namespace

Namespace defines a scope for the identifiers that are used in a program. For using the identifiers defined in the namespace scope we must include the using directive, like

using namespace std;

Here, std is the namespace where ANSI C++ standard class libraries are defined This will bring all the identifiers defined in std to the current global scope.

Output Operator

cout:

The identifier cout (pronounced as 'C out') is a predefined object that represents the standard output stream (the screen) in C++. It is also possible to redirect the output to other output devices.

Eg: cout<<"hello";

The operator << is called the insertion or put to operator. It inserts (or sends) the contents of the variable on its right to the object on its left

Input Operator

cin >> variable;

The identifier cin is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.

The operator >> is known as extraction or get from operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right .

Tokens:

The smallest individual units in a program are known as tokens. C++ has the following tokens:

-Keywords

- Identifiers

- Constants

- Strings

- Operators

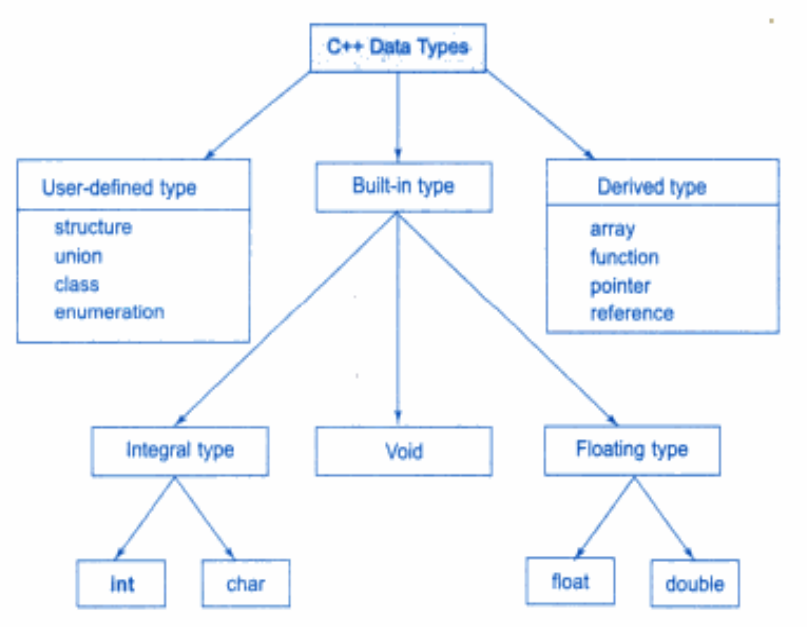
Keywords

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements.

Table 3.1 *C++ keywords*

asm	**double**	new	**switch**
auto	**else**	operator	template
break	**enum**	private	this
case	**extern**	protected	throw
catch	**float**	public	try
char	**for**	**register**	**typedef**
class	friend	**return**	**union**
const	**goto**	**short**	**unsigned**
continue	**if**	**signed**	virtual
default	inline	**sizeof**	**void**
delete	**int**	**static**	**volatile**
do	**long**	**struct**	**while**

C++ Data types:



With the exception of void, the basic data types may have several modifiers preceding them. The modifiers signed, unsigned, long, and short may be applied to character and integer basic data types. However, the modifier long may also be applied to double

Table 3.2 *Size and range of C++ basic data types*

Type	Bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	- 128 to 127
int	2	- 32768 to 32767
unsigned int	2	0 to 65535
signed int	2	- 31768 to 32767
short int	2	- 31768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Uses of the void datatype:

1. Specify a function's return type
2. Empty argument list to a function

```
void funct1(void);
```

3. Declaration of generic pointers

```
void *gp;
```

A **generic pointer** can be assigned a pointer value of any basic data type, but it may not be dereferenced. For example,

int *ip;

gp = ip;

are valid statements. But, the statement,

*ip = *gp;

is illegal. It would not make sense to dereference a pointer to a void value.

Enumerated Data Type

An enumerated data type is a user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The enum keyword automatically enumerates a list of words by assigning them values 0,1,2, and so on. This facility provides an alternative means for creating symbolic constants.

Examples:

enum shape(circle, square, triangle);

enum colour(red, blue, green, yellow};

enum position(off, on);

In C++, the tag names shape, colour, and position become new type names. By using these tag names, we can declare new variables. Examples:

shape ellipse;

colour background;

C++ does not permit an int value to be automatically converted to an enum value. Examples:

colour background = blue;

colour background = 7; //Error in C++

colour background = (colour) 7;//ok

Derived Data Types:

- Arrays

The application of arrays in C++ is similar to that in C. The only exception is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant. For instance,

char string[3] = "xyz";

is valid in ANSI C. It assumes that the programmer intends to leave out the null character \0 in the definition. But in C++, the size should be one larger than the number of characters in the string.

char string[4] "xyz"; // O.K. for C++

- Pointers

As in C, Pointers are variables whose values are memory addresses. Pointers are declared and initialized as in C.

Examples:

int *ip; // int pointer

ip = &x; // address of x assigned to ip

*ip = 10; // 10 assigned to x through indirection

C++ adds the concept of constant pointer and pointer to a constant as follows:

- **Constant pointer :** int * const ptr= &a;

a constant pointer must be initialized at the time of its declaration. The pointer cannot change , but you can modify the value that the pointer points to.

