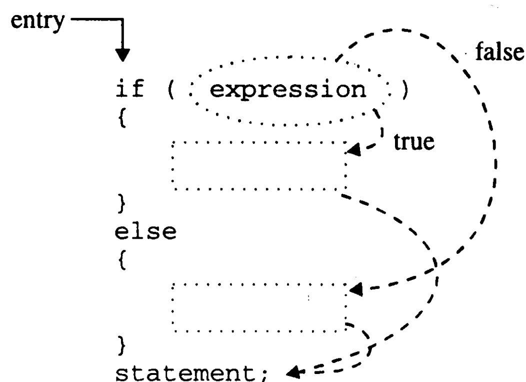


Figure 5.2: Control flow in if-else condition

When test-expression is true (nonzero), the if-part is executed and control passes to the next statement following the `if` construct. Otherwise, the else-part is executed and control passes to the next statement. The program `age3.cpp` illustrates the use of the `if-else` statement.

```
// age3.cpp: use of if..else statement
#include <iostream.h>
void main()
{
   int age;
   cout << "Enter your age: ";
   cin >> age;
   if( age > 12 && age < 20 )
      cout << "you are a teen-aged person. good!";
   else
      cout << "you are not a teen-aged person.";
}
```

Run1

```
Enter your age: 15
you are a teen-aged person. good!
```

Run2

```
Enter your age: 20
you are not a teen-aged person.
```

In `main()`, the statement

```
if( age > 12 && age < 20 )
```

generates different types of output depending on the input values. If the test expression is true, the statement

```
cout << "you are a teen-aged person. good!";
```

is executed. Otherwise, the statement

```
cout << "you are not a teen-aged person.";
```

in the else-part is executed.

Compound Statement with `if-else`

In the `if-else` construct, the if-part, or else-part, or both can have a compound statement as follows:

if(test-expression)
{
  statement 1;
  statement 2;
}
else
{
  statement 3;
  statement 4;
}

The program `lived.cpp` illustrates the use of the compound `if-else` **statements**.

```
// lived.cpp: single if statement validates input data
#include <iostream.h>
void main()
{
   float years, secs;
   cout << "Enter your age in years: ";
   cin >> years;
   if( years < 0 )
      cout << "I am sorry! age can never be negative" << endl;
   else
   {
      secs = years * 365 * 24 * 60 * 60;
      cout << "You have lived for " << secs << " seconds";
   }
}
```

Run1

```
Enter your age in years: -1
I am sorry! age can never be negative
```

Run2

```
Enter your age in years: 25
You have lived for 7.884e+08 seconds
```

5.5 Nested `if-else` Statements

Multi-way decisions arise when there are multiple conditions and different actions to be taken under each condition. A multi-way decision can be written by using `if-else` constructs in the else-part as follows:

```
if( test-expression1 )
  statement1;
else
  if( test-expression2 )
    statement2;
  else
    if( test-expression3 )
      statement3;
```

Here, if test-expression1 is true, the whole chain is terminated. Only if test-expression1 is found false, the chain of events continue. At any stage if an expression is true, the remaining chain will be terminated. The program `age4.cpp` illustrates the use of nested `if-else` statements.

```
// age4.cpp: use of if..else..if statement
#include <iostream.h>
void main()
{
   int age;
   cout << "Enter your age: ";
   cin >> age;
   if( age > 12 && age < 20 )
```

```
      cout << "you are a teen-aged person. good!";
   else
      if( age < 13 )
         cout << "you will surely reach teen-age.";
      else
         cout << "you have crossed teen-age!";
}
```

Run1

```
Enter your age: 16
you are a teen-aged person. good!
```

Run2

```
Enter your age: 25
you have crossed teen-age!
```

In the above program, the nested `if-else` statement takes decisions based on the input data and displays appropriate messages for any given input. It proceeds to match the input data with various conditions when the earlier condition fails to decide the fate of the input data. Note that in case of nested if-else statements, the `else` statement is always associated with the corresponding inner most `if` statement.

Indentation

In all the above examples, the statements inside the `if` construct are indented. The C++ language, however, does not expect indentation of statements. It is done merely for improving program readability. The importance of indenting becomes evident during the usage of nested if statements (if statements within other if statements; any number of nested-if statements are allowed). For example, consider the following if statement

```
if( a > b ) if( .a > c ) big = a;
else big = c;
```

The above statement is perfectly valid as far as the compiler is concerned, but it is very difficult for the programmer to decipher it. An indented version of this is listed below:

```
if( a > b )
   if( a > c )
      big = a;
   else
      big = c;
```

From the above code it can be observed that, indentation enhances the readability of the code and helps in understanding the flow of control with ease.

Nested if-else statements can be conveniently replaced by a new construct called `switch`. It allows to choose among several alternatives; it is.dealt later in this chapter.

5.6 for Loop

The `for` loop is useful while executing a statement a fixed number of times. Even here, more than one statement can be enclosed in curly braces to form a compound statement. The control flow in the for loop is shown in Figure 5.3.

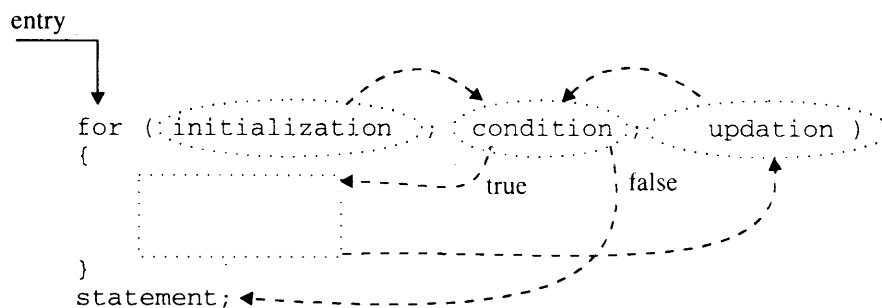


Figure 5.3: Control flow in for loop

The `for` statement is a compact way to express a loop. All the four parts of a loop are in close proximity with the `for` statement. The *initialization part* is executed only once. Next the *test condition* is evaluated. If the test evaluates to false, then the next statement after the for loop is executed. If the *test expression* evaluates to true, then after executing the *body* of the loop, the *update part* is executed. The test is evaluated again and the whole process is repeated as long as the test expression evaluates to true as illustrated in the program `count1.cpp`.

```
// count1.cpp: display numbers 1..N using for loop
#include <iostream.h>
void main()
{
   int n;
   cout << "How many integers to be displayed: ";
   cin >> n;
   for( int i = 0; i < n; i++ )
      cout << i << endl;
}
```

Run

```
How many integers to be displayed: 5
0
1
2
3
4
```

In `main()`, the statement

```
for( int i = 0; i < n; i++ )
    cout << i << endl;
```

has four components. The first three components enclosed in round braces and separated by semicolons are the following:

```
int i = 0
i < n
i++
```

The first component `int i = 0` is called the *initialization expression* and is executed only once prior to the statements within the for loop. The second component `i < n` is called the *test expression* and