MANONMANIAM SUNDARANAR UNIVERSITY

TIRUNELVELI-627 012

DIRECTORATE OF DISTANCE AND CONTINUING EDUCATION

II M.Sc. MATHEMATICS

SEMESTER III

SKILL ENHANCEMENT COURSE - II : PROGRAMMING IN C++

Sub. Code: SMAS31

Prepared by

Dr. Leena Nelson S N

Associate Professor & Head, Department of Mathematics

Women's Christian College, Nagercoil – 1.

PROGRAMMING IN C++ (SMAS31)

UNIT	DETAILS
I	Structures of C++ program – Tokens – Keywords – Identifiers and constants – all data types – Constants – all variables – All operators - Manipulator
II	All Expressions - Conversion – Operator overloading – Operator Precedence – Control Structures – Functions in C++ - Introduction – Main Function – Function Prototyping – Return by reference
III	Inline Functions – arguments – Function overloading – all functions classes and objects
IV	Nesting of member functions – Private member function – Arrays within a class and Objects – Friendly function – Returning Objects – Pointers to members – Local classes
V	Constructors and Destructors – Operator overloading and Type Conversions.

Text Book
E. Balagurusamy, Object Oriented Programming with C++, 4th Edition, Tata McGraw-Hill Company, New Delhi, 2008.

UNIT I

TOKENS, EXPRESSIONS AND CONTROL STRUCTURES

1.1 Introduction

C++ is a superset of C and therefore most constructs of C are legal in C++ with their meaning unchanged. However there are some exceptions and additions. In this chapter, we shall discuss these exceptions and additions with respect to tokens and control structures.

1.2 TOKENS

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

- Keywords
- Identifiers
- Constants
- Strings
- Operators

1.3 KEYWORDS

Table 1.1 gives the complete sets of C++ keywords. Many of them are common to both C and C++.

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

Added by ANSI C++			
bool	export	reinterpret_cast	typename
const_cast	false	static_cast	using
dynamic_cast	mutable	true	wchar_I
explicit	namespace	typeid	

Table 1.1

1.4 IDENTIFIERS AND CONSTANTS

Identifiers refer to the names of variables, functions, arrays, classes etc., created by the programmer. They are the fundamental requirements of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++.

- Only alphabetic characters, digits and underscores are permitted.
- The name cannot start with a digit.
- Uppercase and lowercase letters are distinct.
- A declared keyword cannot be used as a variable name.

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name. ANSI C++ places no limit on its length and, therefore all the characters in a name are significant.

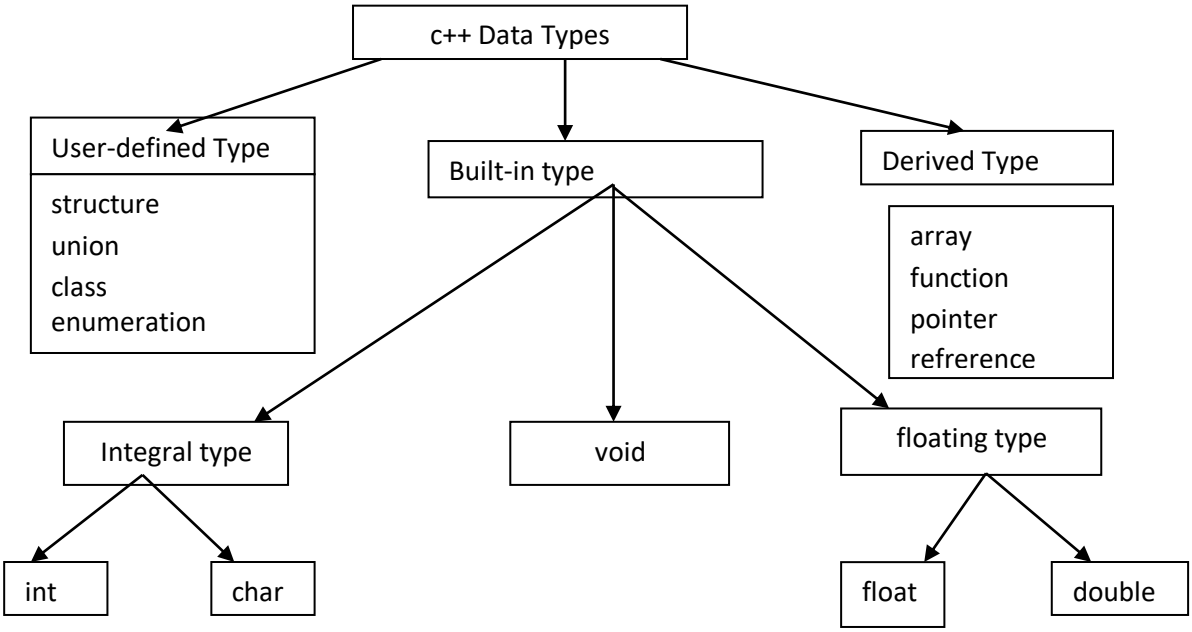
Constants refer to fixed values that do not change during the execution of a program. Like C, C++ supports several kinds of literal constants. They include integers, characters, floating point numbers and strings. Literal constants do not have memory locations. Examples,

```
123          // decimal integer
12.34        // floating point integer
037          // octal integer
0X2          // hexadecimal integer
"C++"        // string constant
'A'          // character constant
L'ab'        // wide-character constant
```

The **wchar_t** type is a wide character literal introduced by ANSI C++ and is intended for character sets that cannot fit a character into a single byte. Wide-character literals begin with the name L.

C++ also recognizes all the backslash character constants available in C.

1.5 BASIC DATA TYPES



Both C and C++ compilers support all the built-in 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	-32768 to 32767
short int	2	-32768 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

Table 1.2 size and range of C++ b1asic data types

The type **void** was introduced in ANSI C. Two normal uses of void are

(i) to specify the return type of a function when it is not returning any value &

(ii) to indicate an empty argument list to a function.

Example : *void func1(void);*

Another interesting use of void is in the declaration of generic pointers.

Example : *void *gp; // gp becomes generic pointer*

A generic pointer can be assigned a pointer value of any basic data type, but it may be dereferenced.

For example, *int *ip; // int pointer*

gp=ip; // assign int pointer to void pointer

are valid statements. But, the statement,

**ip=*gp;*

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

1.6 USER-DEFINED DATA TYPES

Structures and Unions

Arrays are used to group together similar type data elements, structures are used for grouping together elements with dissimilar types.

The general format of a structure definition is as follows:

```
struct name
{
        data_type member1;
        data_type member2;
        ..... .....
        ..... .....
};
```

Let us take the example of a book, which has several attributes such as title, number of pages, price, etc.

```
struct book
{
chartitle[25];
charauthor[25];
int pages;
float price;
};
struct book book1, book2, book3;
```

Here book1, book2 and book3 are declared as variables of the user-defined type book.

Unions are conceptually similar to structures as they allow us to group together dissimilar type elements inside a single unit. But there are significant differences between structures and unions as far as their implementation is concerned. The size of a structure type is equal to the sum of the sizes of individual member types. However, the size of a union is equal to the size of its largest number element. For instance, consider the following union declaration:

```
union result
{
        int marks;
        char grade;
        float percent;
};
```

In C++, structures and unions can be used just like they are used in C.

Table 1.3 Difference between structures and unions

Structure	Union
• A structure is defined with 'struct' keyword. • All members of a structure can b manipulated simultaneously. • The size of a structure object is equal to the sum of the individual sizes of the member objects.	• A union is defined with 'union' keyword. • The members of a union can be manipulated only one at a time. • The size of a union object is equal to the size of largest member object.

• Structure members are allocated distinct memory locations. • Structures are not considered as memory efficient in comparison to unions. • Structure in C++ behave just like a class. Almost everything that can be achieved with a class can also be done with structures.	• Union members share common memory space for their exclusive usage. • Unions are considered as memory efficient particularly in situations when the members are not required to be accessed simultaneously. • Unions retain their core functionality in C++ with slight add-ons like declaration of anonymous unions.

Classes

C++ also permits us to define another user-defined data type known as **class** which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming.

Enumerated Data Type

An enumerated data type is another user-defined type which provides a way for attaching names to numbers. The **enum** keyword (from C) automatically enumerate a list of words by assigning them values 0, 1, 2 and so on. The syntax of an **enum** statement is similar to that of the **struct** statement.

Examples : *enum shape{circle, square, triangle};*

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

enum position{off, on};

The enumerated data types differ slightly in C++ when compared with those in ANSI C. 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; // ellipse is of type **shape***

*colour background; // background is of type **colour***

1.7 STORAGE CLASSES

Automatic : It is the default storage class of any type of variable. Its visibility is restricted to the function in which it is declared. Further, its lifetime is also limited till the time its container function is executing.

External: As the name suggests, an external variable is declared outside of a function but is accessible inside the function block. Also called global variable, its visibility is spread all across the program that means, it is accessible by all the functions present in the program.

Static: A static variable has the visibility of a local variable but the lifetime of an external variable. That means, once declared inside a function block, it does not get destroyed after the function is executed, but retains its value so that it can be used by future function calls.

Register: Similar in behaviour to an automatic variable, a register variable differs in the manner in which it is stored in the memory. Unlike, automatic variables that are stored in the primary memory, the register variables are stored in CPU registers. The objective of storing a variable in registers is to increase its access speed, which eventually makes the program run faster.

Table 1.4 gives a summary of the four storage classes:

	Automatic	External	Static	Register
Lifetime	Function Block	Entire program	Entire program	Function block
Visibility	Local	Global	Local	Local
Initial Value	Garbage	0	0	Garbage
Storage	Stack segment	Data segment	Data segment	CPU Registers
Purpose	Local variables used by a single function	Global variables used throughout the program.	Local variables retaining their values throughout the program	Variables using CPU registers for storage purpose
Keyword	auto	extern	static	Register

1.8 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++

Functions

Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs. Many of these modifications and improvements were driven by the requirements of the object-oriented concept of C++.

Pointers

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.

*char * const ptr1 = "Good"; // constant pointer*

We cannot modify the address that ptr1 is initialized to.

*int const *ptr2 = &m; // pointer to a constant*

ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents of what it points to cannot be changed.

We can also declare both the pointer and the variable as constants in the following way:

*const char *const cp = "xyz";*

This statement declared cp as a constant pointer to the string which has been declared a constant.

1.9 SYMBOLIC CONSTANTS

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

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

In both C and C++, any value declared as const cannot be modified by the program in any way. However, there are some differences in implementation in C++, we can use **const** in a constant expression, such as

const int size = 10;

char name[size];

This would be illegal in C. **const** allows us to create typed constants instead of having to use #define to create constants that have no type information.

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

const size = 10;

means

const int size = 10;

The named constants are just like variables except that their values cannot be changed.

C++ requires a **const** to be initialized. ANSI C does not require an initialize, if none is given, it initializes the **const** to 0.

1.10 TYPE COMPATIBILITY

C++ is very strict with regard to type compatibility as compared to C. For instance, C++ defines **int**, **short int** and **long int** as three different types. They must be cast when their values are assigned to one another. Similarly, **unsigned char**, **char** and **signed char** are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility, or else, a cast must be applied. These restrictions in C++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

Another notable difference is the way **char** constants are stored. In C, they are stored as **int**s, and therefore,

sizeof ('x')

is equivalent to

sizeof(int)

in C. In C++, however char is not promoted to the size of int and therefore,

sizeof('x')

equals

sizeof(char)

1.11 DECLARATION OF VARIABLES

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 that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use.

The example below illustrates this point.

```
int main()
{
  float x;                    // declaration
```

```
    float sum=0;
        for (int i=1; i<5; i+1)          // declaration
        {
            cin>> x;
            sum=sum+x;
        }
    float average;          // declaration
    average = sum / (i-1);
    count << average;
    return 0;
}
```

1.12 DYNAMIC INITIALIZATION OF VARIABLES

In C, a variable must be initialized using a constant expression, and the C compiler would fix the initialization code at the time of compilation. C++, however, 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, the following are valid initialization statements:

```
.....
.....
int n = strlen(string);
.....
float area = 3.14159 * rad * rad;
```

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. The following two statements in the example of the previous section.

```
float average;          // declare where it is necessary
average = sum/i;
```

can be combined into a single statement :

```
float average = sum/i;          // initialize dynamically at run time
```

Dynamic initialization is extensively used in object-oriented programming. We can create exactly the type of object needed, using information that is known only at the run time.

1.13 REFERENCE VARIABLE

C++ introduces a new kind of variable known as reference variable. 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

For example,

```
float total = 100;
float & sum = total;
```

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. Now, the statement,

```
count << total;
```

and

```
count << sum;
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both total and sum to 110. Likewise, the assignment

```
sum = 0;
```

will change the value of both the variable to zero.

C++ assigns additional meaning to the symbol &. Here, & 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
```

The variable x is an alternative to the array element n[10]. The variable a is initialized to the newline constant. This creates a reference to the otherwise unknown location where the newline constant \n is stored.

1.14 OPERATORS IN C++

C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators. We have already seen two such operators, namely, the insertion operator <<, and the extraction operator >> . Other new operators are:

::	scope resolution operator
::*	Pointer-to-member declarator
→*	Pointer-to-member operator
.*	Pointer-to-member operator
delete	Memory release operator
endl	Line feed operator
new	Memory allocation operator
setw	Field width operator

In addition, C++ also allows us to provide new definitions to some of the built-in-operators. That is, we can give several meanings to an operator, depending upon the types of arguments used. This process is known as operator overloading.

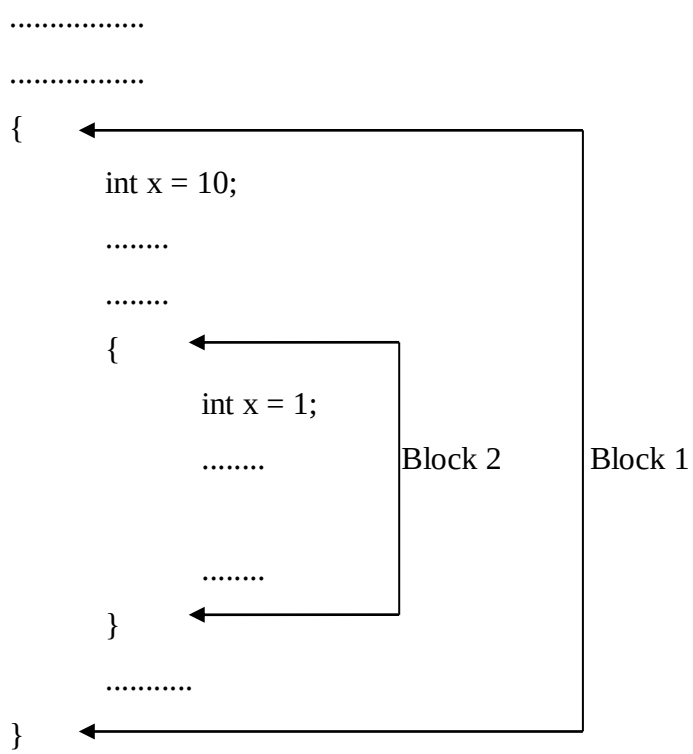
1.15 SCOPE RESOLUTION OPERATOR

Like C, C++ is also a block-structured language. Blocks and scopes can be used in constructing programs. We know that the same variable name can be used to have different meanings in different blocks. 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. Consider the following segment of a program:

```
.....
.....
{
  int x = 10;
  .....
  .....
```

```
}
.....
.....
{
  int x = 1;
  .....
  .....
}
```

The two declarations of x refer to two different memory locations containing different values. Statements in the second block cannot refer to the variable x declared in the first block, and vice versa. Blocks in C++ are often nested. For example, the following style is common.



Block2 is contained in block 1. Note that 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. Within the inner block, the variable x will refer to the data object declared therein.

