1.16. 'inline' FUNCTIONS

While calling a normal function each time, a substantial amount of time is spent on overheads involved in calling and returning mechanism. Typically, the arguments are pushed on to the stack and when a function is called several registers are saved and later restored when the function returns. The problem is that, the execution of machine instructions for these activities takes huge amount of time. Especially when small functions are called, these overheads has to be eliminated.

Hence, to eliminate the cost of calls to small function definitions, C++ introduces a new feature called *inline function*. An *inline function* is a function that is expanded inline within the main program at the point of invocation when it is invoked. This is similar to using a macro. To cause a function to be expanded inline rather than called, the function definition has be preceded with the inline keyword. The syntax for defining inline function is as follows:

```
inline <function_name>(formal_arguments) { body };
```

where,

`inline` → is a keyword used to make a function inline

`function_name` → is the valid name of a function or identifier.

`formal_arguments` → contains the parameter list passed by the calling function.

`body` → consist of function definition.

Program 1.21

```
/* 1.21.cpp */
#include <iostream>
using namespace std;
inline int min(int a, int b)
{
   return a < b ? a : b;    //conditional or ternary operator
}
int main()
{
   cout << min(10, 20);
   return 0;
}
```

As far as the compiler is concerned, the above program is equivalent to the below one.

```
#include <iostream>
using namespace std;
int main()
{
   cout << (10 < 20 ? 10 : 20);
   return 0;
}
```

Though expanding function calls in line can produce faster execution of the code, it can also result in larger code size because of duplication of code. inline functions are not used where functions are called several times, to avoid increase in size of the executable program which occupies a lot of space in the computer's memory during run time. Hence it is best to use inline functions for very small functions. Also it is a good practice to inline only those functions that will significantly improve the performance of the program.

CAUTIONS

Inline is just a request to the compiler and not a command. Hence the working of inline functions is not guaranteed at all times. If a function cannot be made inline, it will simply be called as a normal function.

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

- If there are static variables in the function
- If inline functions are recursive
- For functions involving looping constructs

1.17. SCOPE RESOLUTION OPERATOR (::)

Scope resolution operator (::) is used to resolve the global scope of a particular object. Let us first write a simple C program to understand the necessity of scope resolution operator in C++.

Program 1.22

```
/* 1.22.c */
#include <stdio.h>
int a = 10; /* Global variable */
int main()
{
   int a = 20; /* Local variable */
   printf("a = %d", a); /* prints the value of local variable */
   return 0;
}
```

Output:

```
a = 20
```

In the above program it can be observed that, the output is 20. That is, in the 'main( )' function, the value of local variable 'a' is taken into consideration while printing. There is no way to print the value of the global variable. In otherwords there is no way to resolve the global scope inside the 'main( )' function. Now, let us see how the global scope is resolved in C++ using the scope resolution operator inside the 'main( )' function. This is shown in the program 1.23 below.

Program 1.23

```
/* 1.23.cpp */
#include <iostream>
using namespace std;
int a = 10; // Global variable
int main()
{
   int a = 20; // Local variable
   cout << "a = " << a << endl; // prints value in the local variable
   cout <<" a = " << ::a << endl; //prints value in the global variable
   return 0;
}
```

Output:

```
a = 20
a = 10
```

1.18. STRUCTURES REVIEW

What is the necessity of Structures?

A normal variable can hold a data of one single type depending on the data type of the variable. For example if a variable is declared as

```
int x;
```

then, x is a variable of type int, which can hold a single data of an integer type. Similarly, if a variable is declared as

```
float y;
```

then, y is a variable of type float, which can store only floating point type of data. The above two examples suffers with a common disadvantage that, the variables cannot store

Classes and Objects – An Introduction

2

2.1. INTRODUCTION

In the previous chapter we learnt about the basic differences between C and C++, along with new features of C++. By the end of the first chapter we recalled the use of structures in C language. The knowledge of structures is very important to understand the 'class' concept in C++. Structure and class in C++ are almost similar but with only two basic differences out of which, one will be discussed in this chapter and the other is discussed in chapter 15, section 15.16. This chapter also deals with the working of 'this' pointer which is an internal pointer.