- **Pointer to a constant :** int const * ptr1;

You cannot change the value that ptr points to (**through this pointer**), but you can change the pointer itself to point to another address

Symbolic Constants:

There are two ways of creating symbolic constants in C++:

i. Using the qualifier const, and
ii. Defining a set of integer constants using enum keyword.

any value declared as const cannot be modified by the program in any way. In C++, we can use const in a constant expression, such as

const int size 10;

char name[size];

As with long and short, if we use the const modifier alone, it defaults to int. For example,

const size = 10;

means

const int size = 10;

C++ requires a const to be initialized. The scope of const values is local to the file where it is declared. In ANSI C, const values are global in nature. To reference a const from another file, we must explicitly define it as an extern in C++. Example:

extern const total = 100;

Another method of naming integer constants is by enumeration as under;

enum (X,Y,Z);

This defines X, Y and Z as integer constants with values 0, 1, and 2 respectively.

Variable Declaration:

C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program much easier to write and reduces the errors. It also makes the program easier to understand because the variables are declared in the context of their use.

```
int main()
{
        float x;                    // declaration
        float sum = 0;

        for(int i=1; i<5; i++)      // declaration
        {
                cin >> x;
                sum = sum +x;
        }
        float average;              // declaration
        average = sum/(i-1);
        cout << average;

        return 0;
}
```

Dynamic Initialization of Variables:

C++, permits initialization of the variables at run time. This is referred to as dynamic initialization. In C++, a variable can be initialized at run time using expressions at the place of declaration. For example:

int n = strlen(string);

float area 3.14159 * rad * rad;

where string and rad are variables.

Thus, both the declaration and the initialization of a variable can be done simultaneously at the place where the variable is used for the first time.

Reference Variables

A reference variable provides an alias (alternative name) for a previously defined variable. For example, if we make the variable sum a reference to the variable total, then sum and total can be used interchangeably to represent that variable. A reference variable is created as follows:

data-type & reference-name = variable-name

Example:

float total=100;

float & sum=total;

where total is a float type variable that has already been declared; sum is the alternative name declared to represent the variable total. Both the variables refer to the same data object in the memory.

cout<<< total;

cout<<<sum;

will both print the value 100.

A reference variable must be initialized at the time of declaration. & is not an address operator. The notation float & means reference to float. Other examples are:

int n[10]; int & x = n[10]; // x is alias for n[10]

char & a = '\n';// initialize reference to a literal

A major application of reference variables is in passing arguments to functions. Consider the following:

```
void f(int & x)          // uses reference
{
  x = x+10;              // x is incremented; so also m
}
int main()
{
  int m = 10;
  f(m);                  // function call
  .....
  .....
}
```

Thus x becomes an alias of m after executing the statement

f(m);

Such function calls are known as **call by reference**. Since the variables x and m are aliases, when the function increments x, m is also incremented. The value of m becomes 20 after the function is executed.

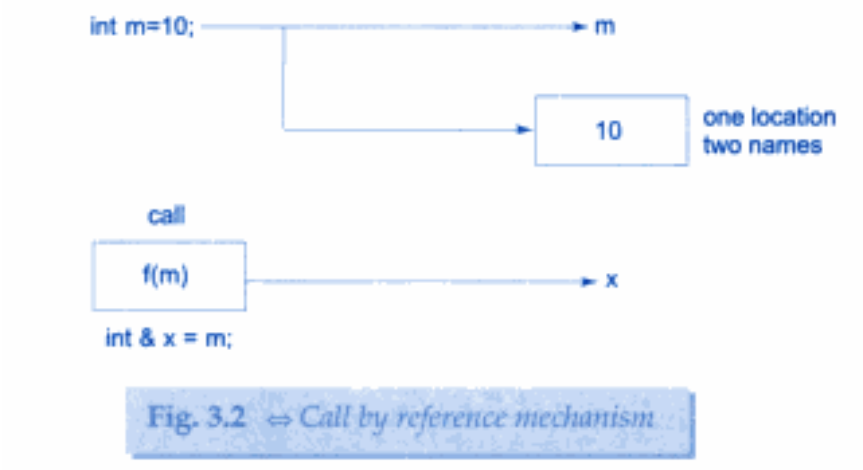


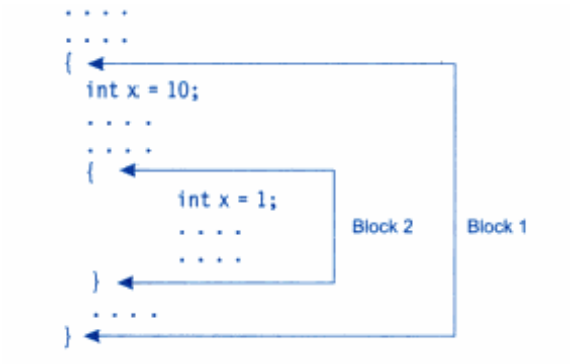
Fig. 3.2 ⇔ *Call by reference mechanism*

