CHAPTER 5

Object-Oriented Programming C++

Well, in the earlier chapters we have concentrated on the various key features of the structured programming that will be made use of in the object-oriented programming. In fact, a few concepts explained earlier are advanced structured programming features available with C++ and not with C. As stated already, these concepts of structured programming are so very crucial to object-oriented programming. Objects will be constructed by exploiting the various structured programming concepts. From this chapter onwards, we shall explore the important concepts of object-oriented programming, namely Data Encapsulation, Data Abstraction, Operator Overloading, Polymorphism and Inheritance, and Class Templates. In this chapter, we shall details on provide data encapsulation and data abstraction concepts.

5.1 Data Encapsulation and Abstraction

As a part of structured programming, we have seen that it is variable declarations [data] that uniquely identify entities in a program. Structured programming as a result of modularization had several functions performing different subtasks in an integrated fashion under the control of `main()` by manipulating the various variables or identifiers or constants, or data declared in the main in other functions. But then all along in structured programming data and functions which operate on these data were viewed as separate entities and more importance was given to functions.

One key software engineering observation is that data is not given the importance that it is due for. It is in fact these data that distinguish or identify entities. And also in structured programming there is no way of associating functions with specific data. Yes, structured programming does support type checking enabled function calls, but this was only to ensure that the function gets called with the correct number of arguments and the correct data type for each argument. So, this only associates functions with their data types and not with specific data.

5.1.1 Data Encapsulation

Object-oriented programming differs from structured programming in this approach. OOP encapsulates data or what can be called as attributes and functions or behaviours into a single unit or entity called a class. Classes can be viewed as packages. The data and functions of a

class are associated or fixed together. It is this association of data and functions together that is so very crucial to object-oriented programming. This concept of packaging or merging data and functions into a single unit is referred to as Data Encapsulation.

The naming is based on the reasoning that among this set of Data and Functions it is data that is more important and encapsulation refers to the concept of merging data and functions together or encapsulating data and functions within a single entity called a Class. Encapsulation is the process of placing multiple items with a single large entity. The multiple items are data and functions and the single large entity is the class. To correlate with set theory, a class is like a set and data, and functions that form the class are elements or members of the set class. It is based on this reasoning that data and functions of a class are often referred to as data members and member functions. [They are members of the class.]

For a better understanding of these concepts, let us consider a day to day example. A class can be treated as the plan of a building. From a single plan multiple buildings can be created. From a class a programmer can create objects. As a part of this example, we have already said that multiple buildings can be created from a single plan which means that the same plan can be reused as many times the end user wishes to. Similarly, a single class can be reused to create multiple objects. End users when distributed with a class can reuse the class with or without modifications to the class and create multiple objects. This is nothing but the reusability concept of object-oriented programming.

5.1.2 Data Abstraction

Another key property of classes is the concept of Information Hiding, also referred to as data abstraction. An object-oriented program would involved many objects to be manipulated, which means there should be interaction or communication between these objects to achieve a solution for the initial problem definition. Thus, although intercommunication between objects is required [supported via interfaces], a class is not allowed to know the details [data or functions of another class].

In other words, details of implementation of a class are hidden from other classes. It is the hiding of implementation details of a class from other classes that promotes data security to a great extent. Data abstraction is that key feature of OOP wherein end users are provided only with interfaces for manipulation of objects and not the implementation details of the class.

5.2 C++ Characteristics

The programming language we will explore in this and the following chapters is C++, an object-oriented language. With structured programming language the programmers approach to code development is action or function oriented, whereas with C++ programming is object-oriented, with more emphasis or importance to data in relation to functions. Literally, on comparison basis C++ can be called data oriented while C or other structured languages called function oriented. Both types of languages data and functions are important but then it is the relative dominance of one over the other that decides the nature of the programming languages.

The basic unit of programming or code development in structured languages is functions while in OO languages it is objects that constitute the basic unit. Objects are instantiated

or created from classes. Thus, programming in an action or structure-oriented approach, programmers develop functions. Group of actions performing some task constitute functions. A program may involve many such functions and it is these functions that interact with each other to make up the program [overall solution to the initial problem definition]. As we have already seen that functions are given primary importance in structured languages and data exists with secondary importance to help functions perform their task.

One way of identifying functions given in a problem statement or system requirements specification document, assuming that English is the language of choice to express such specification documents, the VERBS (act of doing something or actions(s)) are indicatory of the possible functions of a solution for the problem on hand can have. C++ programmers develop classes. From the above interpretation, classes are also referred to as user- or programmer-defined data types. These classes will be used to create objects or instances of it. In fact, the structured programming syntax of `int a;` to declare `a`, variable of integer data type, `int` could be viewed as a class and `a` as an instance or object of class of `int`. The only difference is that objects will be created from programmer-defined data types rather than from predefined data types. Classes can be identified from a system specification by looking out for NOUNS [identifying entities].