Hence, before directly dwelling into the concept of classes and objects in C++, let us first clearly understand what is the drawback of structures in C and find the differences between structures in C and structures in C++. And after this, we shall discuss relationship between the structures in C++ and classes in C++.

2.2. STRUCTURES IN C

Structures in C helps in binding (packing) logically related data of different or same data types together. Once the structure type is defined, we can create variables of that type using declaration statements (also a definition statement).

For example, let us consider a simple program as below.

Program 2.1

```
/*2.1.c */
#include <stdio.h>
struct simple
{
   int a;
   double b;
};
int main( )
{
    struct simple e1, e2;
    e1.a = 10;
    e1.b = 20.0;
    e2.a = 1;
    e2.b = 2.0;
    printf( "%d\n%f\n%d\n%f\n", e1.a, e1.b, e2.a, e2.b );
```

```
    return 0;
}
```

Output:

```
10
20.0
1
2.0
```

In the above program 2.1,

```
struct simple
{
    int a;
    double b;
};
```

is the structure declaration. The keyword struct creates a new data type 'simple' that can hold two elements 'a' and 'b' of different data types. These elements are known as structure elements. But it should be noted that, structure declaration does not actually allocate memory for its members. The memory is actually allocated once the variable of structure type is created.

In the above program, in the main function, the statement,

```
struct simple e1, e2 ;
```

creates two variables 'e1' and 'e2' of type 'struct simple'. At this point of variable creation, the memory is allocated for the structure members or elements 'a' and 'b'. Each variable has its own copy of structure elements 'a' and 'b'. This can be pictorially shown as in Figure 2.1.

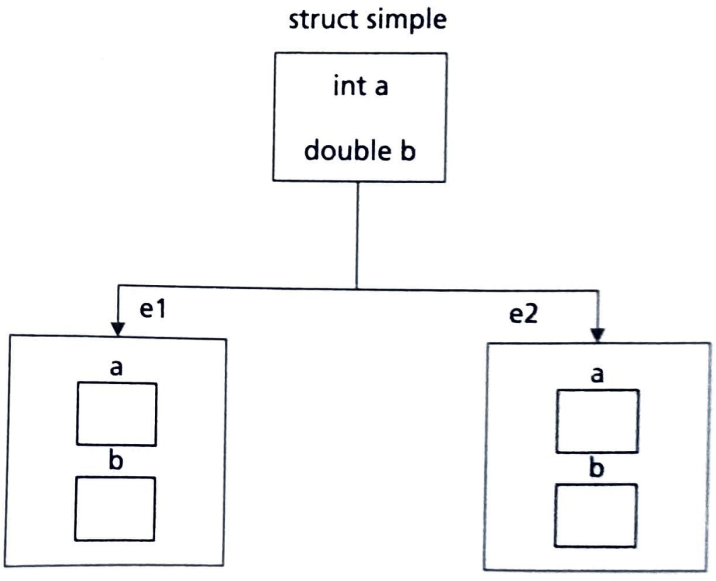

Figure 2.1

The statements,

```
e1.a = 10;
e1.b = 20.0;
e2.a = 1;
e2.b = 2.0;
```

shows how the structure elements of each structure variables are assigned values using the dot operator (.) or object to member access operator. That is,

```
e1.a = 10 ;
```

says, assign the value 10, to the structure element 'a' belonging to the structure variable 'e1'.

Similarly,

```
e2.b = 2.0 ;
```

says, assign the value 2.0 to the structure element 'b', belonging to the variable 'e2'. The output of the program 2.1 can be understood easily by self analysis.

Now, after learning to declare a structure, create its variables and assign values to its elements, let us understand the disadvantage of structures in C. For that, let us consider another program.