Call by value : in call by value (without the & in the parameter passed), a copy of the actual parameter is passed to the function (ie. The formal and actual parameters are different locations) and hence any changes made to the parameter inside the function do not reflect outside its scope.

```
void f(int x)
{
        x = x+10; // x is incremented but not m
}
int main()
{
        int m = 10;
        f(m); // after function call m is still 10
}
```

Scope Resolution Operator

The scope of the variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside a block is said to be local to that block. Blocks in C++ are often nested. For example,



A declaration in an inner block hides a declaration of the same variable in an outer block and, therefore, each declaration of x causes it to refer to a different data object x. To access the outer block's version of a variable from within the inner block, C++ introduces a new operator :: called the **scope resolution operator.** This can be used to uncover a hidden variable. It takes the following form:

```
:: variable-name
```

This operator allows access to the global version of a variable. For example, ::count means the global version of the variable count (and not the local variable count declared in that block). Program 3.1 illustrates this feature.

```
#include <iostream>

using namespace std;
int m = 10;          // global m

int main()
{
      int m = 20;    // m redeclared, local to main

      {
            int k = m;
            int m = 30;    // m declared again
                           // local to inner block

            cout << "we are in inner block \n";
            cout << "k = " << k << "\n";
            cout << "m = " << m << "\n";
            cout << "::m = " << ::m << "\n";
      }

      cout << "\nWe are in outer block \n";
      cout << "m = " << m << "\n";
      cout << "::m = " << ::m << "\n";

      return 0;
}
```

Member dereferencing Operators:

Table 3.3 *Member dereferencing operators*

Operator	Function
::*	To declare a pointer to a member of a class
*	To access a member using object name and a pointer to that member
->*	To access a member using a pointer to the object and a pointer to that member

Memory Management Operators:

We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. C++ still supports malloc() and calloc() functions to allocate memory dynamically at run time.

malloc(): or **"memory allocation"** function is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax of malloc: ptr = (cast-type*) malloc(byte-size)

Eg: ***ptr = (int*) malloc(100 * sizeof(int));***
In the above since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

C++ also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way. Since these operators manipulate memory on the free store, they are also known as **free store operators**.

An object can be created by using new, and destroyed by using delete, as and when required. A data object created inside a block with **new**, will remain in existence until it is explicitly destroyed by using **delete**. Thus, the lifetime of an object is directly under our control and is unrelated to the block structure of the program.

The new operator can be used to create objects of any type. It takes the following general form:

```
pointer-variable  =  new data-type;
```

Here, pointer-variable is a pointer of type data-type. The new operator allocates sufficient memory to hold a data object of type data-type and returns the address of the object. Examples:

p = new int;

q = new float;

where p is a pointer of type int and q is a pointer of type float. Here, p and q must have already been declared as pointers of appropriate types. Alternatively, we can combine the declaration of pointers and their assignments as follows:

```
int *p = new int;
float *q = new float;
```

If sufficient memory is not available for allocation, new returns a null pointer.

We can also initialize the memory using the new operator. This is done as follows:

```
pointer-variable  =  new data-type(value);
```

Example:

```
int *p = new int(25);
float *q = new float(7.5);
```

new can also be used to create an array as follows:

```
int *p = new int[10];
```

When creating multi-dimensional arrays with new, all the array sizes must be supplied. The first dimension may be a variable whose value is supplied at runtime. All others must be constants.

```
array_ptr = new int[3][5][4];    // legal
array_ptr = new int[m][5][4];    // legal
array_ptr = new int[3][5][ ];    // illegal
array_ptr = new int[ ][5][4];    // illegal
```

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

```
delete pointer-variable;
```

The pointer-variable is the pointer that points to a data object created with new. Examples:

delete p;

delete 4:

If we want to free a dynamically allocated array, we must use the following form of delete:

```
delete [ ]p;
```

The new operator offers the following advantages over the function malloc().(They may also be the difference)

1. It automatically computes the size of the data object. We need not use the operator sizeof.

2. It automatically returns the correct pointer type, so that there is no need to use a type cast.

3. It is possible to initialize the object while creating the memory space.

4. Like any other operator, new and delete can be overloaded.

Manipulators:

Manipulators are operators that are used to format the data display. The most commonly used manipulators are endl and setw.

-The endl manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character "\n".

-The setw manipulator is use to justify the output. For example,

cout<<< setw(5) <<< sum << endl;

The manipulator setw(5) specifies a field width 5 for printing the value of the variable sum. This value is right-justified within the field as shown below:

Type Cast Operator

C++ permits explicit type conversion of variables or expressions using the type cast operator as follows:

type-name (expression) ;

Examples: average =sum/float(i);

A type-name behaves as if it is a function for converting values to a designated type. The function-call notation can be used only if the type is an identifier. For example,

p = int * (q);

is illegal. In such cases, we must use C type notation.

p=(int *) q;

Alternatively, we can use typedef to create an identifier of the required type and use it in the functional notation.

typedef int * int_pt;

p = int_pt(q);

Expressions and Their Types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

Constant Expressions: Constant Expressions consist of only constant values. Examples:

```
15
20 + 5 / 2.0
'x'
```

Integral Expressions:

Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions. Examples:

m

5 + int (2.0)

where m is an integer variables.