5.2.1 Notion of a Class

In fact, the concept of classes in C++ is an extension of the structure concept in C. C structures which allow multiple predefined data types to be combined to form a new user-defined data type, but again functions never got associated with specific data types. A C++ structure does allow functions and data to be encapsulated. In this aspect, they are quite similar to classes, but they do differ and this shall be explored at a later stage. Classes enable the programmer to model objects or real world entities that have attributes (data members) and behaviours (functions). Such data types are created using the keyword class.

5.2.2 Member Functions

Member functions are also referred to as methods in some OO languages like JAVA. These member functions or methods are called responses to messages sent to an object. Once a class has been defined, it can be used to create or declare objects of it. We will develop a class called `EXAMPLE` that contains an integer data member and functions `initialize` (to initialize the object) and `increment` (to perform increment operation on the data members). The code for the above operation is as follows:

```
class EXAMPLE
{
private :  int data;
public:  void initialize (int val);
void increment ();
};
```

The name of our class is `EXAMPLE`. The class definition begins with the left brace and the definition ends with the close brace and a semicolon. Our example class contains a single data member (`int` data) purely for the purpose of differentiating objects. The class definition also

contains the prototypes for functions `initialize()` and `increment()`. The task of member function `initialize` is to initialize the data member (data) and member function `increment` increments the current value of data by 1.

5.2.3 Access Specifiers

The two keywords of private and public are referred to as access specifiers and are applied with data members and member functions of a class. Access specifiers are syntactic provisions to implement the data abstraction feature of OO languages. In fact, there are three access specifiers, namely private, public and protected. The protected access specifier will be explained in detail when we will discuss on Inheritance.

Access specifiers are nothing but the visibility or accessibility mode for the data members or member functions of the class. As a part of our earlier discussion, we have mentioned that the implementation details of the class must be hidden for the external world. The private access specifier makes the data members or member functions with which it is being applied accessible or visible from only within the class definition or within the {} of the class. These members are not accessible from outside the class definition.

In fact, the member accessibility operator is the dot operator and → for pointer members. That is to access a normal data member (say `int x`) of a class A whose object is `o1`, the syntax of `o1.x` is adopted. Member functions are as well accessed with the same syntax. Again if there is a member function `fn1()` in a class A, with private access specifier, it remains accessible from only within the class. Private members of a class are not accessible from outside the class, even via the objects.

Public members of a class are accessible from outside the class. Access specifiers are required for each and every data member or member function of the class. But when data members or member functions have similar access specifiers, all of them can be grouped together and the access specifier keyword be mentioned only once at the beginning. The access specifier keyword must be followed by a colon symbol.

Access specifiers need to get mentioned with each data member or member function only if its access specifiers differ from the preceding access specifier. That is, let us assume there are three data members and two member functions of a class Test, of which the data members must carry the private access specifier with them and the member functions should be public. This is done by the following declaration. Thus, members of a class with public access specifiers are accessible from outside the class.

The normal convention is to make data members of a class private and member functions of a class public. The reasoning behind this convention is that data members of a class being private, they will not be accidentally modified from outside the class. Such private data members of a class can be accessed either from within the class or from outside indirectly via the public member functions of the class. It is based on this that member functions which will be accessed from outside the class definition via the objects (and then indirectly access private data members) are declared as public member functions which will be accessed from only within the class or from outside the class within other public member functions of the class can be made as private. Such private member functions are also referred to as helper or utility functions in the fact that they help other public member functions of the class to achieve their task.

Member functions of a class can be defined within the class body itself in which case function prototypes within the class will be omitted. Member functions can also be defined outside the class definition, in which case the prototypes or declarations of such member functions will be provided within the class body or definition. Member functions defined in this manner outside the class body or definition still belongs to the scope of the class. The binary scope resolution operator (:) is used to resolve the scope of the member functions defined outside the class with the class.