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing 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 count (and not the local variable count declared in that block). The following program 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
        count << "we are in inner block \n";
        count << "k= " << k << "\n";
        count << "m= " << m << "\n";
        count << "::m= " << ::m << "\n";

  }
  count << "\n We are in outer block \n";
  count << "m=" << m << "\n";
  count << ":: m =" << ::m << "\n";
  return 0;
}
```

The output of the program would be:

```
We are in inner block
k = 20
m = 30
::m = 10

We are in outer block
m = 20
::m = 10
```

1.16 MEMBER DEREFERENCING OPERATORS

C++ permits us to define a class containing various types of data and functions as members. C++ also permits us to access the class members through pointers. In order to achieve this, C++ provides a set of three pointer-to-member operators.

Table 1.5 shows these operators and their functions

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

1.17 MEMORY MANAGEMENT OPERATORS

C uses malloc( ) and calloc( ) functions to allocate memory dynamically at run time. Similarly, it uses the function free( ) to free dynamically allocated memory. We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. Although C++ supports these functions, it also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

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. The data-type may be any valid data type. The pointer-variable holds the address of the memory space allocated.

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;*

Subsequently, the statements

**p = 25;*

**q = 7.5;*

assign 25 to the newly created int object and 7.5 to the float object.

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

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

Here, value specifies the initial value. Examples:

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

As mentioned earlier, new can be used to create a memory space for any data type including user-defined types such as arrays, structures and classes. The general form for a one-dimensional array is:

pointer-variable = new data-type (size);

Here, size specifies the number of elements in the array. For example, the statement

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

creates a memory space for an array of 10 integers. p[0] will refer to the first element, p[1] to the second element, and so on.

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 q;

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

delete [size] pointer-variable;

The size specifies the number of elements in the array to be freed. The problem with this form is that the programmer should remember the size of the array. Recent versions of C++ do not require the size to be specified. For example,

delete [ ]p;

will delete the entire array pointed to by p.

Program 1.1 Use of new and delete Operators

```
#include <iostream>

#include <conio.h>

using namespace std;

void main ( )

{

  int *arr;
  int size;

  count<<"Enter the size of the integer array: ";
  cin>>size;

  cout<<"Creating an array of size "<<size<<"..";
  arr = new int[size];

  cout<<"\nDynamic allocation of memory for array arr is successful.";
```

```
    delete arr;
    getch( );
}
```

The output of Program 1.1 would be:

Enter the size of the integer array: 5

Creating an array of size 5 ..

Dynamic allocation of memory for array arr is successful.

1.18 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, one for each variable. If we assume the values of the variable as 2597, 14 and 175 respectively, the output will appear as follows:

m=	2	5	9	7

n=	1	4

p=	1	7	5

It is important to note that this form is not the ideal output. It should rather appear as under:

m = 2597

n = 14

p = 175

Here, the numbers are right justified. This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The **setw** manipulator does this job. It is used as follows:

```
count << setw(5) << sum << endl;
```

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

		3	4	5

Program 1.4 Use of Manipulators

```
#include <iostream>
#include <iomanip>        // for setw
using namespace std;
int main( )
{
        int Basic = 950, Allowance = 95, Total = 1045;
        count << setw(10) << "Basic" << setw(10) << Basic << endl
                << setw(10) << "Allowance" << setw(10) << Allowance << endl
                << setw(10) << "Total" << setw(10) << Total << endl;
        return 0;
}
```

The output of the above program would be

```
Basic          950
Allowance       95
Total         1045
```

UNIT II

OPERATORS & FUNCTIONS IN C++

2.1 TYPE CAST OPERATOR

C++ permits explicit type conversion of variables or expressions using the type cast operator.

Traditional C casts are augmented in C++ by a function-call notation as a syntactic alternatives. The following two versions are equivalent:

(type-name) expression *// C notation*

type-name (expression) *// C++ notation*

Example:

average = sum(float)i; *// C notation*

average=sum/float(i); *// C++ notation*

A type-name behaves as if it is a function for converting values to a designated type. The function-call notation usually leads to simplest expressions. However, it 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);

Program 2.1 Explicit Type Casting

```
#include <iostream>
#include <conio.h>
using namespace std;
int main( )
{
  int intvar = 25;
  float floatvar = 35.87;
  count<<"intvar = "<<intvar;
  count<<"\nfloatvar = "<<floatvar;
  count<<"\nfloat(intvar) = "<<float(intvar);
  count<< "\nint(floatvar) = "<<int(floatvar);
  getch( );
}
```

The output of Program 2.1 would be

intvar = 25

floatvar = 35.87

float(intvar) = 25

int(floatvar) = 35

2.2 EXPRESSIONS AND THEIR TYPES

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values. An expression may consist of one or more operands, and zero or more operators to produce a value. Expressions may be of the following seven types:

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

An expression may also use combinations of the above expressions. Such expressions are known as compound 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

*m * n - 5*

*m * 'x'*

5 + int (2.0)

where **m** and **n** are integer variables.

Float Expressions :

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

x + y
*x * y / 10*
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

"xyz"

where **m** is a variable and **ptr** is a pointer.

Relational Expressions :

Relational Expressions yields 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

Shift operators are often used for multiplication and division by powers of two.

ANSI C++ has introduced what are termed as operator keywords that can be used as alternative representation for operator symbols.

2.3 SPECIAL ASSIGNMENT EXPRESSIONS

Chained Assignment :

x = (y = 10) ;

or

x = y = 10;

First 10 is assigned to y and then to x.

A chained statement cannot be used to initialize variables at the time of declaration. For instance, the statement

float a = b = 12.34; // wrong

is illegal. This may be written as

float a = 12.34, b = 12.34; // correct

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. This statement is identical to

y = 50;
x = y + 10

Compound Assignment :

Like C, C++ supports a compound assignment operator which is a combination of the assignment operator with a binary arithmetic operator. 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. The general form of the compound assignment operator is :

variable op= variable2;

where op is a binary arithmentic operator. This means that

variable1 = variable1 op variable2;

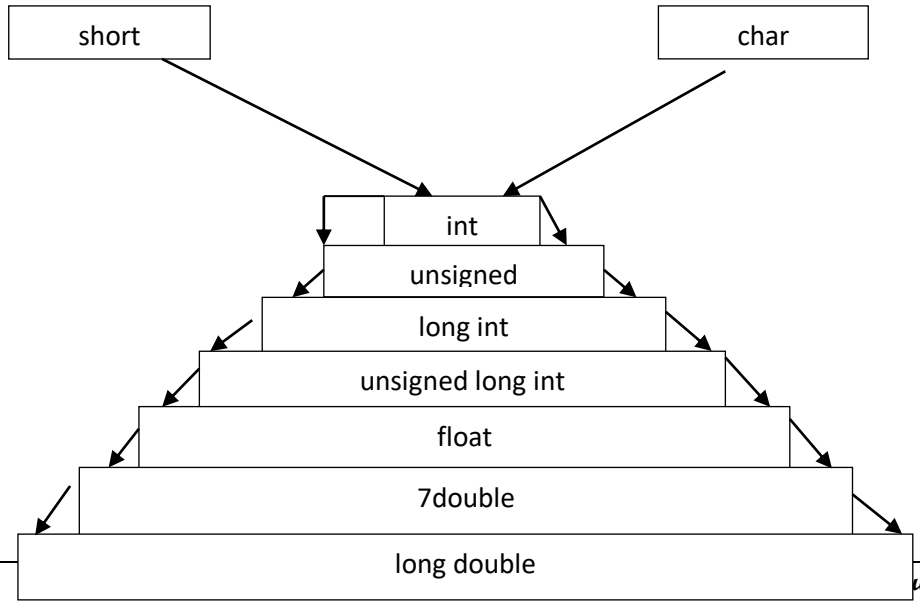
2.4 IMPLICIT CONVERSIONS

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 conversions.

When the compiler encounters an expression, it divides the expressions into sub-expressions consisting of the 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 illustrates this rule.



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 : 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

2.5 OPERATOR OVERLOADING

As stated earlier, overloading means assigning different meanings to an operation, depending on the context. C++ permits overloading of operators, thus allowing us to assign multiple meanings to operators. Actually, we have used the concept of overloading in C also. For example, the operator* when applied to a pointer variable, gives the value pointed to by the pointer. But it is also commonly used for multiplying two numbers . The number and type of operands decide the nature of operation to follow.

The input/output operators << and >> are good examples of operator overloading. Although the built-in definition of the << operator is for shifting of bits, it is also used for displaying the values of various data types. This has been made possible by the header life iostream where a number of overloading definitions for << are included. Thus, the statement

cout << 75.86;

invokes the definition for displaying a double type value, and

cout << "well done";

invokes the definition for displaying a char value. However, none of these definitions in iostream affect the built-in meaning of the operator.

Similarly, we can define additional meanngs to other C++ operators. For example, we can define + operator to add two structures or objects. Almost all C++ operators can be overloaded with a few exceptions such as the member-access operators (.and.*), conditional operator (?:), scope resolution operator (::) and the size operator (sizeof). Definitions for operator overloading are discussed in detail in .....

2.6 OPERATOR PERCEDENCE

Although C++ enables us to add multiple meanings to the operators, yet their association and precendence remain the same. For example, the multiplication operator will continue having higher precendence than the add operator. Table gives the precendence and associativity of all the C++ operators. The groups are listed in the order of decreasing precendence. The labels prefix and postfix distinguish the uses of ++ and --. Also, the symbols +, -,*, and & are used as both unary and binary operators.

A complete list of ANSI C++ operators and their meanings, precendence, associativity and use are given in Appendix E.

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
<< = >> = & = ^= |= , (comma)	left to right

Note : The unary operations assume higher precedence.

2.7 INTRODUCTION

We know that functions play an important role in C program development. Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

Recall that we have used a syntax to the following in developing C programs.

```
void show ( ) ;              / * Function declaration * ?
main ( )
{
        . . . . .
        show ( ) ;           / * Function call * /
        . . . . .
}
void show ( )                / * Function definition * /
{
        . . . . .
```

```
. . . . .                    / * Function body * /
. . . . .
}
```

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered. C++ is no exception. Functions continue to be the building blocks of C++ programs. In fact, C++ has added many new features to functions to make them more reliable and flexible. Like C++ operators, a C++ function can be overloaded to make it perform different tasks depending on the arguments passed to it. Most of these modifications are aimed at meeting the requirements of object-oriented facilities.

2.8 THE MAIN FUNCTION

C does not apply any return type for the main( ) function which is the starting point for the execution of a program. The definition of main( ) would look like this:

```
main( )
{
        // main program statements
}
```

This is perfectly valid because the main( ) in C does not return any value.

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 phototypes:

```
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. Most C++ compilers will generate an error or warning if there is no return statement. Turbo C++ issues the warning

Function should return a value

and then proceeds to compile the program. It is good programming practice to actually return a value from main( ).

Many operating systems lost the return value (called exit value) to determine if there is any problem. The explicit use of return(0) statement will indicate that the program was successfully executed.

2.9 FUNCTION PROTOTYPING

Function prototyping is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. With function prototyping, a template is always used when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself. These checks and controls did not exist in the conventional C functions.

Function prototype is a declaration statement in the calling program 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. Example:

float volume{int x, float y, float z};

Note that each argument variable must be declared independently inside the parantheses. That is, a combined declaration like

float volume{int x, float y, z};

is illegal.

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

float volume{int, float, float};

is acceptable at the place of declaration. At this stage, the compiler only checks for the type of arguments when the function is called.

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 cube1 = volume(b1, w1, h1);     // Function call
```

The variable b1, w1 and h1 are known as the actual parameters which specify the dimensions of cube1. Their types should match with the types declared in the prototype.

2.10 CALL BY REFERENCE

In traditional C, a function call passes arguments by value. The called function creates a new set of variables and copies the values of arguments into them. The function does not have access to the actual variables in the calling program and can only work on the copies of values. This mechanism is fine if the function does not need to alter the values of the original variables in the calling programe. But, there may arise situations where we would

like to change the values of variables in the calling program. For example, in bubble sort, we compare two adjacent elements in the list and interchange their values if the first element is greater than the second. If a function is used for bubble sort, then it should be able to alter the values of variables in the calling function, which is not possible if the call-by-value method is used.

Provision of the reference variables in C++ permits us to pass parameters to the functions by reference. When we pass arguements by reference, the 'formal' arguments in the called function become allases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data. Consider the following function:

```
void swap ( int &a, int &b )    // a and b are reference variables
{
        int t = a ;
        a = b ;                 // Dynamic initialization
        b = t ;
}
```

Now, if m and n are two integer variables, then the function call

```
swap ( m, n ) ;
```

will exchange the values of m and n using their aliases (reference variables) a and b. In traditional C, this is accomplished using pointers and indirection as follows:

```
void swapl ( int *a, int *b )       / * Function definition * /
{
        int t ;
        t = *a ;                    / * assign the value at address a to t */
        *a = *b ;                   / * put the value at b into a */
        *b = t;             / * put the value at t into b */
}
```

This function can be called as follows:

```
swapl ( &x , &y )       / * call by passing * /

                        / * addresses of variables * /
```

This approach is also acceptable in C++. Note that the call-by-reference method is neater in its approach.

2.11 RETURN BY REFERENCE

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

```
int & max ( int &x, int &y 0
{
        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.

UNIT III

INLINE FUNCTIONS, CLASSES & OBJECTS

3.1 INLINE FUNCTIONS

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. However, 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 substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as macros. Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.

C++ has a different solution to this problem. 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);
}
```

The above inline function can be invoked the statements like

```
c = cube(3.0);

d = cube(2.5+1.5);
```

On the execution of these statements, the values of c and d will be 27 and 64 respectively. If the arguments are expressions such as 2.5+1.5, the function passes the value of the expression, 4 in this case. This makes the inline feature far superior to macros.

We should exercise care before making a function inline. The speed benefits of inline function diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of inline functions may be lost. In such cases, the use of normal functions will be more meaningful. Usually, the functions are made inline when they are small enough to be defined in one or two lines. Example:

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

Remember that the inline keyword 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.

Program 3.1 Inline Functions

```
#include <iostream>
using namespace std;
    inline float mul(float x, float y)
    {
        return (x*y);
    }
    inline double div(double p, double q)
    {
        return(p/q);
    }
```

```
int main( )
{
        float a = 12.345;
        float b = 9.82;
        count << mul(a,b) << "\n";
        count << div(a,b) << "\n";
        return 0;
}
```

The output of Program 3.1 would be

121.228

1.25713

3.2 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);

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument rate. 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 or 0.12 to rate.

Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a

particular period of deposit. It also provides a greater flexibility to the programmers. A function can be written with more parameters than are required for its most common application. Using default arguments, a programmer can use only those arguments that are meaningful to a particular situation. Program 4.2 illustrates the use of default arguments.

Program 3.2 Default Arguments

```
#include<iostream>
#include<conio.h>
using namespace std;
int main( )
{
        float amount;
        float value(float p, int n, float r=0.15);        // prototype
        void printline(char ch='*', int len=40);        // prototype
        printline( );                                   // use default values for arguments
        amount = value(5000.00,5);              // default for 3rd argument
        cout<<"\n Final Value = "<<amount<<"\n\n";
        printline('=');          // use default value for second argument
        getch( );
        return 0;
}
float value (float p, int n, floar r)
{
        int year = 1;
        float sum = p;
        while (year<=n)
        {
                sum = sum*(1+r);
                year = year+1;
        }
        return (sum);
}
```

```
void printline(char ch, int len)
{
        for (int i=1; i<=len;i++)
        printf("&c",ch);
        printf("\n");
}
```

The output of Program 4.2 would be:

```
**********************************************
Final Value = 10056.8
Final Value = 37129.3
==============================
```

Advantages of providing the default arguments are:

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.

3.3 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. This type of declaration is significant only when we pass arguments by reference or pointers.

3.4 RECURSION

Recursion is a situation where a function calls itself meaning, one of the statementsin the function definition makes a call to the same function in which it is present. It may sound like an infinite looping condition but just as a loop has a conditional check to take the program control out of the loop, a recursive function also possesses a base case which returns the program control from the current instance of the function to call back to the calling function. For example, in a series of recursive calls to compute the factorial of a number, the base case would be a situation where factorial of 0 is to be computed. Let us consider a few examples (Program 4.3 and 4.4) to understand how the recursive approach works.

Program 3.4 Calculating factorial of a number

```
# include <iostream>
# include <conio.h>
using namespace std;

long fact (int n )
{
        if ( n == 0 ) // base case
        return 1;
        return ( n * fact ( n - 1 ) ; //recursive function call
}

int main ( )
{
        int num ;
        cout << " Enter a positive integer : ";
        cin >> num ;
        cout << "Factorial of " << num << "is" << fact ( num) ;
        getch ( ) ;
        return 0 ;
}
```

The output of program 3.4 would be:

```
Enter a positive integer : 10

Factorial of 10 is 3628800
```

Program 3.5 Solving Tower of hanoi Problem

```
# include <iostream>
# include < conio.h>
using namespace std;

void TOH ( int d, char tower1, char tower2, char tower3)
{
```

```
        if ( d == 1 ) // base case
        {
          cout << "\nShift top disk from tower " << tower1<< " to tower" << tower
2 ;
    return;
        }
    TOH ( d - 1, tower3,  tower2, tower1) ; // recursive function call

    }
    int main ( )
    {
        int disk ;
        cout << " Enter the number of disks: ";
        cin >> disk;

        if ( disk < 1 )
          cout << "\nThere are no disks to shift ";
        else
          cout << "\nThere are " << disk << " disk in tower 1 \n";
        TOH ( disk, '1', '2', '3') ;
        cout << "\n \ n " << disk << "disk in tower 1 are shifted to tower 2 "

        getch ( );
        retorn 0 ;
    }
```

The output of program 3.5 would be :

Enter the number of disks : 3

There are 3 disks in tower 1

Shift top disk from tower 1 to tower 2

Shift top disk from tower 1 to tower 3

Shift top disk from tower 2 to tower 3

Shift top disk from tower 1 to tower 2

Shift top disk from tower 3 to tower 1

Shift top disk from tower 3 to tower 2

Shift top disk from tower 1 to tower 2

3 disks in tower 1 are shifted to tower 2

3.5 FUNCTION OVERLOADING

As stated earlier, overloading refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This 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.

Using the concept of function overloading ; we can design a family of functions with one function name but with different argument lists. 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, on 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
doubke 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
```

A function call first matches the prototype having the same number and types of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments 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 (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique.