Program 2.2

```
/* 2.2.c */
#include <stdio.h>
void fun1( int, int ); /* Function prototype */
void fun2( int , int ); /* Function prototype */
struct simple
{
   int a;
   int b;
};
struct simple e1, e2;
int main( )
{
   fun1( 5, 7 );
   fun2( 6, 8 );
   printf( "%d\t%d\t%d\t%d\t", e1.a, e1.b, e2.a, e2.b );
   return 0;
}
void fun1( int x, int y )
{
   e1.a = x ;
```

```
    e1.b = y ;
}
void fun2( int x, int y )
{
    e2.a = x ;
    e2.b = y ;
}
```

Output :

```
5  7  6  8
```

In the above program 2.2, two variables of type 'struct simple' are created. But unlike the previous program, the structure elements are not directly assigned the values, instead, two functions 'fun1( )' and 'fun2( )' are called from the 'main( )' function and those functions initialize the value to the structure elements. And here lies the threat of data security. That is, in C structures, any function can modify the values of the structure elements,i.e., any functions from any scope can have an access to the structure members. There is no security to the data. If the structure 'struct simple' is dedicated to do some specific operation and if a function unrelated to that specific operation, can change the values of the structure, then the structure variables are said to be polluted. This is a drawback in C structures.

Other than the disadvantage that is discussed above there is another disadvantage in C structures. That is, there is no provision for functions which are logically related to a particular structure and its data members, which operates on the structure data members to be defined or declared inside the structure itself. In other words, in C structures, the function definition cannot be packed within a structure along with the data members. This can be shown with a dummy piece of code.

```
struct employee
{
    int a;
    get_salary( ) /* Not allowed in C structures */
    {
        /* Function definition */
    }
} ;
```

If this concept were to be provided in C structures then the functions related to the structure elements could be packed together and unrelated functions can be kept away from polluting the structure variables. But this concept of packing functions inside the structure declaration is not available in C language.

These disadvantages can be overcome by the structures in C++.

2.3. STRUCTURES IN C++

'struct' keyword is redefined in C++. Let us first take up the example of structures in C++.

Program 2.3

```
/* 2.3.cpp */
#include <iostream>
using namespace std;
struct employee
{
   int salary;
   int bonus;
   void get_salary( int age, int y )
   {
      if ( age < 20 )
      {
         salary = ( y - (y / 2) );
      }
      else
      {
         salary = y;
      }
   }
   int put_salary( )
   {
      return salary;
   }
};
int main( )
{
   struct employee e1, e2;
   e1.get_salary( 19, 10000 );
   e2.get_salary( 22, 10000 );
   cout << e1.put_salary( ) << endl;
   cout << e2.put_salary( ) << endl;
   return 0;
}
```

Output:

```
5000
10000
```

```
    e1.get_salary( 19, 10000 );
    e2.get_salary( 22, 10000 );
    cout << e1.put_salary( ) << endl;
    cout << e2.put_salary( ) << endl;
    e1.salary = 8000;
    e2.salary = 9000;
    cout << e1.salary << endl;
    cout << e2.salary << endl;
    return 0;
}
```

Output:

```
5000
10000
8000
9000
```

In the above program, the data member 'salary' of both the objects 'e1' and 'e2' is accessed or modified from the main function, i.e., without invoking the member function of the structure.

Now, if a non-member function like the 'main( )' can have access or modify the value of the data member belonging to a structure, then how is the security for data member provided or guaranteed? How can we secure the data from being accessed by non-member functions. The answer is by using the *access specifiers* or *access modifiers* or *visibility labels.*

2.4. ACCESS SPECIFIERS

There are three member access specifiers in C++. They are extensively used in C++ to implement the concept called data hiding. They are

- private
- public
- protected

Private:

A data member or member function that is declared as *private* can be accessed from within the structure/class; which means only member functions of that structure / class can access the private data member of that structure/class. The private data members are not accessible outside the class. They can be accessed by the friend functions (discussed

in chapter 3) of this class. ***By default the data member is private for a class, but not for a structure.*** The class concept is explained in next section.