5.2.4 Scope Resolution Operator

The scope resolution operator preceded by the class name is prefixed before the function header. The return type of the function precedes the entire header specification. As a part of the following example, we shall see the syntax of defining member functions. Member functions being defined outside the class definition with their scopes resolved using the :: operator apart from offering better code readability or clarity differ from the other syntax of defining member functions within the class definition in that such member functions define in that, such member functions defined within the class definition are automatically in lined. Whereas member functions defined outside the class with their scopes resolved using the binary scope resolution operator are not automatically in lined. Such member functions can be explicitly in lined by prepending the keyword inline before the respective function header. Let us now get back to the earlier class `EXAMPLE`. The class definition is as follows:

```
Class EXAMPLE {
private :  int data
public:  void initialize (int val);
void increment();
};
```

Note that we have mentioned the public access specifier only once. All member declarations or definitions follow. As we have already stated, an access specifier holds for the future declarations unless it is overridden by a new type of access specifier. If members of a class are to carry alternating or changing access specifiers, then the access specifier should be mentioned with each member declaration statement. Note that for function definitions within the class body access specifier is required to be mentioned once with the respective function header.

In our class `EXAMPLE`, we have two member functions `initialize()` to initialize the data member (data) of the `EXAMPLE` class, and function `increment()` to increment the data member. Private data members of a class cannot be initialized directly at the point of declaration with an assignments statement. This will cause syntax errors. Data members of the class can be initialized by public member functions of the class or by using a special function called constructor which shall be explained in detail in the next section. For the moment we shall use the function `initialize` to initialize the data members of the class to a valid value. The entire code for the above example is as shown in Figure 5.1.

5.2.5 Input/Output Statements in C++

Note that the statement in line is the display statement in C++. The statement `cout << "Invalid Initializer"<< endl` redirects the string `Invalid Initializer` onto the screen

```
1.  #include<iostream.h>
2.  class EXAMPLE
3.  {
4.  private: int data;
5.  public: void initialize (int val);
6.  void increment();
7.  };
8.  void EXAMPLE :: initialize (int val)
9.  {
10. if (val>=0)
11. data = val;
12. else
13. cout<<"Invalid Initializer value"<<endl;
14. }
15. void EXAMPLE:: increment ()
16. {
17. data = data + 1;
18. cout<<"Incremented Data Member is"<<data<< endl;
19. }
20. int main ()
21. {
22. EXAMPLE O1;
23. O1.initliaze();
24. O1.incrrement ();
25. return 0;
26. }
```

Figure 5.1: C++ programming example.

and is the display statement in C++, equivalent to the printf function in C. The operator `<<` is referred to as the stream insertion operator. When this statement is executed, the value to the right of the operator (in this case the string " ") or the right operand is inserted in the output stream. Normally the right operand is printed exactly as they appear between the double quotes.

However, the characters `\n,\a` are not printed on the screen. The backslash, escape sequence is a special character that achieves predefined I/O tasks (explained in chapter). The `endl` string is also called the stream manipulator. It outputs a newline on the screen, in this aspect it is similar to the `\n` escape sequence. It differs from the `\n` in that the `endl` manipulator after redirecting a newline on the screen flushes or clears the output buffer.

This simply means that on some systems where outputs accumulate in the machine in a buffer until there are enough to redirect it to the screen. Thus, the `endl` flushes the output buffer or redirects the entire string or output sequence that is accumulated. Getting back to the code, it is again within the function `main` where normally the classes are instantiated, and on this basis `main` is also referred to as the test deriver function—function that test derives the class. The statement creates an object `O1` of type class `EXAMPLE`.

The next two statements at lines 23 & 24 invoke the functions initialize and increment of class EXAMPLE with respect to the object O1 created in main. Note that the cout statement for display purposes in C++ can redirect a string or the value appearing after the insertion operator << onto the screen. Cout does not require the data types of values to be displayed as was in printf. That is why cout << "Incremented Data Member" << data << endl first displays the string within quotes, then displays the value of data and then prints a newline and flushes the output buffer.

With printf the format specifier of data that is %d for integer should be specified within the function, but there is no such compulsion in C++'s display statement. In fact, cout is not a function as was printf. Cout is an instance or object of the class iostream whose declarations are available in header file ostream.h. In fact, the other complementing input operation is performed in c++ with a cin >> data; statement to read the value of a from the keyboard or the standard input device.

Cin is an instance or object of class istream whose declarations are stored in the header file istream.h. The >> operator is called as the stream extraction operator and extracts input from the standard input devices to the input stream .most C++ programs often require both statements and it is common to include the header file iostream.h[input–output stream] which contains the declarations for the class iostream or IOS. These classes are created by combining the features of class ostream and istream or what is called as by inheriting the features of ostream and istream.

In fact to be precise iostream is created by multiple inheritance from classes ostream and istream. Inheritance is another key feature of oopS that enhances the reusability feature of the software. A class that will have features similar to another but that which also has its own features created by inheriting the other class for similar features and having its own explicit features separately created. The class from which features are inherited is referred to as the parent or base class and the class which inherits features is called the derived class. In the IOS case, the derived class is IOS or iostream and there are two parent or base classes, namely ostream and istream.