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, thereby creating an ambiguous situation as to which version of square( ) should be used.

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. User- defined conversions are often used in handling class objects.

Program 3.6 Function Overloading

```
// Function volume( ) is overloading three times
#include <iostream>
using namespace std;
// Declarations (prototypes)
int volume(int);
double volume(double, int);
long volume(long, int, int);
int main( )
{
        count<<"Calling the volume( ) function for computing the volume of a cube -
                "<<volume(10)<<"\n";
```

```
        count<<"Calling the volume( ) function for computing the volume of a
                cylinder-"<<volume(2.5, 8)<<"\n";
        count<<"Calling the volume( ) function for computing the volume of a
                rectangular box-"<<volume(100L, 75, 15);
        return 0;
}
// Function definitions
int volume(int a)       // cube
{
        return(a*a*a);
}
double volume(double r, int h)          // cylinder
{
        return(3.14519*r*r*h);
}
long volume(long l, int b, int h)       // rectangular box
{
        return(l*b*h);
}
```

The output of the above program would be

Calling the volume( ) function for computing the volume of a cube - 1000

Calling the volume( ) function for computing the volume of a cylinder - 157.26

Calling the volume( ) function for computing the volume of a rectangular box - 112500

3.6 FRIEND AND VIRTUAL FUNCTIONS

C++ introduces two new types of functions, namely, friend function and virtual function. They are basically introduced to handle some specific tasks related to class objects. Therefore discussions on these functions have been reserved until after the class objects are discussed.

CLASSES AND OBJECTS

3.7 Introduction

The most important feature of C++ is the "class". Its significance is highlighted by the fact that Stroustrup initially gave the name "C with classes" to his new language. A class is an extension of the idea of structure used in C. It is the way of creating and implementing a user defined data type.

3.8 C STRUCTURES REVISITED

We know that one of the unique features of the C language is structures. They provide a method for packing together data of different types. A structure is a convenient tool for handling a group of logically related data items. It 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. These fields are known as structure members or elements. The identifier student, which is referred to as structure name or structure tag, can be used to create variables of type student.

Example:

```
struct student A;        // C declaration
```

A is a variable of type student and has three member variables as defined by the template. Member variables can be accessed using the dot or period operator as follows:

```
strcpy(A.name, "John");

A.roll_number = 999;

A.total_marks = 595.5;

Final_total = A.total_marks+5;
```

Structures can have arrays, pointers or structures as members.

LIMITATIONS OF C STRUCTURE

The standard C does not allow the struct data type to be treated like built-in types. For example, consider the following structure:

```
struct complex

{

        float x;

        float y;

};

struct complex c1, c2, c3;
```

The complex numbers c1, c2 and c3 can easily be assigned values using the dot operator, but we cannot add two complex numbers or subtract one from the other. For example,

```
c3=c1 +c2;
```

is illegal in C.

Another important limitation of C structures is that they do not permit data hiding. Structure members can be directly accessed by the structure variables by any function anywhere in their scope. In other words, the structure members are public members.

EXTENSION TO STRUCTURES

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main principles of OOP. Inheritance, a mechanism by which one type can inherit characteristics from other types, is also supported by C++.

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. In other words, 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
```

Remember, this is an error in C.

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.

3.9 SPECIFYING A CLASS

A class is a way to bind the data and its associated functions together. It allows the data (and function) 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. Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is:

```
class class_name
{
        private:
```

```
        variable declarations;
        function declarations;
    public:
        variable declarations;
        function declarations;
};
```

The class declaration is similar to a struct declaration. The keyword class specifies, that what follows is an abstract data of type class_name. The body of a class is enclosed within braces and terminated by a semicolon. 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. Note that these keywords are followed by a colon.

The class members that have been declared as private can be accessed only from within the class. On the other hand, the public members can be accessed from outside the class also. The use of the keyword private is optional. By default, the members of the class are private. If both the labels are missing, then, by default, all the members are private. Such a class is completely hidden from the outside word and does not serve any purpose.

A Simple Class Example

A typical class declaration would look like:

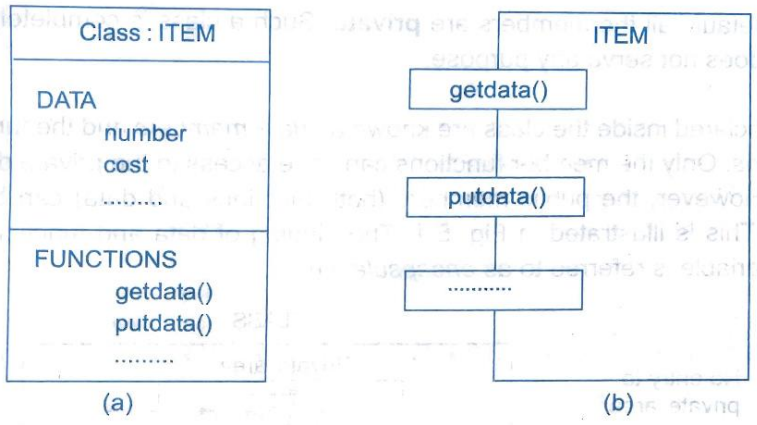
```
class item
{
        int number;             // variables declaration
        float cost;             // privvate by default
public:
        void getdata(int a, float b);    // functions declaration
```

```
    void putdata(void);          // using prototype
};                               // ends with semicolon
```

We usually give a class some meaningful name, such as item. This name now becomes a new type identifier that can be used to declare instances of that class type. The class item contains two data members and two function members. The data members are private by default while both the functions are public by declaration. The function getdata( ) can be used to assign values to the member variables number and cost, and putdata( ) for displaying their values. These functions provide the only access to the data members from outside the class. This means that the data cannot be accessed by any function that is not a member of the class item. Note that the functions are declared but not defined.

CREATING 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 variables). For example,




```
item x;          // memory for x is created
```

creates a variable x of type item. In C++, the class variables are known as objects. Therefore, x is called an object of type item. We may also declare more than one object in one statement. Example:

```
item x, y, z;
```

The declaration of an object is similar to that of a variable of any basic type. 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 we do in the case of structures. That is to say, the definition

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

would create the objects x, y and z of type item.

ACCESSING CLASS MEMBERS

The main( ) cannot contain statements that access number and cost directly. 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);
```

is valid and assigns the value 100 to number and 75.5 to cost of the object x by implementing the getdata( ) function. The assignments occur in the actual function.

```
x.putdata( );
```

would display the values of data members. The statement like

```
getdata(100, 75.5);
```

has no meaning. Similarly, the statement

```
x.number = 100;
```

is also illegal. Although x is an object of the type item to which number belongs, the number can be accessed only through a member function and not by the object directly.

It may be recalled that objects communicate by sending and receiving messages. This is achieved through the member function. For example,

x.putdata( );

sends a message to the object x requesting it to display its contents.

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
        ........
        ........
```

Note : The use of data in this manner defeats the very idea of data hiding and therefore should be avoided.

3.10 DEFINING MEMBER FUNCTION

Member functions can be defined in two places:

* Outside the class definition

* Inside the class definition

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases.

Outside the Class Definition

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'lable' tells the compiler which class the function belongs to. The general form of a member function definition is:

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

The membership lable 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.

For instance, consider the member functions getdata( ) and putdata( ) as discussed above. They may be coded as follows:

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

```
    cout << "Number :" << number << "\n";
    cout << "Cost  :" << cost << "\n";
}
```

Since these functions do not return any value, their return-type is void. Function arguments are declared using the ANSI prototype.

The member functions have some special characteristics that are often used in the program development. These characteristics are :

* Several different classes can use the same function name. The 'membership lable' will resolve their scope.

* Member functions can access the private data of the class. A nonmember function cannot do so.

* A member function can call another member function directly, without using the dot 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 number;
    float cost;
public:
    void getdata(int a, float b);    // declaration
        // inline function
    void putdata(void)               // definition inside the class
    {
```

```
        cout << number << "\n";
        cout << cost << "\n";
    }
};
```

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.

3.11 A C++ PROGRAM WITH CLASS

Program 5.1 Class Implementation

```
#include <iostream>
using namespace std;
class item
{
    int number;     // private by default
    float cost;     // private by default
public:
    void getdata(int a, float b);    // prototype declaration to be defined
    // Function defined inside class
    void putdata(void)
    {
        cout << "number :" << number << "\n";
        cout << "cost :" << cost << "\n";
    }
```

```
};
// ..... Member Function Definition .....
void item :: getdata(int a, float b)      // use membership label
{
        number = a;    // private variables
        cost = b;          // directly used
}
// ..... Main Program .....
int main( )
{
        item x;            // create object x
        cout << "\nobject x" << "\n";
        x.getdata(100, 299.95);          // call member function
        x.putdata( );                    // call member function
        item y;            // create another object
        cout << "\nobject y" << "\n";
        y.getdata(200, 175.50);
        y.putdata( );
        return 0;
}
```

This program features the class item. This class contains two private variables and two public functions. The member function getdata( ) which has been defined outside the class supplies values to both the variables. The use of statements such as

```
number = a;
```

in the function definition of getdata( ). This shows that the member functions can have direct access to private data items.

The member function putdata( ) has been defined inside the class and therefore behaves like an inline function. This function displays the values of the private variables number and cost.

The program creates two objects, x and y in two different statements. This can be combined in one statement.

int x, y; // creates a list of objects

The output of the above program is

object x

number : 100

cost : 299.95

object y

number : 200

cost : 175.5

For the sake of illustration we have shown one member function as inline and the other as an 'external' member function. Both can be defined as **inline** or external functions.

UNIT IV

MEMBER FUNCTIONS

4.1 NESTING OF MEMBER FUNCTIONS

We just discussed that a member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is knowm as nesting of member functions. Program 5.2 illustrates this feature.

Program 4.1 Nesting of Member functions

```
# include <iostream>
# include <conio.h>
# include <string>
using namespace std;
class binary
{
        string s;
        public:
        void read (void)
        {
           cout<< "Enter a binary number :";
           cin>>s;
        }
        void chk_bin (void)
        {
          for (int i = 0; i < s. length ( ) ; i++)
          {
                if (s.at (i) ! = '0' && s.at (i) ! = '1')
                {
                cout<<" \ nIncorrect binary number format... the program will
        quit ";
                getch ( ) ;
```

```
                exit ( 0 ) ;
            }
        }
    }
    void ones ( void )
    {
    chk_bin ( ) ;    // calling member function
    for (int i = 0; i < s.length ( ) ; i++
        {
            if (s.at (i) =='0')
                s.at (i) = '1';
            else
                s.at (i) = '0' ;
        }
    }
    void displayones (void)
    {
        ones ( ) ; // calling member function
        cout<<"\nThe 1's compliment of the above binary number is : "<<s;
    }
} ;
int main ( )
{
    binary b;
    b.read ( );
    b.displayones ( );
    getch ( ) ;
    return 0 ;
}
```

The output of program 4.1 would be :

OUTPUT 1 :

Enter a binary number : 110101

The 1's compliment of the above binary number is : 001010

OUTPUT 2:

Enter a binary number : 1101210

Incorrect binary number format ... the program will quit

NOTE : The above program uses the built-in-class, string for storing the binary number.

4.2 PRIVATE MEMBER FUNCTIONS

Although it is normal practice to place all the data items in a private section and all the functions in public, some situations may require certain functions to be hidden (like private data) from the outside calls. Tasks such as deleting an account in a customer file, or providing increment to an employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using a dot operator. Consider a class as defined below:

```
class simple
{
        int m;
        void read (void) ; // private member function
    public :
        void update (void) ;
        void write (void);
} ;
```

If s1 is an object of sample, then

```
s1.read ( ) ;          // won't work; objects cannot access
                       // private members
```

is illegal. However, the function read ( ) can be called by the function update( ) to update the value of m.

```
void sample : : update (void)
{
        read ( );          // simple call; no object used
}
```

4.3 ARRAYS WITHIN A CLASS

The arrays can be used as member variable in a class. The following class definition is valid.

```
const int size = 10;     // provides value for array size
class array
{
        int a [size] ;    // 'a' is int type array
   public:
        void setval (void) ;
        void display (void) ;
} ;
```

The array variable a[ ] declared as a private member of the class array can be used in the functions, like any other array variable. We can perform any operations on it. For instance, in the above class definition, the member function setval( ) sets the values of elements of the array a[ ], and display( ) function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

Program 4.2 Processing shopping List

```
#include <iostream>
using namespace std;
const m=50;
class ITEMS
{
        int itemCode[m];
        float itemPrice[m];
        int count;
public:
        void CNT(void) (count = 0;)  // initializes count to 0
```

```
        void getitem(void);
        void displaySum(void);
        void remove(void);
        void displayItem(void);
};
//============================
VOID ITEMS :: getitem(void)          // assign values to data
{                                    // members of item
        cout << "Enter item code :";
        cin >> itemCode(count);
        cout << "Enter item cost :";
        cin >> itemPrice(count);
        count++;
}
void ITEMS :: displaySum(void)               // display total value of all items
{
        float sum = 0;
        for (int i=0; i<count; i++)
        sum = sum+ itemPrice[i];
        cout << "\nTotal value :" << sum << "\n";
}
void ITEMS :: remove(void)           // delete a specified item
{
        int a;
        cout << "Enter item code : ";
        cin >> a;
        for (int i=0; i<count; i++)
        if (itemCode[i] == a)
           itemPrice[i] = 0;
}
void ITEMS :: displayItems(void)             // displaying items
{
        cout << "\nCode Price \n";
```

```
        for (int i=0; i<count; i++)
        {
                cout <<"\n" << itemCode[i];
                cout << " " << itemPrice [i];
        }
        cout << "\n";
}
// ============================
int main( )
{
        ITEMS order;
        order.CNT( );
        int x;
        do              // do ..... while loop
        {
                cout << "\n You can do the following;"
                        << "Enter appropriate number \n";
                cout << "\n1: Add an item ";
                cout << "\n2 : Display total value";
                cout << "\n3 : Delete an item";
                cout << "\n4 : Display all items";
                cout << "\n5 : Quit";
                cout << "\n\nWhat is your option?";
                cin >> x;
                switch(x)
                {
                        case 1 : order.getitem( ); break;
                        case 2: order.displaySum( ); break;
                        case 3 : order.remove( ); break;
                        case 4 : order.displayItems( ); break;
                        case 5 : break;
                        default : cout << "Error in input; try again\n";
                }
```

```
        } while(x!=5);                  // do ..... while ends
    return 0;
    }
```