Float Expressions:

Float Expressions are those which, after all conversions, produce floating-point results. Examples:

x + y

5 + float (10)

10.75

where x and y are floating-point variables.

Pointer Expressions:

Pointer Expressions produce address values. Examples:

&m

ptr

ptr + 1

where m is a variable and ptr is a pointer.

Relational Expressions:

Relational Expressions yield results of type bool which takes a value true or false. Examples:

x <= y

a+b == c+d

m+n > 100

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as Boolean expressions.

Logical Expressions:

Logical Expressions combine two or more relational expressions and produces bool type results. Examples:

a>b && x==10

x=-10 || y==5

Bitwise Expressions:

Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits. Examples:

x << 3// Shift three bit position to left

y >> 1// Shift one bit position to right

Special Assignment Expressions:

-Chained Assignment

x = (y = 10);

or

x = y = 10;

First 10 is assigned to y and then to x.

-Embedded Assignment

x = (y = 50) + 10;

(y = 50) is an assignment expression known as embedded assignment. Here, the value 50 is assigned to y and then the result 50+10= 60 is assigned to x.

-Compound Assignment

The general form of the compound assignment operator is:

variablel op= variable2;

where op is a binary arithmetic operator. This means that

variablel = variablel op variable2;

For example, the simple assignment statement

x = x + 10;

may be written as

x += 10;

The operator += is known as compound assignment operator or short-hand assignment operator.

Implicit Conversions:

We can mix data types in expressions. For example,

m = 5+2.75;

is a valid statement. Wherever data types are mixed in an expression, C++ performs the conversions automatically. This process is known as **implicit or automatic conversion**.

When the compiler encounters an expression, it divides the expressions into sub- expressions consisting of one operator and one or two operands. For a binary operator, if the operands type differ, the compiler converts one of them to match with the other, using the rule that the "smaller" type is converted to the "wider" type. For example, if one of the operand is an int and the other is a float, the int is converted into a float because a float is wider than an int. The "water-fall" model shown in Figure.

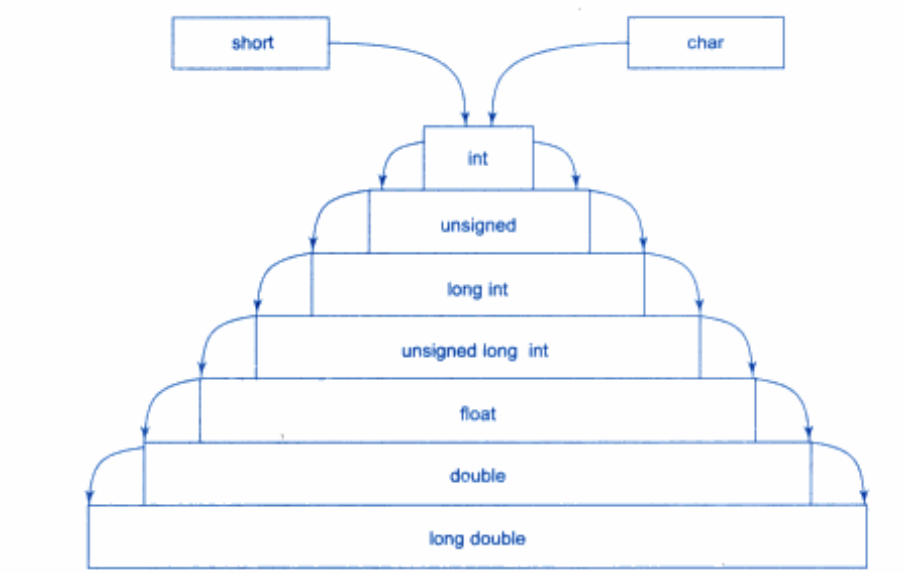


Fig. 3.3 ⇔ *Water-fall model of type conversion*

Whenever a char or short int appears in an expression, it is converted to an int. This is called integral widening conversion. The implicit conversion is applied only after completing all integral widening conversions.

Table 3.4 *Results of Mixed-mode Operations*

RHO / LHO	*char*	*short*	*int*	*long*	*float*	*double*	*long double*
char	int	int	int	long	float	double	long double
short	int	int	int	long	float	double	long double
int	int	int	int	long	float	double	long double
long	long	long	long	long	float	double	long double
float	float	float	float	float	float	double	long double
double	double	double	double	double	double	double	long double
long double	long double	long double	long double	long double	long double	long double	long double

RHO – Right-hand operand
LHO – Left-hand operand

Operator Precedence:

Table 3.5 *Operator precedence and associativity*

Operator	*Associativity*
::	left to right
–> . ( ) [ ] postfix ++ postfix – –	left to right
prefix ++ prefix – – ~ ! unary + unary – unary * unary & (type) sizeof new delete	right to left
–> * *	left to right
* / %	left to right
+ –	left to right
<< >>	left to right
<< = >> =	left to right
= = !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	left to right
= * = / = % = + = =	right to left
<< = >> = & = ^= |=	left to right
, (comma)	