Public :

The data members and member functions that are declared in the *public* section of the structure/class declaration can be accessed from anywhere within a program. ***By default the data members are public for a structure, but not for a class.***

Protected:

The data members and member functions that are declared in the *protected* section of a particular class can be accessed only by the member functions and friends of that class. Also the protected members can be accessed by the member functions and friends derived from that class. It is not accessible outside the class purview. This is dealt in more detail in the chapter 6. Please ignore this until chapter 6.

Let us put these access specifiers into work, so that we understand the concept practically. Check out the below program 2.5.

Program 2.5

```
/*2.5.cpp */
#include <iostream>
using namespace std;
struct simple
{
   private:
         int a;
         int b;
   public :
         int c;
         void set_data( int x, int y )
         {
            a = x;
            b = y;
         }
         int get_data( )
         {
            return ( a + b );
         }
};
int main( )
{
```

```
    struct simple e;
    e.set_data( 5, 5 );
    cout << e.get_data( ) << endl;
    e.b = 20;                    // Error
    e.c = 30;                 // Correct
    return 0;
}
```

In the above structure, the data members 'a' and 'b' are declared as private(note that ':' is very much mandatory after access specifiers), which means that, these data members can be accessed only by the member functions of the structure to which the data member belongs. Hence, the function call

```
e.set_data( );
e.get_data( );
```

are given permission to access the data members and are valid statements. While the statement

```
e.b = 20;
```

in 'main( )' function is invalid and flashes an error message because, 'main( )' is not a member function of the structure 'struct simple' and is not allowed to access or modify the private data member of the discussing structure. But the data member 'c' in the above structure declaration is declared public. Therefore, 'c' can be accessed outside the structure definition also. Hence the statement,

```
e.c = 30 ;
```

in the 'main( )' function is perfectly valid and allowed in C++. As stated earlier, by default the members of a structure are public.

Program 2.6

```
/*2.6.cpp*/
#include <iostream>
using namespace std;
struct simple
{
        int c;                                    // public by default
        void set_data( int x, int y )
        {
           a = x;
           b = y;
        }
        int get_data( )
```

```
        {
            return ( a + b );
        }
private:
        int a;
        int b;
};
int main( )
{
    struct simple e;
    e.set_data( 5, 5 );
    cout << e.get_data( ) << endl;
    e.b = 20;                    // Error
    e.c = 30;                  // Correct
    return 0;
}
```

Note: As struct is a keyword derived from C, C++ uses or replaced the struct keyword by another keyword class. The only difference between class and struct in C++ is that, by default the members of a class are declared private and the members of structures are declared public.

Hence from now on, lets start using the class keyword. Before that lets redefine classes and objects.

2.5. CLASSES

A class is a basic building block of Object Oriented Programming. It is a way to bind the data and its logically related functions together. A class is an abstract data type that can be treated like any other built-in data type. A class definition has two parts.

- Class head
- Class body

Ex:

```
Class head → class name_of_class
Class body → {
                data members;
                member functions;
             };
```

Class head :

Consists of keyword 'class' followed by a user defined class name.

Class body:

Consists of data members (either of private, public, protected or any combination of each) and member function definition or declaration(either of private, public, protected or any combination of each) with a opening brace '{' and a closing brace '}' followed by a semi colon ';'.

An example class definition is as follows:

```
class simple
{
   private :
         int a;
         int b;
   public :
         float c;
         void set_data( int x, int y )
         {
            a = x;
            b = y;
         }
         int get_data( )
         {
            return ( a + b );
         }
};
```

From the above example, 'class' is a keyword used to create a new abstract data type. 'simple' is the name of the class (user-defined) , which forms the new data type. The class declaration starts with an opening brace and ends with a closing brace followed by a semicolon(;) as shown in the example.

private(followed by :)and public(followed by :) are known as *access specifiers* or *access modifiers*, which decide the accessibility of data members of the class by the member functions and non-member functions of the class 'simple'. 'a' and 'b' are the private data members, both of type 'int' which can be accessed only by the class member functions. 'c' is the public data member of type float that can be accessed from anywhere in the program. The data members can be of any data type. The data members cannot be initialized within the class declarations. It can be initialized only using constructors (discussed in detail in chapter 5) once the object of that class is created.