The output of the above program would be

You can do the following; Enter appropriate number

1 : Add an item

2 : Display total value

3 : Delete an item

4 : Display all items

5 : Quit

What is your option ? 1

Enter item code : 111

Enter item cost : 100

You can do the following; Enter appropriate number

1 : Add an item

2 : Display total value

3 : Delete an item

4 : Display all items

5 : Quit

What is your option ? 1

Enter item code : 222

Enter item cost : 200

You can do the following; Enter appropriate number

1 : Add an item

2 : Display total value

3 : Delete an item

4 : Display all items

5 : Quit

What is your option ? 2

Total value : 600

You can do the following; Enter appropriate number

1 : Add an item

2 : Display total value

3 : Delete an item

4 : Display all items

5 : Quit

What is your option ? 3

Enter item code : 222

You can do the following; Enter appropriate number

1 : Add an item

2 : Display total value

3 : Delete an item

4 : Display all items

5 : Quit

What is your option? 4

Code Price

111 100

222 0

333 300

You can do the following; Enter appropriate number

1 : Add an item

2 : Display total value

3 : Delete an item

4 : Display all items

5 : Quit

What is your option? 5

NOTE : The program uses two arrays, namely itemCode[ ] to hold the code number of items and ItemPrice [ ] to hold the prices. A third data member count is used to keep a record of items in the list. The program uses a lot of four functions to implement the operations to be performed on the list. The statement

const int m = 50;

defines the size of the array members

The first function CNT ( ) simply sets the variable count to zero. The second function getitem ( ) gets the item code and the item price interactively and assigns them to the array members itemCode[count] and itemPrice[count]. Note that inside this function count is incremented after the assignment operation is over. The function displaySum( ) first evaluates the total value of the order and then prints the value. The fourth function remove() deletes a given item from the list. It uses the item code to locate it in the list and sets the price to zero indicating that the item is not 'active' in the list. Lastly, the function displayItems ( ) displays all the items in the list.

The program implements all the tasks using a menu-based user interface.

4.4 MEMORY ALLOCATION FOR OBJECTS

We have stated that the memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true. Actually, the member functions 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. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects. This is shown in figure 4.1

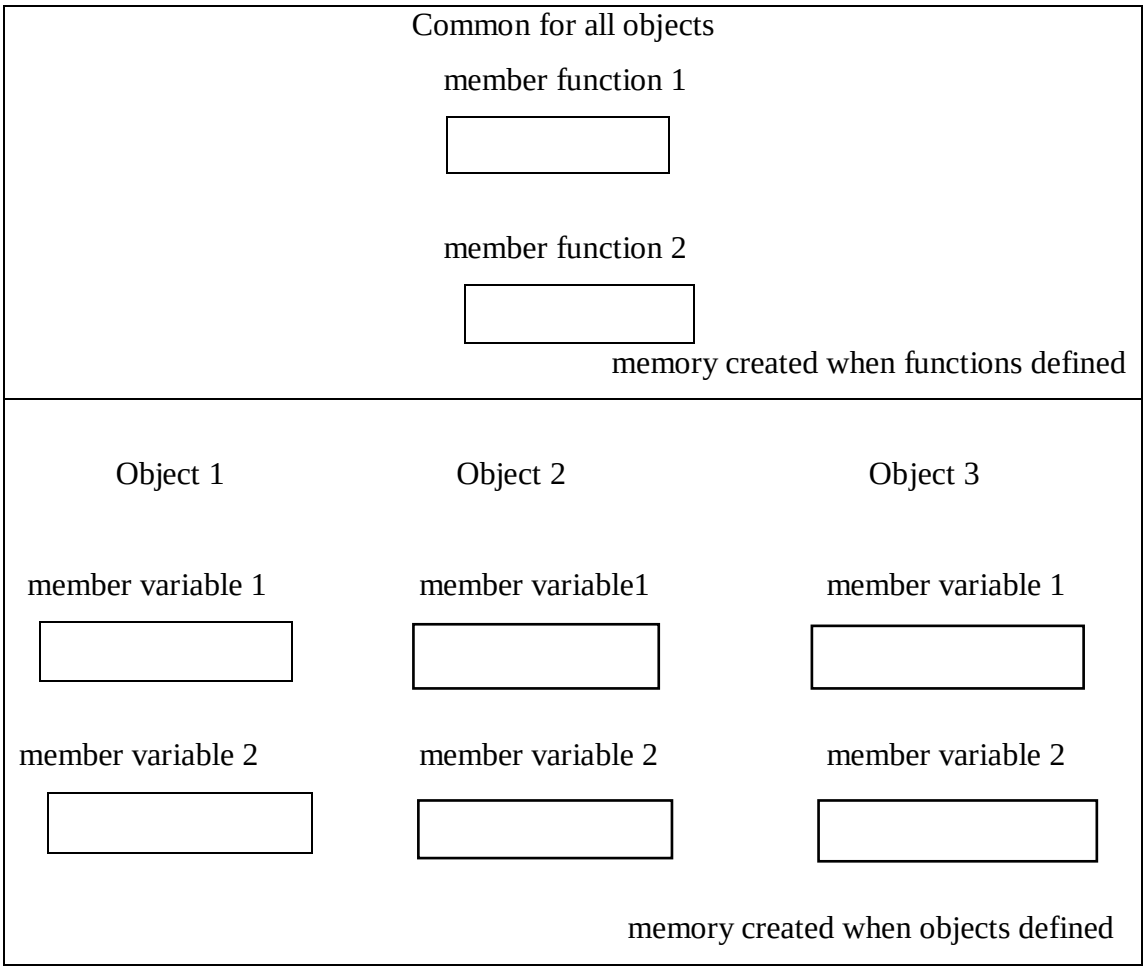


Figure 4.1 Object of memory

4.5 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 :

* It is initialized to zero when the first object of its class is created. No other initialization is permitted.

* 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.

* 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. For example, a static data member can be used as a counter that records the occurrences of all the objects. The program below illustrates the use of a static data member.

Program 4.3 Static Class Member

```
#include <iostream>
using namespace std;
class item
{
        static int count;
        int number;
public:
        void getdata(int a)
        {
                number=a;
                count +1;
        }
        void getcount(void)
        {
                count << "count: ";
                count << count << "\n";
        }
};
int item :: count
int main( )
```

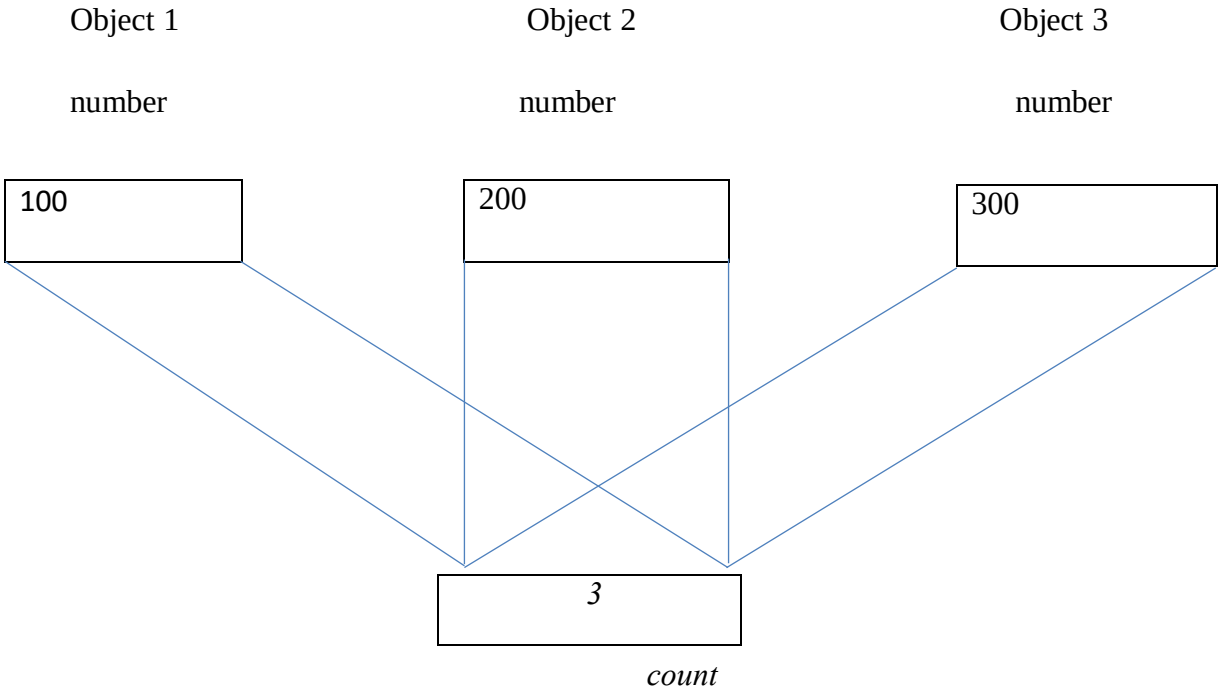
```
{
        item a, b, c;           // count is initialized to zero
        a.getcount( );          // display count
        b.getcount( );
        c.getcount( );
        a.getcount(100);        // getting data into object a
        b.getcount(200);        // getting data into object b
        c.getcount(300);        // getting data into object c

        count << "After reading data" << "\n";
        a.getcount( );          // display count
        b.getcount( );
        c.getcount( );
        return 0;
}
```

The output of the above program would be

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

The static variable count is initialized to zero when the objects are created. The count is incremented whenever the data is read into an object. Since the data is read into object three times, the variable count is incremented three times. Because there is only one copy of count shared by all the three objects, all the three output statements cause the value 3 to be displayed.



(Common to all three objects)

Static variables are like non-inline member functions as they are declared in a class declaration and defined in the source file. While defining a static variable, some initial value can also be assigned to the variable. For instance, the following definition gives count the initial value 10.

int item : : count = 10 ;

4.6 STATIC MEMBER FUNCTIONS

Like static member variable we can also have 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;

Program 4.4 illustrates the implementation of these characteristics. 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.

Program 4.4 STATIC MEMBER FUNCTION

```
# include <iostream>
using namespace std;
class test
{
        int code;
        static int count;                    //static number 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;
}
```

The output of Program 4.4 would be :

```
count : 2

count : 3

object number : 1

object number : 2

object number : 3
```

NOTE : Note that the statement

```
code = ++count;
```

is executed wherever setcode( ) function is invoked and the current value of count is assigned to code. Since each object has its own copy of code, the value contained in code represents a unique number of its object.

Remember, the following function definition will not work:

```
static void showcount ( )
{
    cout << code;                    // code is not static
}
```

4.7 Arrays of objects

We know that an array can be of any data type including struct. Similarly, 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 identifier employee is a user-defined data type and can be used to create objects that relate to different categories of the employees. Example:

```
employee manager [3] ;          // array of manager
employee foreman [15] ;         // array of foreman
employee worker [75] ;          // array of worker
```

The array manager contains three objects (managers), namely, manager [0], manager [1] and manager [2], of type employee class. Similarly, the foreman array contains 15 objects (foreman) and the worker array contains 75 objects(workers).

Since an array of objects behaves like any other array, we can use the usual array-accessing methods to access individual elements, and then the dot member operator to access the member functions. For example, the statement

```
manager [i]. putdata ( ) ;
```

will display the data of the ith element of the array manager. That is, this statement requests the object manager [i] to invoke the member function putdata( ).

An array of objects is stored inside the memory in the same way as a multi-dimensional array. The array manager is represented in Fig. 5.5. Note that only the space for data items of the objects is created. Member functions are stored separately and will be used by all the objects.

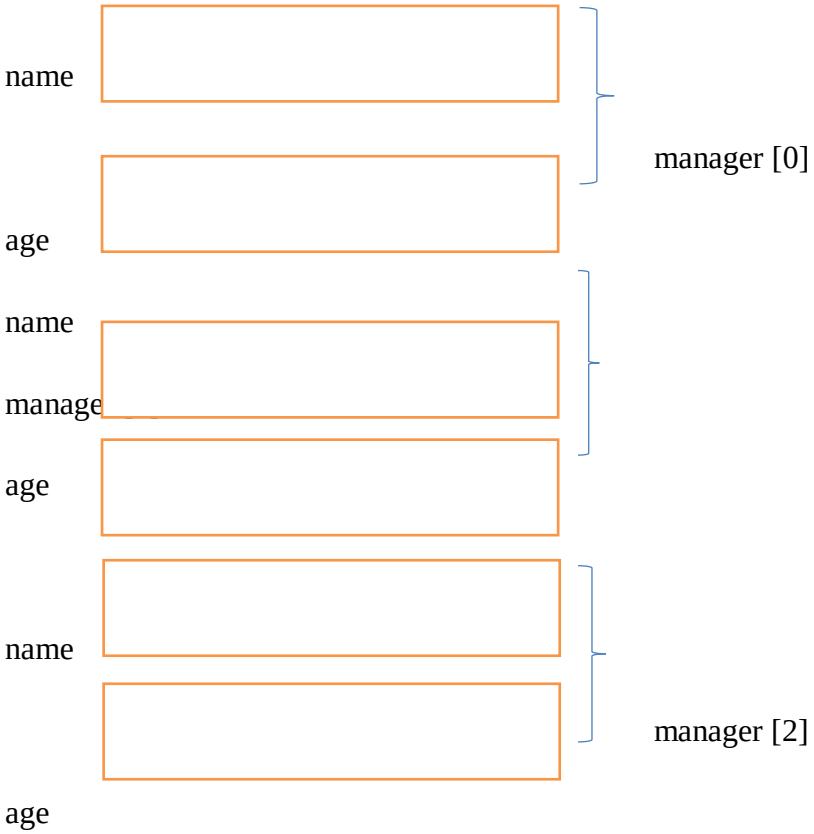


Fig 4.2 Storage of data items of an object array

Program 4.5 illustrates the use of object arrays

Program 4.5 Array of objects

```
#include <iostream>
using namespace std;
class employee
{
        char name [30];         // string as class member
        float age;
   public;
        void getdata (void);
        void putdata (void);
};
void employee : : getdata (void)
{
        cout << "Enter name: ";
```

```
        cin >> name;
        cout << "Enter age: ";
        cin >> age;
}
void employee : : putdata (void)
{
        cout << "Name: " << name << "\n";
        cout << "Age:" << age << "\n";
}
const int size = 3;
int main ( )
{
        employee manager [size];
        for (int i = 0; i < size; i ++)
        {
                cout << "\nDetails of manager" << i + 1 << "\n";
                manager [i].getdata ( );
        }
        cout << "\n";
        for (i = 0; i < size; i++)
        {
                cout << "\nManager" << i + 1 << "\n";
                manager [i].putdata ( );
        }
        return 0;
}
```

This being an interactive program, the input data and the program output are shown below:

```
Interactive input
        Details of manager 1
        Enter name : xxx
        Enter age : 45
```

Details of manager 2

Enter name : yyy

Enter age : 37

Details of manager 3

Enter name : zzz

Enter age : 50

Program output

Manager 1

Name : xxx

Age : 45

Manager 2

Name : yyy

Age : 37

Manager 3

Name : zzz

Age : 50

4.8 OBJECTS AS FUNCTION ARGUMENTS

Like any other data type, an object may be used as a function argument. This can be done in two ways:

* A copy of the entire object is passed to thee 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.