Inheritance is an important concept, we have provided a separate chapter for its discussion. For the moment, this understanding of Inheritance would suffice. As we have already said cout or cin is not a function. They are objects of the respective I/O class and << or >> are functions. They are functions that have been overloaded using the operator overloading feature of C++. Operator overloading again a key concept of OOPS will be explained in detail in later chapters. For the moment, one should be clear to interpret cin or cout statements as functions getting invoked with respect to objects cin our cout.

The operator <<, >> can also be cascaded in a single statement. That is to say cout << "Hai" << value; is allowed. In this, two redirections, one a string, and the other a value display are cascaded within a single statement. This proceeds well because at the end of each redirection or indirection, the overloaded << or >> function call would return to a reference to the object of the respective ostream or istream class. The remaining or cascaded operators are performed with respect to this returned object reference. The compiler parses the above statement as ((cout. << ("Hai")). << (value)).

At the end of this a reference to stream is returned and the remaining insertion operations are performed with respect to the returned reference. This is referred to as cascading or chaining of insertion operators. Similarly, even the extraction operators can be cascaded to the cout statement with the only difference that the reference returned is of type cin or a

reference to `cin` is returned after one indirection or extraction operation is performed. It is referred to as indirection, as values are fed or given in to the system and flow is from the external device into the system. In fact, `<<` and `>>` are overloaded operators in C++.

An operator is said to be overloaded if it has more than one meaning or operation associated with it. `<<`, `>>` when used with normal integers imply left shift and right shift bitwise operators. Thus, the `<<` and `>>` operators have two meanings associated with them, namely insertion or extraction and left shift or right shift operators. Such operators that have more than one meaning with them are called overloaded operators and is implemented using operator overloading. Overloading is implemented using functions [member or non-member] with the restriction that the function name would be the operator that is to be overloaded and preceded by the keyword `operator`. We will not get any further into operator overloading now but shall explore it in detail in chapter.

Note: Member functions are normally shorter than stand alone functions and functions of a non-object-oriented language. This is because the data are already available in the object as data members of the class and hence the member function calls often require no arguments or at least fewer arguments than a function call in a structured or no OO language or what are also called stand alone functions.

5.2.6 Class Declaration Syntax

Attempting to initialize a data member in the class definition is a syntax error. Classes can be instantiated as many times the client wants and there by create multiple objects. Memory allocation is done for multiple objects individually only for data members. The member functions of the class (object version) reside at a common memory location so that multiple objects can share this function code. That is memory allocation across multiple objects of the same class is done only for data members, whereas member functions are not duplicated in locations across multiple objects. Also, objects of the class share one common copy of the member functions of the class.

As we have already stated that member access specifiers need to be mentioned only if the current access specifier differs from the earlier specifier. And also members of a class [data or member functions] default to the private access specifier. That is, if the access specifier for a data member or member function is not explicitly specified in the class definition, then the compiler assumes the access specifier for the respective member to be type private. The best programming practice is to list all private members of the class first and then follow it up with protected members if any and then finally list the public members. With this practice one can avoid frequent repetitions of access specifier's keyword. Since an access specifier specified (or defaulted to private) is valid and applicable for all subsequent data members or member functions until it is overridden by another access specifier. For better understanding let us go through the following example:

```
class SAMPLE {
private:int a;
int b;
int c;
int d;
void fn1();
```

```
public:
.
.
.
};
```

5.2.7 Choice of Access Specifiers

Here, the private access specifier for the data member (`int a`) [though not a compulsion to be explicitly mentioned as all members default to the private mode] is applicable for all subsequent members till the prototype for member function `fn1`. Now we have a new access specifier public and this is applicable for all subsequent members until we have another specifier differing from the current one. As we have already stated readers are advised to group all private members [data or member functions] first, the followed by protected members and then finally the public members of the class. There is no requirement that all grouped private or protected public members of the class be placed between a pair of { } braces. The compiler automatically treats members of a class not overridden by a new access specifier differing from the earlier specifier as belonging to the same group.

5.2.8 Possible Coding Errors

Possible syntax errors with class definition is missing out the semicolon after the class definition. Applying size of operator on objects of a class reflects only the size of the data members. Member functions size is not reflected in the size of the object. This was explained earlier. The reasoning behind this is that it is data members that will remain unique and vary across objects. It is these data members that differ across objects and require explicit memory allocations. Hence it is only these data members memory consumption that is reflected in the size of operator application on an object.