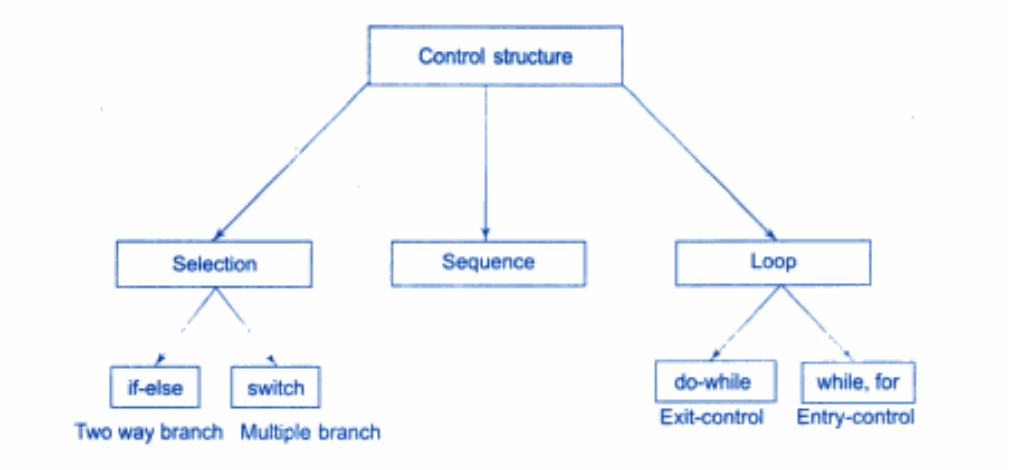
The unary operations assume higher precedence.

Control Structures:

1. Sequence structure (straight line)

2. Selection structure (branching)

3. Loop structure (iteration or repetition)



The **if** statement

The if statement is implemented in two forms:

- Simple if statement
- if...else statement

Form 1

```
if(expression is true)
{
   action1;
}
action2;
action3;
```

Form 2

```
if(expression is true)
{
   action1;
}
else
{
   action2;
}
action3;
```

-

The switch statement: This is a multiple-branching statement where, based on a condition, the control is transferred to one of the many possible points. This is implemented as follows:

```
switch(expression)
{
  case1:
  {
  action1;
  }
  case2:
  {
  action2;
  }
  case3:
  {
  action3;
  }
  default:
  {
  action4;
  }
}
action5;
```

The do-while statement:

The do-while is an exit-controlled loop. Based on a condition, the control is transferred back to a particular point in the program. The syntax is as follows:

```
do
{
  action1;
}
while(condition is true);
action2;
```

The while statement:

This is also a loop structure, but is an entry-controlled one. The syntax is as follows:

```
while(condition is true)
{
  action1;
}
action2;
```

The for statement:

The for is an entry-entrolled loop and is used when an action is to be repeated for a predetermined number of times. The syntax is as follows:

```
for(initial value; test; increment)
{
  action1;
}
action2;
```

The main() function:

In C++, the main() returns a value of type int to the operating system. C++, therefore, explicitly defines main() as matching one of the following prototypes:

int main();

int main(int argc, char *argv[]);

The functions that have a return value should use the return statement for termination. The main() function in C++ is, therefore, defined as follows:

```
int main()
{
    ......
    ......
    return 0;
}
```

Since the return type of functions is int by default, the keyword int in the main() header is optional.

Function prototyping:

A function prototype is a function declaration statement and is of the following form:

```
type function-name (argument-list);
```

The argument-list contains the types and names of arguments that must be passed to the function. Eg:

```
float volume(int x, float y, float z);
```

In a function declaration (prototype), the names of the arguments are dummy variables and therefore, they are optional. That is, the form

float volume (int, float, float); is valid

In general, we can either include or exclude the variable names in the argument list of prototypes. The variable names in the prototype just act as placeholders and, therefore, if names are used, they don't have to match the names used in the function call or function definition.

In the function definition, names are required because the arguments must be referenced inside the function. Example:

```
float volume(int a,float b,float c)
{
        float v = a*b*c;
        .....
        .....
}
```

The function volume() can be invoked in a program as follows:

float cubel = volume (b1, w1,h1); // Function call

The variable bl, w1, and h1 are known as the actual parameters. Their types should match with the types declared in the prototype.

Return by Reference:

A function can also return a reference. Consider the following function:

```
int & max(int &x,int &y)
{
        if (x > y)
              return x;
        else
              return y;
}
```

Since the return type of max() is int &, the function returns reference to x or y (and not the values). Then a function call such as max(a, b) will yield a reference to either a or b depending on their values. This means that this function call can appear on the left-hand side of an assignment statement. That is, the statement

max(a,b) = -1;

is legal and assigns -1 to a if it is larger, otherwise -1 to b.

Inline Functions

Every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a lot of time may be spent in such overheads. To eliminate the cost of calls to small functions, C++ proposes a new feature called **inline function**.

An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code. The inline functions are defined as follows:

```
inline function-header
{
   function body
}
```

Example:

```
inline double cube(double a)
{
   return(a*a*a);
}
```

To make a function inline, prefix the keyword inline to the function definition. All inline functions must be defined before they are called. The inline keyword, however, merely sends a request, not a command, to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a switch, or a goto exists.