Program 4.6 illustrates the use of objects as function arguments. It performs the addition of time in the hour and minutes format.

Program 4.6 Objects as Arguments

```
#include <iostream>
using namespace std;
class time
{
        int hours;
        int minutes;
   public:
        void gettime ( int h, int m)
        (hours = h; minutes = m;)
        void puttime (void)
        {
                cout << hours << "hours and ";
                cout << minutes << "minutes " << "\n";
        }
        void sum (time, time); // declaration with objects as arguments
};
        void time : : sum (time t1, time t2)     // t1, t2 are objects
{
        minutes = t1.minutes + t2.minutes;
        hours = minutes / 60;
        minutes = minutes%60
        hours = hours + t1.hours + t2.hours;
}
int main ( )
{
        time T1, T2, T3;
        T1.gettime (2, 45);      // get T1
```

```
    T2.gettime (3, 30);     // get T2
    T3. sum (T1, T2);       // T3 = T1 + T2
    cout << "T1 = ";        T1.puttime ( );        // display T1
    cout << "T2 =";         T2.puttime ( );        // display T2
    cout << "T3=";          T3.puttime ( );        // display T3
    return 0;
}
```

The output of Program 4.6 would be :

T1 = 2 hours and 45 minutes

T2 = 3 hours and 30 minutes

T3 = 6 hours and 15 minutes

NOTE : Since the member function sum ( ) is invoked by the object T3, with the objects T1 and T2 as arguments, it can directly access the hours and minute variables of T3. But, the members of T1 and T2 can be accessed only by using the dot operator (like T1.hours and T1.minutes). Therefore, inside the function sum( ), the variables hours and minutes refer to T3, T1.hours and T1.minutes refer to T1, and T2.hours and T2.minutes refer to T2.

Figure 4.3 illustrates how the members are accessed inside the function sum( ).

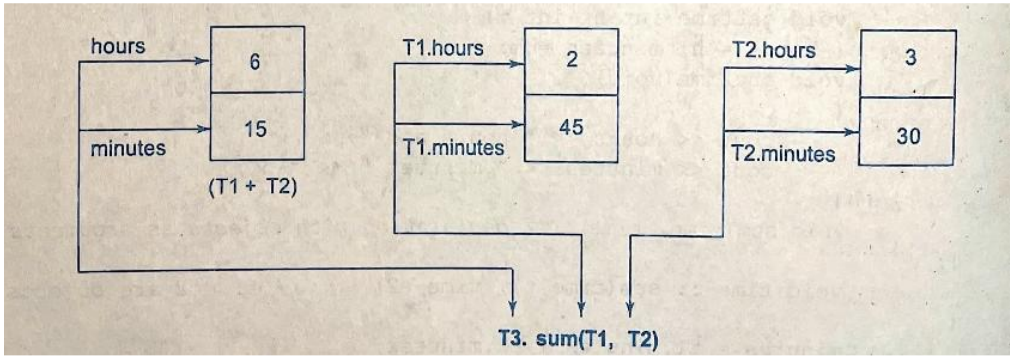


Fig. 4.3 Accessing members of objects within a called function

An object can also be passed as an argument to a nonmember function. However, such functions can have access to the public member functions only through the objects passed as arguments to it. These functions cannot have access to the private data members.

4.9 FRIENDLY FUNCTIONS

We have been emphasizing throughout this chapter that the private numbers cannot be accessed from outside the class. That is, 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 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 proceeded by the keyword **friend**. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword friend or the scope operator... 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 affecting its meaning.

* Usually, it has the object as arguments.

The friend functions are often used in operator overloading which will be discussed later.

Program 4.7 illustrates the use of a friend function.

Program 4.7 Friendly Function

```
#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 = "  << man (X)   << "\n";
```

```
    return 0;
}
```

The output of Program 4.7 would be:

Mean value = 32.5

NOTE : The friend function accesses the class variables a and b by using the dot operator and the object passed to it. The function call mean(X) passes the object X by value to the friend function.

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
};
```

Program 4.8 demonstrates how friend functions work as a bridge between the classes.

Program 4.8 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 output of Program 4.8 would be :

```
20
```

NOTE: 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.

As pointed out earlier, a friend function can be called by reference. In this case, local copies of the objects are not made. Instead a pointer to the address of the object is passed and the called function directly works on the actual object used in the call.

This method can be used to alter the values of the private numbers of a class. Remember altering the values of private numbers is against the basic principles of data hiding. It should be used only when absolutely necessary.

Program 4.9 shows how to use a common function to exchange the private values of two classes. The function is called by reference.

Program 4.9 Swapping Private Data of Classes

```
#include <iostream>
using namespace std;
```

```
class class_2;
class class_1
{
        int value1;
    public:
        void indata (int a) {value1 = a ;}
        void display (void) {cout << value1 << "\n";}
        friend void exchange (class_1 &, class_2 &);
};
class class_2
{
        int value2;
    public:
        void indata (int a) {value2 = a;}
        void display(void) {cout << value2 << "\n";}
        friend void exchange (class_1 &, class_2 &);
};
void exchange (class_1 & x, class_2 & y)
{
        int temp = x.value1;
        x.value1 = y.value2;
        y.value2 = temp;
}
int main( )
{
        class_1 C1;
        class_2 C2;

        C1.indata (100);
        C2.indata (200);

        cout << "Values before exchange" << "\n";
        C1.display ( );
```

```
        C2.display ( );
        exchange {C1, C2);    // swapping

        cout << "Values after exchange " << "\n";
        C1.display ( );
        C2.display ( );

        return 0;
}
```

The objects x and y are aliases of C1 and C2 respectively. The statements

```
int temp = x.value1
x.value1 = y.value2;
y.value2 = temp;
```

directly modify the values of value1 and value2 declared in class_1 and class_2.

The output of Program 4.9 would be:

```
Values before exchange

100
200
Values after exchange
200

100
```

4.10 RETURNING OBJECTS

A function cannot only receive objects as arguments but also can return them. The example in Program 4.10 illustrates how an object can be created (within a function) and returned to another function.

Program 4.10 Returning Objects

```
# include <iostream>
```

```
# include <conio.h>
using namespace std;
class matrix
{
        int m [3] [3];
public:
        void read (void)
        {
                cout << "Enter the elements of the 3x3 matrix:\n";
                int i, i;
                for (i = 0; i < 3; i++)
                        for (j = 0; j < 3; j++)
                        {
                                cout << "m["<<i<<"] ["<<j<<"] = ";
                                cin >> m [i] [j];
                        }
        }
        void display (void)
        {
                int i, j;
                for (i = 0; i < 3; i++)
                {
                        cout << "\n";
                        for (j = 0; j < 3 ; j++)
                        {
                                cout << m [i] [j] << "\t";
                        }
                }
        }
        friend matrix trans (matrix);
};
matrix trans (matrix m1);
{
```

```
        matrix m2;      // creating an object
        int i, j;

        for (i = 0; i < 3; i++)
                for (j = 0; j < 3; j++)
                        m2.m [i] [j] = m1.m[j] [i];
        return (m2);    // returning an object
}
int main ( )
{
        matrix mat1, mat2;

        mat1. read ( );
        cout << "\nYou entered the following matrix:";
        mat1.display ( );

        mat2 = trans (mat1);
        cout << "\nTransposed matrix:";
        mat2.display ( );

        getch ( );
        return 0;
}
```

Upon execution, Program 4.10 would generate the following output:

```
Enter the elements of the 3x3 matrix:

m [0] [0] = 1
m [0] [1] = 2
m [0] [2] = 3
m [1] [0] = 4
m [1] [1] = 5
m [1] [2] = 6
```

m [2] [0] = 7
m [2] [1] = 8
m [2] [2] = 9

You entered the following matrix:

1 2 3
4 5 6
7 8 9

Transposed matrix:

1 4 7
2 5 8
3 6 9

The program finds the transpose of a given 3x3 matrix and stores it in a new matrix object. The display member function displays the matrix elements.

4.11 CONST MEMBER FUNCTIONS

If a member function does not alter any data in the class, then we may declare it as a const member funcction as follows:

void mul (int, int) const;

double get_balance ( ) const;

The qualifier const is appended to the function prototypes (in both declaration and definition). The compiler will generate an error message if such functions try to alter the data values.

4.12 POINTERS TO MEMBERS

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a "fully qualified" class

member name. A class member pointer can be declared using the operator : :* with the class name. For example, given the class

```
class A
{
        private:
                int m;
        public:
                void show ( );
};
```

We can define a pointer to the member m as follows:

```
int A : :* ip = &A : : m;
```

The *ip* pointer created thus acts like a class in that it must be invoked with a class object. In the statement above, the phrase A: :* means "pointer-to-member of A class". The phrase &A: : m means the "address of the m member of A class".

Remember, the following statement is not valid.

```
int *ip = &m;                    // won't work
```

This is because m is not simply an int type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer ip can now be used to access the member m inside member functions (or friend functions). Let us assume that a is an object of A declared in the member function. we can access m using the pointer ip as follows:

```
cout << a.*ip;                   // display

cout << a.m;                     // same as above
```

Now, look at the following code:

```
ap = *a;                         // ap is pointer to object a

cout << ap -> *ip;               // display m
```

```
cout << ap -> m;                // same as above
```

The dereferencing operator ->* is used to access a member when we use pointers to both the object and the member. The dereferencing operator.* is used when the object itself is used with the member pointer. Note that *ip is used like a member name.

We can also design pointers to member functions which, then, can be invoked using the dereferencing operators in the main as shown below:

```
(object-name .* pointer-to-member function) (10);
```

```
(pointer-to-object ->* pointer-to-member function) (10)
```

The precedence of ( ) is higher than that of .* and ->*, so the parenthesis are necessary.

Program 4.11 illustrates the use of dereferencing operators to access the class members.

Program 4.11 Dereferencing Operators

```
#include <iostream>
using namespace std;
class M
{
        int x;
        int y;
    public:
        void set__xy (int a, int b)
        {
                x = a;
                y = b;
        }
        friend int sum (M m);
};
int sum (M m)
{
        int M : :* px = &M : : x;
        int M : :* py = &M : : y;
```

```
        M *pm = &m;
        int S = m.*px + pm ->*py;
        return S;
}
int main ( )
{
        M n;
        void (M : : *pf) (int, int) = &M : : set_xy;
        (n.*pf) (10, 20);
        cout << "SUM = " << sum (n) << "\n";

        M *op = &n;
        (op-> *pf) (30, 40);
        cout << "Sum = " << sum (n) << "\n";

        return 0;
}
```

The output of Program 4.11 would be:

```
sum = 30

sum = 70
```

4.13 LOCAL CLASSES

Classes can be defined and used inside a function or a block. Such classes are called local classes. Examples:

```
void test (int a)              // function
{
        . . . . .
        . . . . .
        class student          // local class
        {
            . . . . .
            . . . . .          //class definition
```

```
        . . . . .
    };
    . . . . .
    . . . . .
    student s1 (a);          // create student object
    . . . . .                // use student object
}
```

Local classes can use global variables (declared above the function) and static variables declared inside the function but cannot use automatic local variables. The global variables should be used with a scope operator (: :).

There are some restrictions in constructing local classes. They cannot have static data members and member functions must be defined inside the local classes. Enclosing function cannot access the private members of a local class. However, we can achieve this by declaring the enclosing function as a friend.

UNIT V

CONSTRUCTORS AND DESTRUCTORS,

OPERATOR OVERLOADING & TYPE CONVERSIONS

5.1 INTRODUCTION

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as putdata( ) and setvalue( ) to provide initial values to the private member variables. For example, the following statement

A.input ( );

invokes the member function input( ), which assigns the initial values to the data items of object A. Similarly, the statement

x.getdata (100,299.95);

passes the initial values as argumets to the function getdata( ), where these values are assigned to the private variables of object x. All these 'function call' statements are used with the appropriate objects that have already been created. These functions cannot be used to initialize the member variables at the time of creation of their objects.

Providing the initial values as described above does not conform with the philosophy of C++ language. We stated earlier that, one of the aims of C++ is to create user-defined data types such as class, that behave very similar to the built-in types. This means that we should be able to initialize a class type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example,

int m = 20;

float x = 5.75;

are valid initialization statements for basic data types.

Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variable. But it has not happened with objects we have so far studied. It is therefore clear that some more features of classes need to be explored that would

enable us to initialize the objects when they are created and destroy them when their presence is no longer necessary.

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as automatic initialization of objects. It also provides another member function called the distractor that destroys the objects when they are no longer required.

5.2 CONSTRUCTORS

A constructor is a special member function , whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```
// classwith aconstructor
class integer
{
        int m, n;
    public:
        integer (void);                    // constructor declared
        .....
        .....
};
integer : : integer (void)                 // constructor defined
{
        m = 0; n=0;
}
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

```
integer int1;                              // object int1 created
```

not only creates the object int1 of type integer, but also initialize its data members m and n to zero. There is no need to write any statement to invoke the constructor function (as we do

with the normal member functions). If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately. This would be very inconvenient, if there are a large number of objects.

A constructor that accepts no parameters is called the default constructor. The default constructor for class A is A::A(). If no such constructor is defined, then the compiler supplies a default constructor. Therefore, a statement such as

A a;

invokes the default constructor of the compiler to create object a.

The constructor functions have some special characteristics. These are:

*They should be declared in the public section.

*They are invoked automatically when the objects are created.

*They do not have return types, not even void and therefore, and they cannot return values.

*They cannot be inherited, though a derived class can call the base class constructor.

*Like other C++ functions, they can have default arguments.

* Constructors cannot be virtual.

*We cannot refer to this to their addresses.

*An object with the constructor (or destructor) cannot be used as a member of a union.

*They make 'implicit calls' to the operators new and delete when member allocation is required.

Remember, when a constructor is declared for a class, initialization of the class objects become mandatory.

5.3 PARAMETERIZED CONSTRUCTORS

The constructor integer(), defined above, initializes the data members of all the objects to zero. However, in practice, it may be necessary to initialize the various data elements of

different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the object are created. The constructors that can take arguments are called parameterized constructors.

The constructor integer( ) may be modified to take arguments as shown below:

```
class integer
{
  int m, n;
    public:
        integer (int x, int );                    // parameterized constructor
        .....
        .....
};
integer : : integer (int x, int y)
{
        m = x; n=y;
}
```

When a constructor has been parameterized, the object declaration statement such as,

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

* By calling the constructor explicitly
* By calling the constructor implicitly.

The following declaration illustrates the first method:

```
integer int1 = integer (0,100);          // explicit call
```

This statement creates an integer object int1 and passes the values, 0 and 100 to it. The second is implemented as follows:

```
integer int1(,100);                      // implicit call
```

This method, sometimes called the shorthand method, is used very often as is shorter, looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor. Program 5.1 demonstrates the passing of arguments to the constructor functions.

Program 5.1 Class with Constructors

```
#include <iostream>

using namespace std;

class integer
{
        int m, n;
    public :
        integer (int, int);                    // constructor declared

        void display (void)
        {
            cout << " m = " << m << "\n";
            cout << " n = " << n << "\n";
        }
};
integer : : integer (int x, int y)             // constructor defined
{
        m = x; n = y;
}
int main ( )
{
        integer int1 (0,100);   // constructor called implicitly
        integer int2 = integer (25, 75);       // constructor called explicitly
        cout << "\nOBJECT1" << "\n";
        int1.display ( );
        cout << "\nOBJECT2" << "\n";
        int2.display ( );
```

```
        return 0;
}
```

The output of Program 5.1 would be :

```
OBJECT1
m = 0
n = 100

OBJECT2
m = 25
n = 75
```

The constructor functions can also be defined as inline functions. Example:

```
class integer
{
        int m, n;
    public:
        integer (int x,int y)      // Inline constructor
        {
          m = x; y = n;
        }
        .....
        .....
};
```

The parameters of a constructor can be of any type except that of the class to which it belongs. For example,