5.3 Interfaces and Implementation

Separating interfaces from implementation is one of the fundamental issues of good software development. The public member functions of a class are also referred to as public services or public interfaces of the class. In fact, to be precise interfaces are the prototypes of member functions of a class with the public access specifier. Classes will be distributed to end users or clients. Thus end users or clients should be given only details they actually require and should not be overloaded with unnecessary distributions.

C++ is one language that promotes information hiding. The implementation details of a class are hidden from the end users or clients. Implementation details are nothing but function definitions, to be precise the member function definitions. These functions are not distributed to end users or clients. As was the case with C (wherein only `printf's` declaration is provided in the to be included header files], it is only the prototypes or declarations of the prototypes or declarations of these member functions that are distributed to clients.

5.3.1 Principle of Least Privilege

Prototypes of member functions are also referred to as interfaces or services. It is only these public interfaces that are distributed to end users. Thus, there is a need to separate interface from implementation. In fact, it is only the compiled object versions of an implementation that is distributed to end users. This is again nothing but the principle of least privilege, wherein end users have access to the interfaces and object versions of the implementation and not the source file. But then this is not with the intention of just hiding the source file. This syntax of separating interface from implementation avoids unnecessary recompilation of already compiled code and makes available to end users details that are required.

Thus, the accepted software engineering practice is to place the class declarations in header file to be included by clients who wish to use the class. This is the public interface and provides the clients with function prototypes for it to call the member functions. The member function's definitions are stored in a separate source file (`.cpp`) which forms the class's definitions. The end user or clients statements are normally defined in function `main()` in another source file (`.cpp`). The class declarations are made use of by including the respective user defined header file. Recoding our earlier EXAMPLE class to follow the principle of separating interface from implementation, the code looks as shown in Figure 5.2.

At the time of compilation the clients code in `main.cpp` is compiled and linked along with the member functions object code. Member function's source file is not compiled again unless it changes in definition. Clients interested in creating interfaces need to just compile the main program with the source code file (implementation) that generates respective object versions. Readers are advised to refer the programming language manual on how to compile multiple source files.

Note: Despite the fact that the public and private member access specifiers may be repeated and intermixed, list all the public members of a class first in one group and the list all the private members in another group. This syntax clearly emphasizes the class's interface. Keeping all data members of a class private and having public interfaces to modify these data members hides implementation details from the end users or clients promotes program modifiability and reduces bugs.

Defining member functions inside the class definitions automatically in lines the member functions. This can lead to performance improvements in terms of execution speed, but violates the principle of separating interface from implementation as a result of which if the implementation changes in definition the clients code must be recompiled. When interfaces are clearly separated from implementation and the implementation changes its function definition, provided the interfaces remain constant, the clients code need not be recompiled. It gets linked with the object version of the changed implementation, thus avoiding the unnecessary compilation time overhead.

5.3.2 Class vs. Structure

As a good coding practice only small member functions [in terms of lines of code] and stable member functions [those that are not susceptible to frequent changes] should be defined within the class definition, because change is a rule than a exception. C++ class construct is a natural evolution from the struct construct. But then struct construct is disadvantageous in that invalid values can be assigned to the members of a structure since the program has direct

```
"Example.h"
class Example
{
private : int data;
Public : void initialize(int val=0);
void increment();
};
"example.cpp"
#include "example.h"
void Example::initialize(int ini)
{
void EXAMPLE :: initialize (int val)
{
if (val>=0)
data = val;
else
cout << "Invalid Initializer value"<<endl;
}
void EXAMPLE:: increment ()
{
data = data + 1;
cout << "Incremented Data Member is"<<data<<endl;
}
"main.cpp"
int main ()
{
EXAMPLE O1;
O1.initliaze();
O1.increment ();
return 0;
}
```

Figure 5.2: C++ programming example (interfaces/implementation based).

access to data. Again, if the structure's implementation changes all programs that use this structure must also be changed since there is no interface available as we have with classes to modify data members and thereby ensure that data remains in consistent state. Other disadvantage with structures is that they cannot be compared in their entirety; they should be compared member by member.

5.4 Constructors and Destructors

In the earlier example(class EXAMPLE), we had two member functions of which one was to initialize the data member of the class. Also, C++ does not allow the data members to be initialized at the point of declaration. It is only via the member functions of the class that we set valid values for data members of the class after having read values for them. The member function initialize initialized the data member (data) of the class EXAMPLE.

But then for the object to be initialized, we require an explicit call by the programmer in the form of `objectname.initialize (5)`. C++ provides an automatic way of initializing objects [initializing the data members that constitute the object] thru constructors. Constructors are also member functions of the class. But they have a syntactic restriction that the member function name should be the same as that of the class name. In fact, the compiler distinguishes between normal member functions and constructor member functions based on the fact that constructors have the same name as that of the class name.