2. For functions not returning values, if a return statement exists.

3. If functions contain static variables.

4. If inline functions are recursive.

Default Arguments:

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. Here is an example of a prototype (i.e. function declaration) with default values:

float amount (float principal, int period, float rate=0.15);

A subsequent function call like,

```
value = amount(5000,7);        // one argument missing
```

passes the value of 5000 to principal and 7 to period and then lets the function use default value of 0.15 for rate. The call

```
value = amount(5000,5,0.12);       // no missing argument
```

passes an explicit value of 0.12 to rate.

Only the trailing arguments can have default values .

Advantages of default arguments:

1. We can use default arguments to add new parameters to the existing functions.

2. Default arguments can be used to combine similar functions into one.

const Arguments:

In C++, an argument to a function can be declared as const as shown below.

int strlen(const char *p);

int length(const string &s);

The qualifier const tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated.

Function Overloading:

Overloading refers to the use of the same thing for different purposes. Function overloading means that we can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in OOP.

The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded add() function handles different types of data as shown below:

```
// Declarations
int add(int a, int b);                    // prototype 1
int add(int a, int b, int c);             // prototype 2
double add(double x, double y);           // prototype 3
double add(int p, double q);              // prototype 4
double add(double p, int q);              // prototype 5

// Function calls
cout << add(5, 10);              // uses prototype 1
cout << add(15, 10.0);           // uses prototype 4
cout << add(12.5, 7.5);          // uses prototype 3
cout << add(5, 10, 15);          // uses prototype 2
cout << add(0.75, 5);            // uses prototype 5
```

The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual argu- ments are the same, and use that function.

2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as, char to int, float to double to find a match.

3. When either of them fails, the compiler tries to use the built-in conversions to the actual arguments. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

long square (long n) double square (double x)

A function call such as,

square(10)

will cause an error because int argument can be converted to either long or double.

4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match

Structures

In C, structure is a user-defined data type with a template that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations. For example, consider the following declaration:

```
struct student
{
      char   name[20];
      int    roll_number;
      float total_marks;
};
```

The keyword struct declares student as a new data type that can hold three fields of different data types. Example:

struct student A; // C declaration

A is a variable of type student. Member variables can be accessed using the dot or period operator as follows:

A. roll_number= 999;

A.total_marks =595.5;

In C++, a structure can have both variables and functions as members. It can also declare some of its members as 'private' so that they cannot be accessed directly by the external functions. In C++, the structure names are stand-alone and can be used like any other type names i.e., the keyword struct can be omitted in the declaration of structure variables. For example, we can declare the student variable A as

student A; // C++ declaration

Note: The only difference between a structure and a class in C++ is that, by default, the members of a class are private, while, by default, the members of a structure are public.

A class:

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data type that can be treated like any other built-in data type.

The general form of a class declaration is:

```
class class_name
{
       private:
            variable declarations;
            function declarations;
       public:
            variable declarations;
            function declaration;
};
```

The keyword class specifies, that what follows is an abstract data of type class_name. The class body contains the declaration of variables and functions. These functions and variables are collectively called class members. They are usually grouped under two sections, namely, private and public to

denote which of the members are private and which of them are public. The keywords private and public are known as visibility labels.

The private class members can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also.

The variables declared inside the class are known as **data members** and the functions are known as **member functions**. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class.

Objects:

Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable). In C++, the class variables are known as **objects**.For example, consider the class item declared as follws:

```
class item
{
        int code;
        int price;
        public:
                void getdata(int c, float p)
                {
                        code=c;
                        price=p;
                }
}
```

The following statement,

```
item x;
```

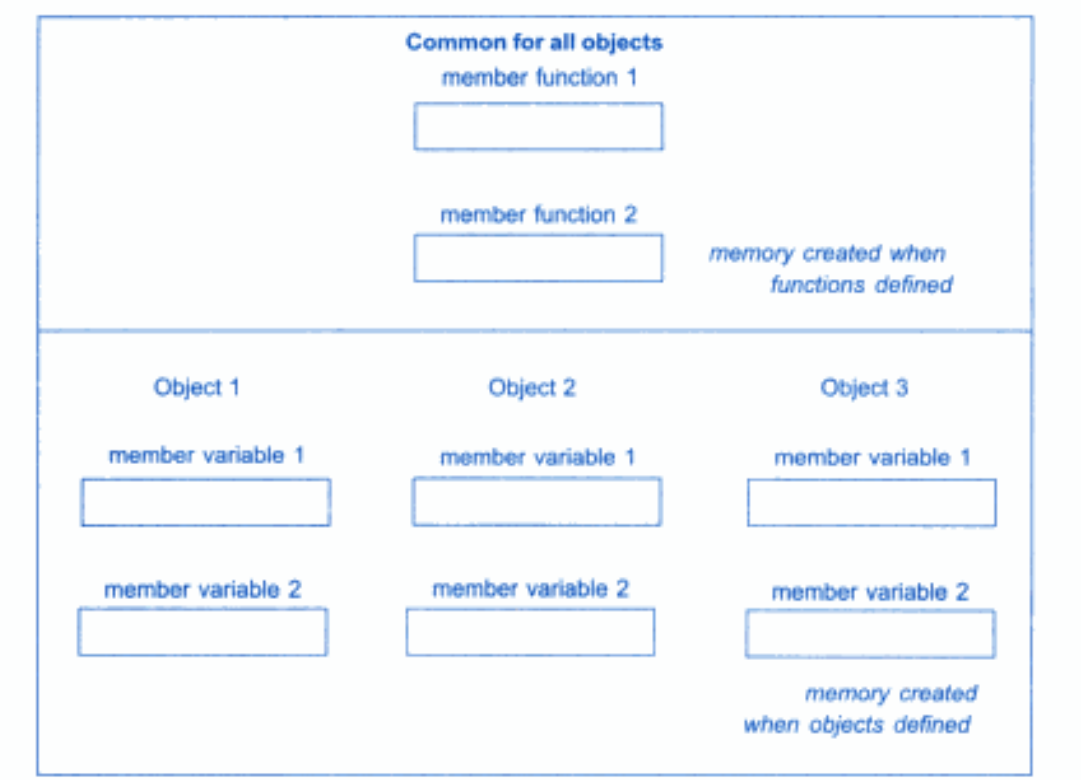
creates an object x of type item. The necessary memory space is allocated to an object at this stage. Objects can also be created when a class is defined by placing their names immediately after the closing brace as follows,

```
class item
{
        .....
        .....
        .....
} x,y,z;
```