```
class A
{
        .....
        .....
  public :
        A (A);
};
```

is illegal.

However, a constructor can accept a reference to its own class as a parameter. Thus, the statement

```
Class A
{
        .....
        .....
public:
        A (A&);
};
```

is valid. In such cases, the constructor is called the copy constructor.

5.4 MULTIPLE CONSTRUCTORS IN A CLASS

So far used two kinds of constructors. They are:

```
integer ( );                        // No arguments

integer (int, int);                 // Two arguments
```

In the first case, the constructor itself supplies the data values and no values are passes by the calling program. In the second case, the function call passes the appropriate values from main( ). C++ permits as to use both these constructors in the same class. For example, we would define a class as follows:

```
class integer
{
        int m, n;
    public:
        integer ( ) (m=0;n=0; }             // constructor 1
        integer (inta, int b)

        {m = a; n=b;)                       // constructor 2

        integer (integer & i)

        { m = i.m; n= i.n;)                 // constructor 3
```

```
};
```

This declares three constructors for an **integer** object. The first constructor receives no arguments, the second receives two integer arguments and the third receives one integer object as an argument. For example, that declaration

```
integer I1;
```

would automatically invoke the first constructor and both m and n of I1 t0 zero. The statement

```
integer I2(20,40) ;
```

would call the second constructor which will initialize the data members m and n of 12 to 20 and 40 respectively. Finally, the statement

```
integer I3 (I2) ;
```

would invoke the third constructor which copies the values of I2into I3. In other words, it said the value of every data element of 13 to the value of the corresponding data element of 12. As mentioned earlier, such a constructor is called the copy constructor. The process of sharing the same name by two or more functions is referred to as function overloading. Similarly when more than one constructor function is defined in a class, we say that the constructor is overloaded.

Program 5.2 shows the use of overloaded constructors.

Program 5.2 Overloaded Constructors

```
# include <iostream>

using namespace std;

class complex

{

        float x, y;
    public :
        complex ( ) {   }                          // constructor no arg
        complex (float a)   {x = y = a;)           // constructor-one arg
```

```
        complex (float real, float imag)          // constructor-two args
        {x = real; y = imag;)

        friend complex sum (complex, complex);

        friend void show (complex) ;

};

complex sum (complex c1, complex c2)              // friend

{

        complex c3;

        c3.x = c1.x + c2.x;

        c3.y = c1.y + c2.y;

        return (c3);

}

void show (complex c)                             // friend

{

        cout << c.x << " + j " << c.y << "\n";

}
int main ( )
{
        complex A (2.7, 3.5);                     // define & initialize
        complex B (1.6);                          // define & initialize

        complex C;                                // define

        C = sum (A, B);                           // sum ( ) is a friend

        cout << "A = "; show (A);                 // show ( ) is also friend

        cout << "B = "; show (B);
```

```
        cout << "C = "; show (C);

// Another way to give initial values (second method)

        complex P, Q, R;                    // define P, Q and R

        P = complex (2.5, 3.9)                    // initialize P

        Q = complex (1.6, 2.5)              // initialize Q

        R = sum (P< Q);

        cout << "\n";

        cout << "P = "; show (P);

        cout << "Q = "; show (Q);

        cout << "R = "; show (R);

        return 0;

}
```

The output of Program 5.2 would be:

```
A = 2.7 + j3.5

B = 1.6 + j1.6

C = 4.3 + j5.1


P = 2.5 + j3.9

Q = 1.6 + j2.5

R = 4.1 + j6.4
```

Note : There are three constructors in the class complex. The first constructor, which takes no arguments, is used to create objects which are not initialized; the second, which takes one argument, is used to create objects and initialize them; and the third, which takes two

arguments, is also used to create objects and initialize them to specific values. Note that the second method of initializing values looks better.

Let us look at the first constructor again.

```
complex ( ) {   }
```

It contains the empty body and does not do anything. We just stated that this is used to create objects without any initial values. Remember, we have defined objects in the earlier examples without using such a constructor. Why do we need this constructor now? As pointed out earlier, C++ compiler has an implicit constructor which creates objects, even though it was not defined in the class.

This works fine as long as we do not use any other constructors in the class. However, once we define a constructor, we must also define the "do-nothing" implicit constructor. This constructor will not do anything and is defined just to satisfy the compiler.

5.5 CONSTRUCTORS WITH DEFAULT ARGUMENTS

It is possible to define constructors with default arguments. For example, the constructor complex ( ) can be declared as follows:

```
complex (float real, float imag = 0)
```

The default value of the argument imag is zero. Then, the statement

```
complex C(5.0) ;
```

assigns the value 5.0 to the real variable and 0.0 to imag (by default). However the statement

```
complex C(2.0, 3.0) ;
```

assigns 2.0 to real and 3.0 to imag. The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor A : : A( ) and the default argument constructor A : : A(int = 0). The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

A a;

The ambiguity is whether to 'call' **A: :A( ) or A: :A(int = 0).**

5.6 DYNAMIC INITIALIZATION OF OBJECTS

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment. Program 5.3 illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

Program 5.3 DYNAMIC INITIALIZATION OF OBJECTS

```
// Long-term fixed deposit system

# include <iostream>

using namespace std;

class Fixed_deposit

{

        long int P_amount;          // Principal amount
        int Years;                  // Period of investment
        float Rate;                 // Interest rate
        float R_value;              // Return value of amount
    public :
        Fixed_deposit ( ) {    }
        Fixed_deposit (long int p, int y, float r = 0.12);
        fixed deposit (long int p, int y, int r);
        void display (void0;
};

Fixed_deposit : : Fixed_deposit (long int p, int y, float r)
```

```
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount ;
        for (int i = 1; i <= y; i++)
                R_value = R_value * (1.0 + r);
}
Fixed_deposit : : Fixed_deposit (long int p, int y, int r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount ;
        for (int i=1; i <=y; i++)
          R_value = R_value*(1.0+float (r) /100);
}
void Fixed_deposit : : display(void)
{
        cout << "\n"
        << "Principal Amount = " << P_amount << "\n"
        << "Return value = " << R_value << "\n";
}
```

```
int main( )
{
        Fixed_deposit FD1, FD2, FD3;        // deposit created
        long int p;                         // principal amount
int y;                                      // investment period, years
float r;                                    // interest rate, decimal form
int R;                                      // interest rate, percent form
cout  << "Enter amount, period, interest rate (in percent) " << "\n";
cin  >> p >> y >> R;
FD1 = Fixed_deposit (p,y,r);
cout << "Enter amount, period, interest rate (decimal form)" << "\n";
cin >> p >> y >> r ;
FD2 = Fixed_deposit (p,y,r);
cout << "Enter amount and period" << "\n";
cin >> p >> y;
FD3 = Fixed_deposit (p,y);
cout << "\nDeposit 1";
FD1.display ( );
cout << "\nDeposit 2";
FD2.display ( );
cout << "\nDeposit 3";
FD3.display ( );
return 0;
```

}

The output of Program 5.3 would be:

Enter amount, period,interest rate (in percent)

10000 3 18

Enter amount, period, interest rate (in decimal form)

10000 2 0.18

Enter amount and period

10000 3

Deposit 1

Principal Amount = 10000

Return Value = 16430.3

Deposit 2

Principal Amount = 10000

Return Value = 16430.3

Deposit 3

Principal Amount = 10000

Return Value = 14049.3

The program uses three overloaded constructors. The parameter values to these constructors are provided at run time. The user can provide input in one of the following forms:

1. Amount, period and interest in decimal form.
2. Amount, period and interest in percent form.
3. Amount and period.

NOTE : Since the constructors are overloaded with the appropriate parameters, that one that matches the input values is invoked. For example, the second constructor is invoked for the

forms (1) and (3), and the third is invoked for the form (2). Note that, for form (3), the constructor with default argument is used. Since input to the third parameter is missing, it uses the default value for r.

5.7 COPY CONSTRUCTOR

We briefly mentioned about the copy constructor in Sec. 6.3. We used the copy constructor

integer (integer &i);

in Sec 6.4 as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare and initialize an object from another object. For example, the statement

integer I2 (I1);

would define the object I2 and at the same time initialize it to the values of I1. Another form of this statement is

integer I2 = I1;

The process of initializing through a copy constructor is known as copy initialization. Remember, the statement

I2 = I1;

will not invoke the copy constructor. However, if I1 and I2 are objects, this statement is legal and simply assigns the values of I1 and I2, member-by-member. This is the task of the overloaded assignment operator (=). We shall see more about this later.

A copy constructor takes a reference to an object of the same class as itself as an argument. Let us consider a simple example of constructing and using a copy constructor as shown in Program 5.4

Program 5.4

```
#include <iostream>
using namespace std;
class code
{
```

```
        int td;
    public:
        code ( ) { }                        // constructor
        code (int a)  { id = a;}            // constructor again
        code (code & x)                     // copy constructor
        {
          id = x.id;                        // copy in the value
        }
        void display(void)
        {
          cout << id;
        }
};
int main ( )
{
        code A (100) ;          // object A is created and initialized
        code B (A) ;                    // copy constructor called
        code C = A;                     // copy constructor called again
        code D;                         // D is created, not initialized
        D = A;                          // copy constructor not called
        code << "\n id of A: "; A.display ( );
        code << "\n id of B: "; B.display ( );
        code << "\n id of C: "; C.display ( );
        code << "\n id of D: "; D.display ( );
return 0;
}
```

The output of Program 5.4 would be:

id of A : 100

id of B : 100

id of C : 100

id of D : 100

Note : A reference variable has been used as an argument to the copy constructor. We cannot pass the argument by value to a copy constructor.

When no copy constructor is defined, the compiler supplies its own copy constructor.

5.8 DYNAMIC CONSTRUCTORS

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator. Program 5.5 shows the use of new, in constructors that are used to construct strings in objects.

Program 5.5 Constructors with new

```
#include <iostream>
#include <string>
using namespace std;
class string
{
        char *name;
        int length;
   public:
        String ( )                              // constructor - 1
        {
          length = 0;
          name = new char [length + 1];
        }
        String (char *s)                        // constructor - 2
        {
           length = strlen (s);
           name = new char [length + 1];        // one additional
                                                // character for \0
        strcpy (name, s);
        }
```

```
        void display (void)
        {cout << name << "\n"; )
        void join (String &a, String &b)
};

void string : : join (String &a, String &b)
{
        length = a.length +b.length;
        delete name;
        name = new char [length+1]; // dynamic allocation
        stropy (name, a.name);
        stropy (name, b.name);\
};
int main ( )
{
        char *first = "Joseph";
        String name1 (first), name2 ("Louis"), name3 ("Lagrange"),
        s1, s2;
        s1. join (name1, name2);
        s2.join (s1, name 3);
        name1. display ( );
        name2.display ( );
        name3. display ( );
        s1. display ( );
        s2. display ( );
        return 0;
}
```

The output of the program 5.5 would be:

Joseph

Louis

Lagrange

Joseph Louis

Joseph Louis Lagrange

Note : *The program uses two constructors. The first is an empty constructor that allows us to declare an array of strings. The second constructor initializes the length of the string, allocates necessary space for the string to be stored and creates the string itself. Note that one additional character space is allocated to hold the end-of-string character '/0'.*

The member function join( ) concatenates two strings. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string functions strcpy( ) and strcat( ). Note that in the function join ( ), length and name are members of the object that calls the function, while a.length and a.name are members of the argument object a. The main ( ) function program concatenates three strings into one string. The output is as shown below:

Joseph Louis Lagrange

5.9 CONSTRUCTING TWO-DIMENSIONAL ARRAYS

We can construct matrix variables using the class type objects. The example in program 5.6 illustrates how to construct a matrix of size m x n.

Program 5.6 Constructing Matrix Objects

```
#include <iostream
using namespace std;
class matrix
{
        int **p; // pointer to matrix
        int d1, d2        // dimensions
    public :
        matrix (int x, int y);
        void get_element (int i, int j, int value)
        { p [i] [j] = value;}
        int & put_element (int i, int j)
        (return p[i] [j];}
};
matrix : : matrix (int x, int y);
```

```
{
        d1 = x;
        d2 = y;
        p = new int *[d1];                  // creates an array pointer
        for (int i = 0; i < d1; i++)
        p[i] = new int [d2];                // creates space for each row
}
int main ( )
{
        int m, n;

        cout << " Enter size of matrix: ";
        cin >> m >> n;
        matrix A(m,n); // matrix object A constructed

        cout << "Enter matrix elements row by row \n";
        int i, j, value;
        for (i = 0; i < m; i++)
                for (j = 0; j < n; j++)
                {
                cin >> value;
                A.get_element (i, j, value);
                }
        cout << "/n";
        cout << A.put_element (1, 2);

        return 0;
};
```

The output of Program 5.6 would be:

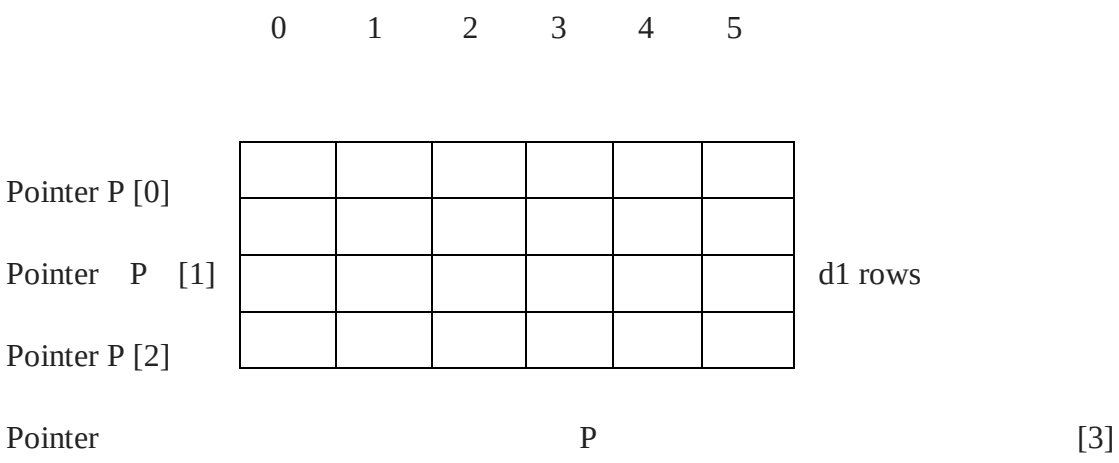
```
Enter size of matrix : 3 4
Enter matrix elements row by row
11 12 13 14
15 16 17 18
```

19 20 21 22

17

17 is the value of the element (1, 2).

d2 columns



x represents the element P[2] [3]

The constructor first creates a vector pointer to an int of size d1. Then, it allocates, iteratively an int type vector of size d2 pointed at by each element p[i]. Thus, space for the elements of a d1 x d2 matrix is allocated from free store as shown above.

5.10 CONST OBJECTS

We may create and use constant objects using const keyword before object declaration. For example, we may create X as a constant object of the class matrix as follows :

const matrix X(m, n); // object X is constant

Any attempt to modify the values of m and n will generate compile-time error. Further, a constant object can call only const member functions. As we know, a const member is a function prototype or function definition where the keyword const appears after the function's signature.

Whenever const objects try to invoke nonconst member functions, the compiler generates errors.

5.11 DESTRUCTORS

A destructor, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name by is preceded by a tilde. For example, the destructor for the class integer can be defined as shown below:

```
~integer ( ) { }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future us.