Characteristics of member functions :

- A member function's name is invisible outside the class, i.e, the member functions are declared within the scope of its class.

- A member function can be defined inside the class or outside the class(usin scope resolution operator).
- A member function can have access to private, protected and public data membe of the its class, but cannot access private data members of another class.

2.6. OBJECTS IN C++

Objects are the variables of classes. Object is an instance of a class. An object is a abstraction of real world entity. It has a set of data and functions, that operate on dat Once a class is defined, we can create any number of objects of that class type. The clas declaration just gives a template of the data and functions. The memory for the clas members are allocated only once the class object is defined or created. The process o creating objects is known as instantiation. The syntax for declaring objects is as follows

```
<class_name> <object_name1>,<object_name2>,…..<object_name3>;
```

For example:

For the above class declaration 'class simple', the objects can be created as follows:

```
simple e1, e2…en;
```

where, `e1, e2,...en` are the different objects for the same class 'simple'. It is at th point that the memory for the class members are allocated. Each object has an individu copy of data members, but all objects share a single copy of member functions.

Characteristics of Objects :

- An object has its own copy of data members.
- The scope of an object is determined by the place in which the object defined.
- An object can be passed to a function like a normal variable.
- The members of the class can be accessed by the object using the *object to memb access operator* or *dot operator* (.) .

For example:

```
e1. get_data( ); // object e1 accesses the member function of the
                 // class 'simple'.

e2. c = 20; // assigning value to the data member 'c' of the class
            // belonging to the object e2. 'e2' is a variable or
            // object of some class.
```

Let us work on a simple program shown below to make things very clear.

Program 2.7

```
/* 2.7.cpp*/
#include <iostream>
using namespace std;
class largest
{
   private:
         int a;
         int b;
   public :
         void input_data( int x, int y )
         {
            a = x;
            b = y;
         }
         int big( )
         {
            if ( a > b )
               return a;
            else
               return b;
          }
};

int main( )
{
      largest l;          // 'l' is an object of class largest
      int a, b;
      cout << "Enter the two numbers " << endl;
      cin >> a >> b;
      l.input_data( a, b); //calling the class member function using
                           // object and dot operator
      cout << The largest number is " << l.big( ) << endl;
      return 0;
 }
```

2.7. CHARACTERISTICS OF ACCESS SPECIFIERS(PRIVATE, PUBLIC AND PROTECTED)

The characteristics of access specifiers are as follows:

- private section of a class can have both the data members and member functions, but usually data members are made private for data security.

Example:

```
class A
  {
     private :
        int a;
        int put_data( );
     public:
        int c;
        int get_data( );
  };
```

- It is not mandatory that, private section has to be declared first in the class and then the public section. It can be done in the reverse manner also.

Example:

```
class A
{

   public :
         int c;
         int get_data( );
   private :
         int a;
         int put_data( );
};
```

- If no member access specifier or modifier is specified, then, by default the members are private for the class.

Example:

```
class A
{
      int a; //private by default
      int put_data( );//private by default
   public:
      int c;
      int get_data( );
};
```

- There may be any number of private, public or protected sections in a class declaration.

- protected access modifier or specifier is used for declaring the class members which can be accessed by its own class and its derived class.(protected access specifier is explained in detail in chapter 6).

2.8. HOW TO DEFINE A FUNCTION OUTSIDE A CLASS ?

Till now we have seen, the definition of a function inside the class declaration. But can we define a member function of a class outside the class declaration. Yes. It can be done using the *scope resolution operator* (::). But the prototype of the function has to be within the class declaration.