Memory Allocation for Objects:

Memory space for objects is allocated when they are declared and not when the class is specified. Member functions, however, are created and placed in the memory space only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same

member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated separately for each object.



Accessing Class Members:

Private data of a class can be accessed only through the member functions of that class. The following is the format for calling a member function:

```
object-name.function-name (actual-arguments);
```

For example, the function call statement

x.getdata (100,75.5);

A variable declared as public can be accessed by the objects directly. Example:

```
class xyz
{
      int x;
      int y;
  public:
      int z;
};
      .....
      .....
xyz p;
p.x = 0;          // error, x is private
p.z = 10          // OK, z is public
      .....
      .....
```

The member functions have some special characteristics such as:

- Several different classes can use the same function name. The 'membership label' will resolve their scope.
- Member functions can access the private data of the class. A non-member function cannot do so.

- A member function can call another member function directly, without using the dot operator.

Defining Member Functions:

Member functions can be defined in two places:

-Outside the class definition:

Member functions that are declared inside a class have to be defined separately outside the class. When defined outside the class, a member function should include a membership 'identity label' in the header. The general form of a member function definition is:

```
return-type class-name :: function-name (argument declaration)
  {
        Function body
  }
```

The membership label class-name :: tells the compiler that the function function-name belongs to the class class-name. That is, the scope of the function is restricted to the class- name specified in the header line. The symbol :: is called the scope resolution operator.

-Inside the class definition:

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class. For example, we could define the item class as follows:

```
class item
{
        int code;
        int price;
        public:
                void getdata(int c, float p)
                {
                        code=c;
                        price=p;
                }
}
```

When a function is defined inside a class, it is treated as an **inline** function. Therefore, all the restrictions and limitations that apply to an inline function are also applicable here. Normally, only small functions are defined inside the class definition.

Making an Outside Function Inline:

We can define a member function outside the class definition and still make it inline by just using the qualifier inline in the header line of function definition. Example:

```
class item
{
      .....
      .....
  public:
      void getdata(int a, float b);          // declaration
};
```

```
inline void item :: getdata(int a, float b)  // definition
{
      number = a;
      cost = b;
}
```

Static Data Members:

A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are:

1. It is initialized to zero when the first object of its class is created. No other initialization is permitted.
2. Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
3. It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. The type and scope of each static member variable must be defined outside the class definition. Since they are associated with the class itself rather than with any class object, they are also known as class variables.

```
#include <iostream>

using namespace std;

class item
{
          static int count;
          int number;
  public:
          void getdata(int a)
          {
                number = a;
                count ++;
          }
          void getcount(void)
          {
                cout << "count: ";
                cout << count  << "\n";
          }
};

int item :: count;

int main()
{
```

```
    item a, b, c;          // count is initialized to zero
    a.getcount();          // display count
    b.getcount();
    c.getcount();

    a.getdata(100);        // getting data into object a
    b.getdata(200);        // getting data into object b
    c.getdata(300);        // getting data into object c

    cout << "After reading data" << "\n";

    a.getcount();          // display count
    b.getcount();
    c.getcount();
    return 0;
}
```

PROGRAM 5.4

The output of the Program 5.4 would be:

```
count: 0
count: 0
count: 0
After reading data
count: 3
count: 3
count: 3
```

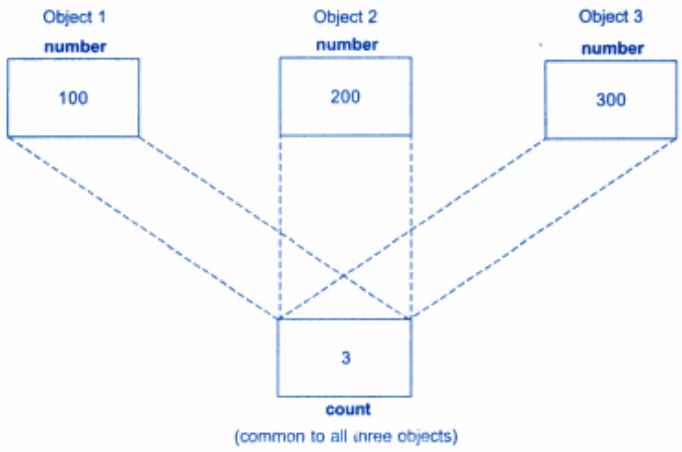


Fig. 5.4 ⇔ *Sharing of a static data member*

While defining a static variable, some initial value can also be assigned to the variable as follows,

int item count = 10;

Static Member Functions:

A member function that is declared static has the following properties:

• A static function can have access to only other static members (functions or variables) declared in the same class.