Whenever new is used to allocate memory in the constructions, we should use delete to free that memory. For example, the destructor for the matrix class discussed above may be defined as follows:

```
matrix : : ~matrix ( )
{
        for ( int i = 0; i < d1; i++)
        delete p [i];
        delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

The example below illustrates that the destructor has been invoked implicitly by the compiler.

Program 5.7 Implementation of Destructions

```
#include <iostream>
using namespace std;
int count = 0;
class test
{
```

```
    public:
        test ( )
        {
        count++
        cout << "\n\nConstructor Msg: Object number "<< count <<
        "created. .";
        }
        ~test ( )
        {
        cout<<"\n\nDestructor Msg: Object number "<< count <<"
        destroyed. .";
        count -- ;
        }
};

int main ( )
{
        cout<<"Inside the main block. .";
        cout<<"\n\nCreating first object T1. .";
        test T1;

        {       // Block 1
                cout << "\n\nInside Block 1. .";
                cout <<"\n\nCreating two more objects T2 and T3. .";
                test T2, T3;
                cout << "\n\nLeaving Block 1. ."
        }
        cout << "\Back inside the main block. .";
        return 0;
}
```

The output of Program 5.7 would be:

```
Inside the main block. .

Creating first object T1. .
```

Constructor Msg: Object number 1 created. .

Inside Block 1. .

Creating two more objects T2 and T3. .

Constructor Msg: Object number 2 created. .

Constructor Msg: Object number 3 created. .

Leaving Block 1. .

Destructor Msg: Object number 3 destroyed.

Destructor Msg: Object number 2 destroyed.

Back inside the main block. .

Destructor Msg: Object number 1 destroyed. .

NOTE : *A class constructor is called everytime an object is created. Similarly, as the program control leaves the current block the objects in the block start getting destroyed and destructors are called for each one of them. Note that the objects are destroyed in the reverse order of their creation. Finally when the main block is exited, destructors are called corresponding to the remaining objects present inside main.*

Similar functionality as depicted in Program 6.7 can be attained by using static data members with constructors and destructors. We can declare a static integer variable count inside a class to keep a track of the numberof its object instantiations. Being static, the variable will be initialized only once. i.e., when the first object instance is created. During all subsequent object creations, the constructor will increment the count variable by one. Similarly, the destrucor will decrement the count variable by one as and when an object gets destroyed. To realize this scenario, the code in Program 5.7 will change slightly, as shown below:

```
#include <iostream>
using namespace std;
class test
{
provate:
```

```
    static int count = 0;
public:
.
.
}
test ( )
{
.
count ++;
}
~test ( )
{
.
count --;
}
```

The primary use of destructors is in freeing up the memory reserved by the object before it gets destroyed. Program 5.8 demonstrates how a destructor releases the memory allocated to an object.

Program 5.8 Memory Allocation to an Object Using Destructor

```
#include <iostream>
#include<conio.h>
using namespace std;
class test
{
    int*a;
    public:
    test (int size)
    {
        a = new int [size];
```

```
            cout << "\n\nConstructor Msg : Integer array of size "<<size<<"

            created. .";

        }
        ~test ( )
        {
            delete a;
            cout <<"\n\nDestructor Msg: Freed up the memory allocated for integer
array";
        }
    };
    int main( )
    {
            int s;
            cout <<"Enter the size of the array. .";
            cin>>s;
            cout << "\n\nCreating an object of test class. .";
            test T(s);
            cout << "\n\nPress any key to end the program. .";
            getch ( );
            return 0;
    }
```

The output of Program 5.8 would be:

Enter the size of the array..5

Creating an object of test class. .

Constructor Msg: Integer array of size 5 created. .

Press any key to end the program. .

Destructor Msg: Freed up the memory allocated for integer array

OPERATOR OVERLOADING AND TYPE CONVERSIONS

5.12 INTRODUCTION

Operator overloading is one of the many exciting features of C++ language. It is an important technique that has enhanced the power of extensibility of C++. We have stated more than once that C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can overload (give additional meaning to) all the C++ operators except the following,

* Class member access operators(., .*).

* Scope resolution operator (: :).

* Size operator (size of).

* Conditional operator (?:).

The reason why we cannot overload these operators maybe attributed to the fact that these operators take names (example class name) as their operand instead of values, as is the case with other normal operators.

Although the semantics of an operator can be extended, we cannot change its syntax, the grammatical rules that govern its use such as the number of operands, precedence and associativity. For example, the multiplication operator will enjoy higher precedence than the addition operator. Remember, when an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

5.13 DEFINING OPERATOR OVERLOADING

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function

called operator function, which describes the task. The general form of an operator function is,

```
return type classname : : operator op (arglist)
{
        Function body // task defined
}
```

where return type is the type of value returned by the specified operation and op is the operator being overloaded. operator op is the function name, where operator is a keyword.

Operator functions must be either member functions (non-static) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions. Arguments may be passed either by value or by reference. Operator functions are declared in the class using prototype as follows:

```
vector operator +(vector);                  // vector addition
vector operator - ( );                      // unary minus
friend vector operator + (vector, vector);  // vector addition
friend vector operator - (vector);          // unary minus
vector operator- (vector &a);               // subtraction
int operator == (vector);                   // comparison
friend int operator == (vector, vector)     // comparison
```

vector is a data type of class and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics).

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operation
2. Declare the operator function operator op( ) in the public part of the class, it may be either remember function or a friend function.
3. Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

op x or x op

for unary operators and

x op y

for binary operators, op x (or x op) would be interpreted as

operator op (x)

for friend functions. Similarly, the expression x op y would be interpreted as either

x.operator op (y)

in case of member functions, or

operator op (x, y)

in case of friend functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.

5.14 OVERLOADING UNARY OPERATORS

Let us consider the unitary minus operator. A minus operator, when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an int or float variable. The unary minus when applied to an object should change the sign of each of its data items.

The following program shows how the unary minus operator is overloaded.

Program 5.9 Overloading unary minus

```
#include <iostream>
using namespace std;
class space
{
        int x;
        int y;
        int z;
    public:
        void getdata (int a, int b, int c);
        void display (void)
        void operator - ( );                    // overload unary minus
};
void space : : getdata (int a, int b, int c)
{
        x = a;
        y = b;
        z = c;
}
void space : : display (void)
{
        cout << "x = "<<x<<" ";
        cout <<"y = "<<y<<" ";
        cout <<"z = "<<z<<"\n";
}
void space : : Operator - ( )
{
        x = -x;
        y = -y;
        z = -z;
}
```

```
int main ( )
{
        space S;
        S.getdata (10, -20, 30);
        cout << "S : ";
        S.display ( );
        -S;                         // activates operator -( ) function
        cout << "-S : ";
        S.display ( );
        return 0;
}
```

The output of the Program 5.9 would be:

S : x = 10 y = -20 z = 30

-S : x = -10 y = 20 z = -30

NOTE: The function operator -( ) takes no argument. Then, what does this operator function do?It changes the sign of data members of the object S. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

Remember, a statement like

```
S2 = -S1;
```

will not work because, the function operator-( ) does not return any value. It can work if the function is modified to return an object.

It is possible to overload a unary minus operator using a friend function as follows:

```
friend void operator -(space &S);          // declaration
void operator -(space &S)                  // definition
{
        s.x = -s.x;
        s.y = -s.y;
        s.z = -s.z;
}
```

NOTE : Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator-( ). Therefore, the changes made inside the operator function will not reflect in the called object.

5.15 OVERLOADING BINARY OPERATORS

We have just seen how to overload an unary operator. The same mechanism can be used to overload a binary operator. In Chapter 6, we illustrated, how to add two complex numbers using a friend function. A statement like

```
C = sum (A, B);          // functional notation
```

was used. The functional notation can be replaced by a natural looking expression.

```
C = A + B;       // arithmetic notation
```

by overloading the + operator using an operator +( ) function. The Program 5.10 illustrates how this is accomplished.

Program 5.10 Overloading + Operator

```
#include <iostream>
using namespace std;
class complx
{
        float x;                                    // real part
        float y;                                    // imaginary part
    public:
        complex ( ) {    }                          // constructor 1
        complex (float real, float imag)            // constructor 2
        { x = real; y = imag; }
        complex operator + (complex);
        void display (void);
};
complex complex : : operator + (complex c)
{
        complex temp;                               // temporary
```

```
        temp.x = x + c.x;                    // these are
        temp.y = y + c.y;                    // float additions
        return (temp);
}

void complex : : display (void)
{
        cout << x << " + j" << y << "\n";
}
int main ( )
{
        complex C1, C2, C3;                  // invokes constructor 1
C1 = complex (2.5, 3.5);                     // invokes constructor 2
        C2 = complex (1.6, 2.7);
        C3 = C1 + C2;

        cout << "C1 = "; C1.display( );
        cout << "C2 = "; C2.display( );
        cout << "C3 = "; C3.display( );

        return 0;
}
```

The output of Program 5.10 would be:

```
C1 = 2.5 + j3.5
C2 = 1.6 + j2.7
C3 = 4.1 + j6.2
```

NOTE : Let us have a close look at the function operator+( ) and see how the operator overloading is implemented.

```
complex complex : : operator+ (complex c)
{
        complex temp;
        temp.x = x + c.x;
```

```
        temp.y = y + c.y;
        return (temp);
    }
```

We should note the following features of this function:

1. It receives only one complex type argument explicitly.
2. It returns a complex type value.
3. It is a member function of complex.

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from? Now let us look at the statement that invokes this function:

```
C3 = C1 + C2;                    // invokes operator+ ( ) function
```

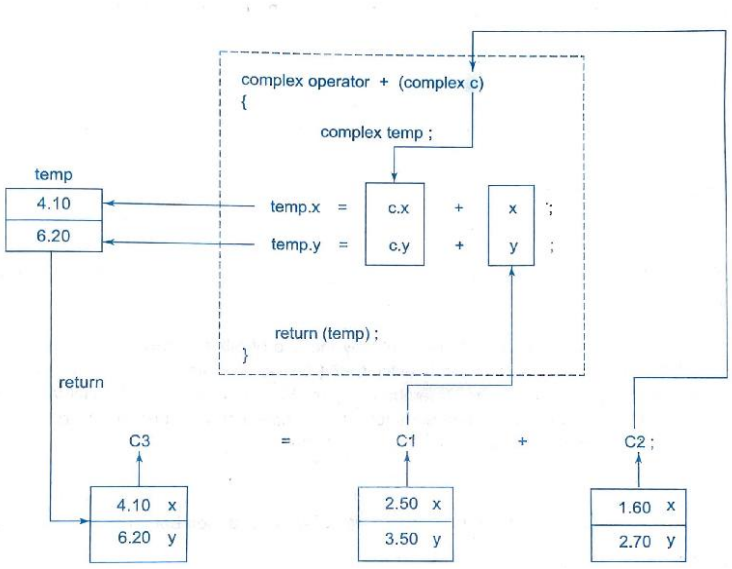
We know that a member function can be invoked only by an object of the same class. Here, the object, C1 takes the responsibility of invoking the function and C2 plays the role of an argument that is passed to the function. The above invocation statement is equivalent to

```
C3 = C1.operator + (C2) ;        // usual function call syntax
```

Therefore, in the operator+( ) function, the data members of C1 are accessed directly and the data members of C2 (that is passes as an argument) are accessed using the dot operator. Thus, both the objects are available for the function. For example, in the statement

```
temp.x = x + c.x;
```

c.x refers to the object C2 and x refers to the object C1.temp.x is the real part of temp that has been created specially to hold the results of addition of C1 and C2. The function returns the complex temp to be assigned to C3. The following figure shows how this is implemented.



As a rule, in overloading of binary operators, the left-hand operand is used to invoke the operator function and the right-hand operand is passed as an argument.

We can avoid the creation of the temp object by replacing the entire function body by the following statement:

return complex ((x + c.x), (y + c.y)); // invokes constructor 2

What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object. Such an object is called a temporary object and goes out of space as soon as the contents are assigned to another object. Using temporary objects can make the code shorter, more efficient and better to read.

5.16 OVERLOADING BINARY OPERATORS USING FRIENDS

As stated earlier, friend functions may be used in the place of member functions for overloading a binary operator, the only difference being that a friend function requires two arguments to be explicitly passed to it, while a member function requires only one.

The complex number program discussed in the previous section can be modified using a friend operator function as follows:

1. Replace the member function declaration by the friend function declaration.

   friend complex operator+ (complex, complex);

2. Redefine the operator function as follows:

```
complex operator + (complex a, complex b)
{
        return complex ( (a.x + b.x), (a.y + b.y) );
}
```

In this case, the statement

```
C3 = C1 + C2;
```

is equivalent to

```
C3 = operator + (C1, C2);
```

In most cases, we will get the same results by the use of either a friend function or a member function. Why then an alternative is made available? There are certain situations where we would like to use a friend function rather than a member function. For instance, consider a situation where we need to use two different types of operands for a binary operator, say, one an object and another built-in type data as shown below,

```
A = B + 2; (or A = B * 2 ;)
```

where A and B are objects of the same class. This will work for a member function, but the statement

```
A = 2 + B; (or A = 2 * B)
```

will not work. This is because the left-hand operand which is responsible for invoking the member function should be an object of the same class. However, friend function allows both approaches. How?

It may be recalled that an object need not be used to invoke a friend function but can be passed as an argument. Thus, we can use a friend function with a built-in type data as the left-hand operand and an object as the right-hand operand. Program 5.11 illustrates this, using scalar multiplication of a vector. It also shows how to overload the input and output operators >> and <<.

Program 5.11 Overloading Operators using Friends

```
#include <iostream.h>
const size = 3;
class vector
{
        int v[size];
public:
        vector ( );                 // constructs null vector
        vector (int *x);            // constructs vector from array
        friend vector operator *(int a, vector b);      // friend  1
        friend vector operator *(vector b, int a);      // friend  2
        friend istream & operator >> (istream &, vector &);
        friend ostream & operator << (ostream &, vector &);
};
vector : : vector ( )
{
        for (int i = 0; i < size; i++)
                v [i] = 0;
}
vector : : vector (int *x)
{
        for (int i = 0; i < size ; i++)
                v [i] = x[i];
}
vector operator *(int a, vector b)
{
        vector c;
        for (int i = 0; i < size; i++)
                c.v[i] = a * b.v[i];
        return c;
}
vector operator *(vector b, int a)
{
```

```
        vector c;
        for (int i = 0; i<size; i++)
                c.v [i] = b.v[i] * a;
        return c;
}
istream & operator >> (istream &din, vector &b)
{
        for int i = 0; i < size; i++
                din >> b.v [i];
        return (din);
}
ostream & operator << (ostream &dout, vector &b)
{
        dout << " (" << b.v [0];
        for (int i = 1; i < size; i++)
                dout << ", "<< b.v [i]
        dout << ") ";
        return (dout);
}
int x[size] = {2,4,6};
{
        vector m;                               // invokes constructor 1
        vector n = x;                           // invokes constructor 2
        cout << "Enter elements of vector m " << "\n";
        cin >> m ;                              // invokes operator>> ( ) function
        cout << "\n";
        cout << "m = " << nm << "\n";           // invokes operator << ( )
        vector p, q;
        p = 2 * m;                              // invokes friend 1
        q = n * 2                               // invokes friend 2
        cout << "\n";
        cout << "p = " << p << "\n";                    // invokes operator << ( )
        cout << "q = " << q << "\n";
```

```
    return 0;
}
```

The output of Program 5.11 would be:

```
Enter elements of vector m

5 10 15

m = (5, 10, 15)

p = (10, 20, 30)

q = (4, 8, 12)
```

The program overloads the operator * two times, thus overloading the operator function operator*( ) itself. In both the cases, the functions are explicitly passed two arguments and they are invoked like any other overloaded function, based on the types of its arguments. This enables us to use both the forms of scalar multiplication such as

```
p = 2 * m;      // equivalent to p = operator* (2, m);

q = n * 2;      // equivalent to q = operator* (n,2);
```

The program and its output are largely self-explanatory. The first constructor

```
vector ( );
```

constructs a vectors whose elements are all zero. Thus

```
vector m;
```

creates a vector m and initializes all its elements to 0. The second constructor

```
vector (int &x);
```

creates a vector and copies the elements pointed to by the pointer argument x into it. Therefore, the statements