Constructor's very purpose is to set (initialize) data members of the class. Constructors are called automatically when an object of the class is created. Hence they require arguments passing mechanism to be available. Thus constructors similar to normal member functions can accept arguments. Constructors cannot have return types or values associated with them because constructors are expected to perform the object initialization task and not other processing (though syntactically allowed) should be performed within constructors. Thus there is no need to return values after a constructors task is over, for the initialization would have been performed.

The reasoning behind calling these member functions as constructors is that these member functions should construct the object (setting the data members to valid values) whenever a class is instantiated or an object of a class is created. A naive classification of constructors is default and user-or programmer-defined constructors. Default constructors are those that accept no arguments. Again they can be defined by the programmer. If not defined by the programmer, the compiler automatically creates a default constructor. But then with compiler created default constructors there is no surety that the data members of the object would be initialized with valid and programmer desired or expected initializers. Hence it is always advised to go in for programmer-defined default constructors.

Default constructors [either compiler or programmer] are called automatically when the client or end user who creates an object or instance of the class does not supply the initializer arguments at the point of creation like `EXAMPLE 01`. In such cases where object `01` is being created and there are no initializer arguments specified, the compiler calls (automatically) the programmer-defined default constructor, if there is one. Else the constructor automatically creates and class one.

Getting on with programmer-defined constructors, there can be more than one programmer-defined constructor for a class or constructors for a class can be overloaded. This is nothing but the concept of function overloading. For overloaded functions to be called correctly, the functions should differ in their signatures as we have explained already in the earlier chapters. For overloaded constructors to be called correctly, each constructor's signature should differ from the other. Let us now look at an example that makes use of both default and user-defined constructors. Consider the following class definition

```
"Test.h"
Class Test
{
private:  int a;
float b;
public:  Test ();
test (int,float);
display();
```

```
#include "test.h"
Test::Test()
{
a=0;
b=0.0;
}
Test:  Test(int v1,float v2)
{
a=(v1>=0)?v1:0;
b=(v2>=0)?v2:0;
}
};
```

In the above example, we have two constructors defined. The first `Test()` is the default constructor which sets the data members of the objects to valid starting point values. The second constructor is programmer-defined which accepts two arguments and after validation assigns the arguments passed to the data members of the objects. Note that we can exploit the syntax of defaulting arguments when defining constructors.

For example, in the above class `Test;` the user-defined constructor can be specified a prototype or declaration in `Test.h` as follows:

```
Test (int=0,float =0.0);
```

in which case if the values are not provided at the point of object creation, then the default values are assigned. Thus, the user-defined constructor also serves the purpose of a default constructor. There can be only one default constructor for a class. Thus, if a user-defined constructor is defaulting, its arguments then the default constructor should not be defined by the programmer, since there can be only one default constructor for the class.

Now looking at how constructors get invoked at the point of object creation [normally within the test driver function], within main function statement at line `(Test O1)`, creates an object `O1` and invokes the default constructor of `Test` class. This explanation is with our earlier syntax of defining a default constructor and a programmer-defined constructor that is not defaulting its arguments. However with the other syntax as well it remains the same. Thus the statement `Test O1` is equivalent to the call `O1.Test()` and the statement `Test O2(3,5.6)` invokes `O2.Test(3,5.6)`. Now with the second method of user-defined constructors defaulting its arguments or the user-defined constructor also serving as a default constructor, the statements:

Test O1: User-defined constructor called with all its arguments defaulted. Equivalent to the call `O1. Test (0, 0.0)`.

Test O2 (5): User-defined constructor called with one argument passed and the remaining defaulted. Equivalent to the call `O2.Test(5,0.0)`.

Test O3(5,5.2): User-defined constructor with none of its arguments defaulted, that is all arguments are passed at the point of object creation itself. Equivalent to the call `O3.Test (5, 5.2)`.

5.4.1 Destructors

Now getting on with destructors, as constructors are used to create or construct or build objects destructors are used to destruct or destroy objects. In terms of resources, the purpose or task of destructors is to reclaim memory that has been allocated for the members of the class when the program terminates and these members will not be used anymore. Destructors are also member functions and carry the same name as that of the class name, but they also have an additional tilde character or symbol (~) preceding the function header.

The reasoning behind this convention is that ~ symbol in computer terminologies associates with the negation or not operator or complement. Thus, it is the complement or negation of constructor and precisely the task of destructor is to destroy or reclaim memory or opposite of constructors. There can be only one destructor defined for a class. The destructor can be either programmer-defined or can be generated by the compiler on the programmers behalf in which case they are referred to as default destructors.