The syntax is as follows:

```
<data_type> <class_name> : : <function_name>( )
{
   // member function definition
}
```

where,

`data_type` → return type of the function_name( )

`class_name` → The class to which the `function_name` is a member function

`function_name` → The name of the member function of a class `class_name`.

Let us write a program to understand this.

Program 2.8

```
/* 2.8.cpp */
#include <iostream>
using namespace std;
class simple
{
      int a;
      int b;
   public:
      int add( int , int  ); // prototype of the member function.
} ;
int simple :: add( int x, int y) //member function definition outside
                                 //the class.
```

```
{
    int z ;
    a = x ;
    b = y ;
    z = a + b ;
    return z ;
}
int main( )
{
    simple e ;
    cout << e.add( 5, 10 ) << endl ;
    return 0 ;
}
```

Output :

```
15
```

In the above program, the function definition for 'add( )' is as follows.

```
int simple :: add( int x, int y)
{
        int z ;
        a = x ;
        b = y ;
        z = a + b ;
        return z ;
}
```

As shown above, the function 'add ( )' is defined outside the class using scope resolution operator (::). The scope resolution operator tells the scope to which the function belongs. In other words, in the above function definition, the scope resolution operator (::) says that, the function 'add( )' belongs to the class 'simple' and is defined outside the class declaration.

2.9. INITIALIZATION OF VARIABLES IN C++

In C++, the initialization of variables is done in two ways. The first type is like the way we initialize in C language using equal (=)sign.

Ex: `int i = 16;`

The other way is as shown in the example below

Ex: `int i(16);`

The second form is like that because, all built –in data types are considered as classes and the variables of these data types are treated as objects.

2.10. THE ARROW OPERATOR

The arrow (->) operator is also called as *pointer to member access operator.* The arrow operator is used when member functions or member data has to be accessed through a pointer which is pointing to an object of a class, or in other sense, when a pointer contains the address of the object of a particular class.

The general form for using arrow operator is as follows:

```
pointer_to_object->class_member;
```

Check out the following program 2.9 to understand how arrow operator works.

Program 2.9

```
#include <iostream>
using namespace std;
class simple
{
      int a;
      int b;
   public:
      int c;
      int add( int, int );
};
int simple :: add( int x, int y )
{
   a = x;
   b = y;
   return ( a + b );
}
int main( )
{
   int sum;
   simple e;
   sum = e.add( 6, 5 );
   cout << sum << endl;
   e.c = 10;
   cout << e.c << endl;
   simple *e1 = &e;
   sum = e1->add( 5, 25 );
   cout << sum << endl; // accessing the class member using
                        // arrow operator
```

```
    e1->c = 20;//accessing the class member using arrow operator
    cout << e1->c << endl;

    return 0;
}
```

Output:

```
11
10
30
20
```

In the above program, notice the two ways of accessing the members of a class. The one way is using the object and dot operator as shown below.

```
e.add( 6, 5);
e.c = 10;
```

Here, 'e' is an object of class 'simple'. The other way is using the arrow operator and pointer to an object as shown below.

```
e1->add( 5, 25 );
e1->c = 20;
```

Here, 'e1' is a pointer which is pointing to the object 'e' of class simple. Hence we are using the arrow operator.

2.11. THE 'this' POINTER

'this' is a keyword which is used to store the address of the object that invokes a member function. Each member function when invoked, a pointer implicitly holds the address of the object itself, and it is called the *'this'* pointer. There is no need for a user to declare the 'this' pointer. It is internally defined. When an object is used to invoke a class member function, then, the address of that object is automatically assigned to the *'this'* pointer.

Explicitly using the 'this' pointer:

We can explicitly use the 'this' pointer in a function definition as shown in the following example program 2.10.