• A static member function can be called using the class name (instead of its objects) as follows:

class-name :: function-name;

In the program below, the static function showcount() displays the number of objects created till that moment. A count of number of objects created is maintained by the static variable count.

The function showcode() displays the code number of each object.

```
#include <iostream>

using namespace std;

class test
{
        int code;
        static int count;           // static member variable
public:
        void setcode(void)
        {
        code = ++count;
        }
        void showcode(void)
        {
        cout << "object number: " << code << "\n";
        }
        static void showcount(void)    // static member function
        {
        cout << "count: " << count << "\n";
        }
};

int test :: count;
int main()
{
        test t1, t2;

        t1.setcode();
        t2.setcode();

        test :: showcount();    // accessing static function

        test t3;
        t3.setcode();

        test :: showcount();

        t1.showcode();
        t2.showcode();
        t3.showcode();

        return 0;
}
```

PROGRAM 5.

Output of Program 5.5:

```
count: 2
count: 3
object number: 1
object number: 2
object number: 3
```

Arrays of Objects:

We can also have arrays of variables that are of the type class. Such variables are called arrays of objects. Consider the following class definition:

```
class employee
{
        char name[30];
        float age;
    public:
        void getdata(void);
        void putdata(void);
};
```

The following statements create objects that relate to different categories of the employees. Example:

employee manager[3];

employee foreman [15];

employee worker [75];

Objects as Function Arguments:

An object may be passed to a function argument in two ways:

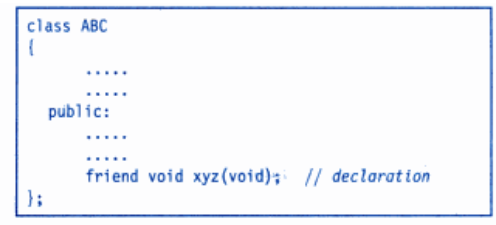
- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the function.

The first method is called pass-by-value. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The second method is called pass-by-reference. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Friendly Functions:

A non-member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. For example, consider a case where two classes, manager and scientist, have been defined. We would like to use a function income_tax() to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

To make an outside function "friendly" to a class, we have to simply declare this function as a friend of the class as shown below:



```
class ABC
{
      .....
      .....
  public:
      .....
      .....
      friend void xyz(void);   // declaration
};
```

The function declaration should be preceded by the keyword friend. The function is defined elsewhere in the program like a normal C++ function. The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a friend in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

A **friend function possesses certain special characteristics**:

- It is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.(e.g. A.x).
- It can be declared either in the public or the private part of a class without affect- ing its meaning.
- Usually, it has the objects as arguments.

```
#include <iostream>

using namespace std;

class sample
{
        int a;
        int b;
    public:
        void setvalue() {a=25; b=40; }
        friend float mean(sample s);
};
float mean(sample s)
{
        return float(s.a + s.b)/2.0;
}

int main()
{
        sample X;       // object X
        X.setvalue();
        cout << "Mean value = " << mean(X) << "\n";

        return 0;
}
```

Member functions of one class can be friend functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

```
class X
{
     .....
     .....
     int fun1();        // member function of X
     .....
};

class Y
{
     .....
     .....
     friend int X :: fun1();        // fun1() of X
                                    // is friend of Y
     .....
};
```

The function fun1() is a member of class X and a friend of class Y.

We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class**. This can be specified as follows:

```
class Z
{
     .....
     friend class X;    // all member functions of X are
                        // friends to Z
};
```

A FUNCTION FRIENDLY TO TWO CLASSES

```
#include <iostream>

using namespace std;

class ABC;      // Forward declaration
//-----------------------------------------------------//
class XYZ
{
        int x;
  public:
        void setvalue(int i) {x = i;}
        friend void max(XYZ, ABC);
};
//-----------------------------------------------------//
class ABC
{
        int a;
  public:
        void setvalue(int i) {a = i;}
        friend void max(XYZ, ABC);
};
```

```
//-----------------------------------------------------//
void max(XYZ m, ABC n)      // Definition of friend
{
        if(m.x >= n.a)
                cout << m.x;
        else
                cout << n.a;
}
//-----------------------------------------------------//
int main()
{
        ABC abc;
        abc.setvalue(10);
        XYZ xyz;
        xyz.setvalue(20);
        max(xyz, abc);

        return 0;
}
```

The function max() has arguments from both XYZ and ABC. When the function max() is declared as a friend in XYZ for the first time, the compiler will not acknowledge the presence of ABC unless its name is declared in the beginning as

class ABC;

This is known as 'forward' declaration.

Returning Objects:

A function cannot only receive objects as arguments but also can return them. Give an eg program.