Whether it is programmer-defined or default destructors, they cannot accept arguments or return values. The reasoning behind not allowing arguments to be passed to destructors is that the purpose of destructors is memory reclaiming or what is called termination house keeping. Functions which perform memory reclaiming logic, there will not be any assignment or any other processing logic associated and hence there is no need to pass any arguments.

Destructors are not allowed to return values since they do not perform any processing logic other than memory reclaiming that can be passed on to the external world. Normally, destructors are defined by the programmer only if members are allocated memory dynamically or at run time [using new operator, which shall be explained further]. Members that have been allocated memory dynamically using new operator are reclaimed by applying the delete operator on the dynamically allocated data member within the destructor definition.

Most probably the dynamic allocation of memory using new operator would be performed within the constructor definition. Normally, destructors get called in the reverse order of constructor calls or that is to say the most recently constructed object is destructed first, then the next most recently constructed object and so on. For better understanding, let us consider the following example (the earlier declared `Test` class is being used here):

```
int main()
{
Test O1 (5,6.2);
Test O2;
Test O3(5);
return 0;
}
```

The order in which the constructors and destructors are called is as follows: First the constructor for O1 is called, followed by the constructor for O2 and then finally the constructor for object O3 is called. When the closing brace is encountered (the scope in which the object exists terminates), the destructor for O3 is called first, followed by the destructor for O2, and then finally the object O1 is destructed or destroyed.

Destructors are automatically called when the program terminates and the objects are no longer required to be retained in memory. Normally it is at the end of main (the close {brace}) definition that objects get destroyed or destructors get invoked. But then as we had the concept

of storage classes with variables determining the memory allocation or deallocation objects also have storage classes associated with them and this does change the order of destructors being called. Storage classes and objects shall be explained in the next section in detail.

For the moment, our assumption shall be that objects of a class are created only within the definition of function `main()` in which case the argument that objects get destroyed in the reverse order of construction holds true. As we have stated earlier it is only for dynamically allocated members that the programmer can reclaim memory within the destructor definition using delete operator [will be explained later].

Members that have been allocated memory statically at compile time or by the compiler are automatically reclaimed based on the scope and storage class. The reasoning behind is that programmer can reclaim memory only for those members for which one has allocated memory at run time dynamically and not for those members whose allocations have been done by the compiler statically at compile time. In the above Test example, the destructor declaration or prototype would be `~Test ()`. The definition of this function can contain a display to identify that the destructor of the class has been in fact called. Since there are no members that have been allocated memory dynamically, we do not have any memory reclaiming logic here in this case. However, programs involving dynamic allocations have destructors reclaiming memory [further examples will do so].

Note: Attempting to declare a return type for a constructor and returning a value from a constructor definition will cause syntax errors. Also attempting to pass arguments to destructor or returning value from destructor will cause syntax errors.

If there exists a member function of a class that is performing the operation of initializing the members of the class to valid values, then such a member function can be called within the constructor and thereby avoid repeating code and promote easier code maintenance. In fact, such member functions are normally made private since they are visible to public interfaces or member functions of the class.

It is these private member functions that help in other public member functions or interfaces of the class in achieving their task are referred to as helper or utility functions. Note that constructors and destructors are normally given public access specifiers because their very purpose of existence requires access to such functions from the outside world and hence the justification to provide public access specifier for constructors and destructors.

Declaring default values for arguments of a function in both the prototype [header file] and the member function definition will cause syntax errors. Overloading of destructors is not allowed. Let us now look at another example for better understanding of features discussed so far. The following program should compute the average percentage scored by a student in 8 subjects, code for which is shown in Figure 5.3.

In the above example, the `totalmarks()`, a private member function is the helper or utility that is made use of in computing the average percentage scored by a student in 8 subjects. With the assumption that the logic involved in computation of average is naive, let us now proceed to the next section that elaborates on class member accessibility.

5.5 Controlled Access to Class Members

The member access specifiers public, protected and private in the ascending order of secured access are used to provide controlled access to data members and member functions of a

```
"student.h"
Class Student {
int totalmarks();
int marks[8];
public: Student();
void readmarks();
void setmarks(int,int);
void print();
};
"student.cpp"
#include<iostream.h>
#include"student.h" Student::Student ()
{
for (int i =0;i<8; i++)
marks[i]=0;
}
void Student:: readmarks()
{
int mark;
for (int i =0;i<8; i++)
{
cout <<"Enter mark for subject"<<i;
cin >>mark;
setmarks(i,mark);
}
}
void Student:: setmarks(int subj,int mark)
{
if (mark>=0)
marks[subj]=mark
else
cout <<"Invalid Marks Entry \n";
}
void Student::print()
{
int localtotal = totalmarks();
cout<<"Rounded off average is "<<localtotal/8;
}
int Student::totalmarks()
{
int ltotal =0;
for (int i =0;i < 8; i++)
ltotal + = marks[i];
return ltotal;
}
int main ()
{
Student s1;
s1.readmarks();
s1.print();
return 0;
}
```