```
int x[3] = {2, 4, 6};

vector n = x;
```

creates n as a vector with components 2, 4 and 6.

NOTE : We have used vector variables like m and n in input and output statements just like simple variables. This has been made possible by overloading the operators >> and << using the functions:

friend istream & operator >> (istream &, vector &);

friend ostream & operator << (ostream &, vector &);

istream and ostream are classes defined in the iostream.h file which has been included in the program.

5.17 MANIPULATION OF STRINGS USING OPERATORS

ANSI C implements string using character arrays, pointers and string functions. there are no operators for manipulating the strings. One of the main drawbacks of string manipulations in C is that whenever a string is to be copied, the programmer must first determine its length and allocate the required amount of memory.

Although these limitations exist in C++ as well, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers. Recently, ANSI C++ committee has added a new class called string to the C++ class library that supports all kinds of string manipulations.

For example, we shall be able to use statements like

string3 = string1 + string2

if (string1 >= string2) string = string1;

Strings can be defined as class objects which can be then manipulated like the built-in types. Since the strings vary greatly in size, we use new to allocate memory for each string and a pointer variable to point to the string array. Thus we must create string objects that can hold these two pieces of information, namely, length and location which are necessary for string manipulations. A typical string class will look as follows:

```
class string
{
        char *p;                        // pointer to string
        int len;                        // length of string
```

```
    public:
    . . . . .                    // member functions
    . . . . .                    // to initialize and
    . . . . .                    // manipulate strings
};
```

We shall consider an example to illustrate the application of overloaded operators to strings. The example shown in Program 5.12 overloads two operators, + and <= just to show how they are implemented. This can be extended to cover other operators as well.

Program 5.12 Mathematical Operations on Strings

```
# include <string.h>
# include <iostream.h>
class string
{
    char *p;
    int len;
  public:
    string ( ) {len = 0; p = 0;}          // create null string
    string ( const char *s);              // create strings from arrays
    string (const string & s);            // copy constructor
    ~ string ( ) {delete p;}              // destructor

    // + operator
    friend string operator + (const string & s, const string &t);
    // <= operator
    friend int operator <= (const string &s, const string &t);
    friend void show (const string s);
};
string : : string (const char *s)
{
    len = strlen (s);
    p = new char [len+1];
    strcpy (p, s);
}
```

```
        string : : string (const string &s)
{
        len = s.len;
        p = new char [len + 1];
        strcpy (p,s.p);
}
// overloading + operator
string operator + (const string &s, const string &t)
{
        string temp;
        temp.len = s.len + t.len;
        temp.p = new char [temp.len + 1];
        strcpy (temp.p, s.p);
        strcat (temp.p, t.p);
        return (temp);
}
// overloading <= operator
int operator <= (const string &s, const string &t)
{
        int m = strlen (s.p);
        int n = strlen (t.p);
        if (m <= n) return (1);
        else return (0);
}
void show (const string s)
{
        cout << s.p;
}
int main ( )
{
        string s1 = "New ";
        string s2 = "York ";
        string s3 = "Delhi ";
```

```
    string string1, string2, string3;
    string1 = s1;
    string2 = s2;
    string3 = s1 + s3;
    cout << "\nstring1 = "; show (string1);
    cout << "\nstring2 = "; show (string2);
    cout << "\n";
    cout << "\nstring3 = "; show (string3);
    cout << "\n\n";
    if (string1 <= string3)
    {
        show (string1);
        cout << " smaller than ";
        show (string3);
        cout << "\n";
    }
    else
    {
        show (string3);
        cout << " smaller than ";
        show (string1);
        cout << "\n";
    }
    return 0;
}
```

The output of program 5.12 would be:

string1 = New

string2 = York

string3 = New Delhi

New smaller than New Delhi

5.18 SOME OTHER OPERATOR OVERLOADING EXAMPLES

Overloading the Subscript Operator [ ]

The subscript operator is normally used to access and modify a specific element in an array. Program 5.13 demonstrates the overloading of the subscript operator to customize its behaviour.

Program 5.13 Overloading of the Subscript Operator

```
# include <iostream>
# include<conio.h>
using namespace std;
class arr
{
        int a [5];
    public:
        arr (int *s)
        {
                int i;
                for (i = 0; i < 5; i++)
                    a [i] = s [i];
        }
        int operator [ ] (int k) // Overloading the subscript operator
        {
                return (a [k]);
        }
};
int main( )
{
        int x [5] = {1, 2, 3, 4, 5};
        arr A(x);
        int i;
        for (i = 0; i <5; i++)
        {
```

```
            cout << x[i] << "\t";// Using subscript operator to access the
        private array elements
        }
        getch ( );
        return 0;
}
```

The output of Program 5.13 would be:

1 2 3 4 5

As can be seen in the above program, we have used the subscript operator along with the object name to access the private array elements of the object.

Overloading the Pointer-to-member (->) Operator

The pointer-to-member operator (->) is normally used in conjunction with an object pointer to access any of the object's members. Program 5.14 demonstrates the overloading of the -> operator.

Program 5.14 Overloading of Pointer-to-member Operator

```
#include<iostream>
#include<conio.h>
using namespace std;
class test
{
   public:
        int num;
        test (int j)
        {
                num = j;
```

```
        }
        test * operator -> (void)
        {
                return this;
        }
};
int main ( )
{
        test T(5);
        test *Ptr = &T;
        cout << "T.num = "<< T.num;                    // Accessing num
                                                              normally
        cout << "\nPtr->num = "<<Ptr->num;      // Accessing num using
                                                  normal object pointer
        cout << "\nT->num = "<<T->num;           // Accessing num using
                                                  overloaded -> operator
        getch ( );
        return 0;
}
```

The output of Program 5.14 would be:

```
T.num = 5
Ptr->num = 5
 T->num = 5
```

The above program demonstrates both the normal (Ptr->num) as well as the overloaded (T->num) behaviour of the -> operator.

The statement,

return this;

returns a pointer to itself; that is a pointer to the test class object.

5.19 RULES FOR OVERLOADING OPERATORS

Although it looks simple to redefine the operators, there are certain restrictions and limitations in overloading them. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
4. There are some operators that cannot be overloaded. (See table 7.1)
5. We cannot use friend functions to overload certain operators.(See table 7.2). However, member functions can be used to overload them.
6. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
7. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
8. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
9. Binary arithmetic operators such as +, -, * and / must explicitly return a value. They must not attempt to change their own arguments.

Table Operators that cannot be overloaded

Size of	Size of operator
.	Membership operator
.*	pointer-to-member operator
: :	Scope resolution operator
?:	Conditional operator

Table Where a friend cannot be used

=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
->	Class member access operator

5.20 TYPE CONVERSIONS

We know that when constants and variables of different types are mixed in an expression, C applies automatic type conversion to the operands as per certain rules. Similarly, an assignment operation also causes the automatic type conversion. That type of data to the right of an assignment operator is automatically converted to the type of the variable on the left. For example, the statements.

int m;

float x = 3.14159;

m = x;

convert x to an integer before its value is assigned to M. Thus, the fractional part is truncated. The type conversions are automatic as long as the data types involved are built-in types.

What happens when they are user-defined data types?

Consider the following statement that adds two objects and then assigns the result to a third object.

```
v3 = v1 + v2;          // v1, v2 and v3 are class type objects
```

When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. We have seen, in the case of class objects, that the values of all the data members of the right-hand object are simply copied into the corresponding members of the object on the left-hand. What if one of the operands is an object and the other is a built-in type variable? Or, what if they belong to two different classes?

Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types. We must, therefore design the conversion routines by ourselves, if, such operations are required.

Three types of situations might arise in the data conversion between uncompatible types:

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

We shall discuss all the three cases in detail.

Basic to Class Type:

The conversion from basic type to class type is easy to accomplish. It may be recalled that the use of constructors was illustrated in a number of examples to initialize objects. For example, a constructor was used to build a vector object from an int type array. Similarly, we used another constructor to build a string type object from a char* type variable. These are all examples where constructors perform a defacto type conversion from the argument's type to the constructor's class type.

Consider the following constructor:

```
string : : string (char *a)
{
        length = strlen (a);
        P = new char [length + 1];
        strcpy (P, a);
}
```

This constructor builds a string type object from a char* type variable a. The variable length and p are data members of the class string. Once this constructor has been defined in the string class, it can be used for conversion from char* type to string type. Example:

```
string s1, s2;
char* name1 = "IBM PC";
char* name2 = "Apple Computers";
s1 = string (name1);
s2 = name2;
```

The statement

```
s1 = string (name1);
```

first converts name1 from char* type two string type and then assigns the string type values to the object s1. The statement

```
s2 = name2;
```

and does the same job by invoking the constructor implicitly.

Let us consider another example of converting an int type to a class type.

```
class time
{
        int hrs;
                int mins;
        public:
```

```
. . . . .
. . . . .
time (int t)              // constructor
{
        hrs = t/60;       // t in minutes
        mins = t % 60
}
};
```

The following conversion statements can be used in a function:

```
time T1;                  // object T1 created

int duration = 85

T1 = duration;            // int to class type
```

After this conversion, the hrs member of T1 will contain a value of 1 and mins member a value of 25, denoting 1 hours and 25 minutes.

NOTE: The constructs used for the type conversion take a single argument whose type is to be converted.

In both the examples, the left-hand operand of = operator is always a class object. Therefore, we can also accomplish this conversion using an overloaded = operator.

Class to Basic Type:

The constructors did a define job in type conversion from a basic to class type. What about the conversion from a class to basic type? The constructor functions do not support this operation. Luckily, C++ allows us to define an overloaded casting operator that could be used to convert a class data to a basic type. The general form of an overloaded casting operator function, usually referred to us. A conversion function, is:

```
operator typename ( )
{
```

. . . . .

. . . . . (Function statements)

. . . . .

}

This function converts a class type data to typename. For example, the operator double( ) converts a class object to type double, the operator int ( ) converts a class type object to type int, and so on.

Consider the following conversion function:

```
vector : : operator double ( )
{
double sum = 0;
for (int i = 0; i < size; i++)
        sum = sum + v[i] * v[i];
return sqrt (sum);
}
```

This function converts a vector to the corresponding scalar magnitude. recall that the magnitude of a vector is given by the square root of the sum of thee squares of its components. The operator double( ) can be used as follows:

```
double length = double (V1);
        or
double length = V1;
```

where V1 is an object of type vector. Both the statements have exactly the same same effect. When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do the job.

The casting operator function should satisfy the following conditions:

* It must be a class member.

* It must not specify a return type.

* It must not have any arguments.

Since it is a member function, it is invoked by the object and, therefore, the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

In the string example described in the previous section, we can do the conversion from string to char* as follows:

```
string : : operator char* ( )
{
        return (p);
}
```

One class to Another Class Type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But there are situations where we would like to convert one class type data to another class type.

Example:

```
objX = objY;            // objects of different types
```

objX is an object of class X and objY is an object of class Y. The class Y type data is converted to the class X type data and the converted value is assigned to the objX. Since the conversion takes place from class Y to class X, Y is known as the source class and X is known as the destination class.

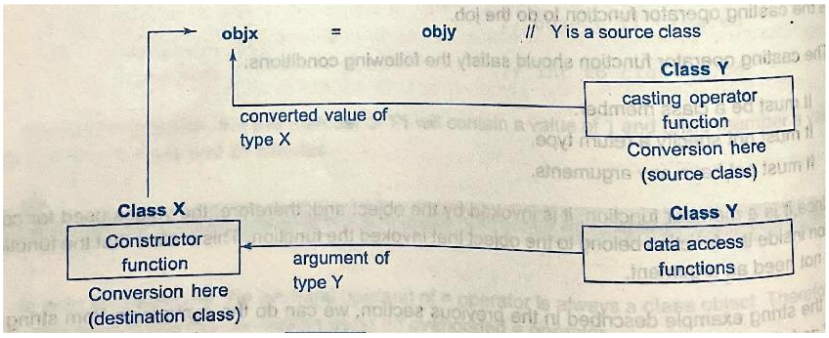
Such conversions between objects of different classes can be carried out by either a constructor or a conversion function. The compiler treats them the same way. Then, how do we decide which form to use? It depends upon where we want the type-conversion function to be located in the source class or in the destination class.

We know that the casting operator function

operator typename ( )

converts the object of which it is a member to typename. The typename may be a built-in type or a user-defined one (another class type). In the case of conversions between objects, typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used (i.e., source class). The conversion takes place in the source class and the result is given to the destination class object.

Now consider a single-argument constructor function which serves as an instruction for converting the argument's type to the class type of which it is a member. This implies that the argument belongs to the source class and is passed to the destination class for conversion. This makes it necessary that the conversion constructor be placed in the destination class. The following figure illustrates these two approaches.



The following table provides a summary of all the three conversions. It shows that the conversion from a class to any other type (or any other class) should make use of of a casting operator in the source class. On the other hand, to perform the conversion from any other type/class to a class type, a constructor should be used in the destination class.

Table : type conversions

Conversion required	Conversion takes place in Source class	 Destination class
Basic → class	Not applicable	Constructor
Class → basic	Casting operator	Not applicable
Class → class	Casting operator	Constructor

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument.

Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destination class.

A DATA CONVERSION EXAMPLE:

Let us consider an example of an inventory of products in store. One way of recording the details of the products is to record their code number, total items in the stock and the cost of each item. Another approach is to just specify the item code and the value of the item in the stock. The example shown in Program 5.15 uses two classes and shows how to convert data of one type to another.

Program 5.15 Data Conversions

```
# include <iostream>
using namespace std;
class invent2;                  // destination class declared
class invent1;                  // source class
{
int code;                       // item code
int items;                      // no. of items
float price;                    // cost of each item
public:
invent1 (int a, int b, float c)
{
code = a;
items = b;
price = c;
}
void putdata ( )
```

```
{
cout << "Code: " << code << "\n";
cout << "Items: " << items << "\n";
cout << "Value: " << price << "\n";
}
int getcode ( ) {return code; }
int getitems ( ) {return items; }
float getprice ( ) {return price; }
operator float ( ) { return (items *price); }
/ * operator invent2 ( )                    // invent1 to invent2
{
invent2 temp;
temp.code = code;
temp.value = price * items;
return temp;
} * /
};      // End of source class
class invent2                               // destination class
{
int code;
float value;
public:
invent2 ( )
```

```
{
code = 0; value = 0;
}
invent2 (int x, float y)                    // construction for initialization
{
code = x;
value = y;
}
void putdata ( )
{
cout << "Code: " << code << "\n" ;
cout << "Value " << value << "\n";
}
invent2 (invent 1 p)                        // construction for conversion
{
code = p.getcode ( );
value = p.getitems ( ) * p.getprice ( );
}
};                                          // End of destination class
int main ( )
{
invent1 s1 (100, 5, 140.0);
invent2 d1;
```

```
float total_value;
/* invent1 To float */
total_value = s1;
/* invent1 To invent2 */
d1 = s1;
cout << "Product details - invent1 type" << "\n";
s1. putdata ( );
cout << "Value = " << total_value << "\n\n";
cout << "Product details - invent2 type " << "\n";
d1.putdata ( );
return 0;
}
```

The output of Program 5.15 would be:

```
Product details - invent1 type
Code : 100
Items : 5
Value : 140
Stock value
Value = 700
Product details-invent2 type
Code : 100
Value : 700
```

NOTE : We have used the conversion function

operator float ( )

in the class invent1 to convert the invent1 type data to a float. The constructor

invent2 (invent1)

is used in the class invent2 to convert the invent1 type data to the invent2 type data.

Remember that we can also use the casting operator function

operator invent2 ( )

in the class invent1 to convert invent1 type to invent2 type. However, it is important that we do not use both the constructor and the casting operator for the same type conversion, since this introduces an ambiguity as to how the conversion should be performed.

Study Learning Material Prepared by

Dr. S.N. LEENA NELSON M.Sc., M.Phil., Ph.D.

Associate Professor & Head, Department of Mathematics,

Women's Christian College, Nagercoil – 629 001,

Kanyakumari District, Tamilnadu, India.