Program 2.10

```
/* 2.10.cpp*/
#include <iostream>
using namespace std;
```

```
class simple
{
      int a;
   public:
      void set_data( int x)
      {
         this->a = x;
      }
      void display(void)
      {
         cout << "The value of a = " << this->a << endl;
         cout << "The address of object e = " << this << endl;
      }
};
int main( )
{
   simple e;
   e.set_data( 5 );
   e.display( );
   return 0;
}
```

Output:

```
The value of a = 5
The address of object e = 0x8feefff4
```

In the above program, we can observe that, to assign a value to the data member 'a', we do it through

```
this->a = x;
```

Because, 'this' is an inbuilt pointer which always points or contains the address of the object with which the member functions is invoked. As soon as the member function is invoked, the address of the object with which it is invoked gets stored in the 'this' pointer.

Implicitly using the 'this' pointer :

Note that, in the below program, we are accessing the class member 'a' without using the 'this' pointer.

Program 2.11

```
/* 2.11.cpp*/
#include <iostream>
using namespace std;
```

```
class simple
{
        int a;
   public:
        void set_data( int x)
        {
           a = x;
        }
        void display(void)
        {
           cout << "The value of a = " << a << endl;
        }
};
int main( )
{
   simple e;
   e.set_data( 5 );
   e.display( );
   return 0;
}
```

Output:

```
5
```

The above program looks like a normal program, without the keyword 'this' anywh in the program. Since C++ allows or permits the use of short hand form

```
a = x;
```

we have not used the 'this' pointer explicitly, however, we have been using the 't pointer implicitly when the member functions are invoked. The pointer to the cl object is passed by the compiler when calling the the function definition as an additio argument along with the user set parameters. More on this is explained in detail in coming section.

What happens during a compilation of a C++ program including a cla declaration?

Let us take the example program 2.11.cpp as a reference program for our discussi During compilation of the above program, 2.11.cpp, the compiler first checks whet any attempt is made to access the private members of an object by non-member functio If yes, the compilation stops and reports an error. If no, the complier converts the C+ code with the 'class' keyword into a C++ code with the 'struct' keyword.

Before compilation:

```
class simple
{
        int a;
    public:
        void set_data( int x)
        {
           a = x;
        }
        void display(void)
        {
           cout << "The value of a = " <<  a << endl;
           cout << "The address of object e = " << this << endl;
        }
};
int main( )
{
     simple e;
     e.set_data( 5 );
     e.display( );
     return 0;
}
```

During compilation :

```
struct simple
{
      int a;
   void set_data( simple * const this, int x )
   {
      this->a = x;
   }
   void display( simple * const this )
   {
      cout << "The value of a = " << this-> a << endl;
      cout << "The address of object e = " << this << endl;
   }
};
int main( )
{
   simple e;
   set_data( &e, 5 );
   display( &e ) ;
   return 0;
}
```

The changes after compilation are:

- The invoking functions 'e1.set_data( )' becomes 'set_data(&e, 5)', i.e., th function passes the address of the object to the function definition. Similary th 'e1.display( )' becomes 'display(&e)'.
- The function definition 'void set_data(int x)' becomes 'void set_data(simple const this, int x)' . 'this' is a pointer which holds the address of the object bein passed.
- The data members are referred using 'this' pointer i.e. a = x , becomes

```
this->a = x;
```

Characteristics of 'this' pointer :

- The 'this' pointer is a built-in pointer.
- The 'this' pointer points to the object or contains the addresses of the object whic is in operation.
- The data member of the object can be accessed using the this pointer.
- As 'this' pointer is a pointer variable arrow operator is used to access the dat members.

2.12. SUMMARY

- We can declare and define the functions within a structure declaration.
- By default, the members of the structures are public and the members of a class ar private.
- The three access specifiers available in C++ are private, public and protected.
- Classes and objects are the basic building blocks of object oriented programming i C++.
- An object is an instance of a class.
- 'this' is an internal pointer which points to the object of a particular class.
- Member functions can be defined within a class or outside the class. For defining th member function outside the class, scope resolution operator is used.