Figure 5.3: C++ code for average computation.

class. The default access mode for all class members is private, so that all members after the class definition beginning and before the first access specifier are private. A member's access specifier as we have seen is applicable until the next differing member access specifier or till the end of the class definition [close brace {}].

Intermixing of member access specifiers across statements is allowed but can be quite confusing. Hence it is advised to group all public members first, followed up by protected members and then finally the private members. A class's private members can be accessed only by member functions and friend [will be explained later]. The public members of a class can be accessed from outside or within any function in the program. Public members provide the users of a class an overall view about the class's services or behaviours available. This set of services forms the public interface of the class.

The private members of a class are not accessible to clients. This forms the implementation. A non-member or non-friend function trying to access a private member of a class causes syntax error. The reason behind mentioning the public members of a class at the first is to provide the clients [who will be given this class declaration header file] a clear view of the interfaces that are available for his or her se. Data members made private and member functions public facilitate easier debugging because data manipulation errors are now localized to within the member functions of the class or friends of the class.

Since non-member functions' or non-friend functions' first place do not have any sort of syntactic access to private data members of a class and hence no chance of any data manipulations within non-member or non-friend functions. Access to a class's private data is done carefully by providing appropriate set and get functions that read values for private data from user and set the data members to the read in values after validation on the values read in to ensure the consistency and integrity of the data members.

Private member functions of a class that shall be invoked by other public member or friend functions of the class are referred to as helper or access or utility, or predicate functions. Predicate naming is based on the fact that such helper or utility function that performs simple Boolean conditions checking of the form that results in true or false values. An example would be an `isempty()` helper or predicate function made available within a class Stack to avoid popping when the stack is empty. Thus such private member functions exist purely to help the major public member functions or friends of the class in achieving their overall task.

The earlier example of students uses a utility function called totalmarks that computes the average of marks scored by a student. Such functions are also called accessor methods. Based on the task that they perform, they can be categorized as read or get member functions that just read and return private data values. Set or write member functions that modify the private data values, member functions such as constructors that perform initialization task and other conversion functions as we will see later that convert objects of onto type to another. In fact, such member functions are called conversion constructors and also the helper or utility functions that help the public member functions in achieving their task are referred to as predicate functions.

5.6 Scope of Class Members

The data members [variables declared in the class definitions] and member functions [prototypes declaration] of a class have class scope. Within class scope all the members of the class

are accessible by all the member functions following it and can be directly referenced by name. Member functions can be overloaded by other member functions of the class. Variables within member function definition have function scope [known only within the function].

A member function defining a variable with the same name as that of a data member name, the function scope variables overwrites the data member or class scope variable within the function definition. Such hidden class scope variables can be accessed within function definition by making use of binary scope resolution operator and prefixing the class name before it. Hidden global variables are accessed by making use of unary scope resolution operator. The syntax for the above two operations is as follows:

```
Classname::data member/variable name;
::globalvariablenames
```

Normally, it is the dot operator that is used with object name or a reference to the object to access the object members. A pointer to an object accesses the data members of the object using the arrow operator. These operators are often referred to as the dot and arrow member selection operators, these operators performing the job of selecting members of an object. The same syntax is also used to select members of a structure.

5.7 Order of Constructors or Destructor Calls

Constructors and destructors that perform the task of initializing and destructing the objects are called automatically by the compiler at the point of object creation and program termination when the object is no longer required to be in memory. As we said storage classes does determine the order in which these member functions calls are made, based on the scope in which the class is instantiate or objects of the class are created. Although normally the destructors are invoked in the reverse order of construction, storage classes of objects can alter this ordering of calls.

The following rules hold true:

Objects that exist in global scope have their constructors called first before any other function's (inclusive of main) execution begins. But then no ordering can be guaranteed with respect to constructor calls among global objects [more than one global object existing]. The destructors for such a global object is called when `main()` ends or `exit()` function is called. In case if the program terminates abnormally, i.e. by calling `abort()`, destructors for global objects are not called.

5.7.1 Automatic Local Objects

Constructors are called each time when execution reaches the point where such objects are created. Similar to automatic variables their corresponding destructors are called every time when the scope or block in which subjects are created or exist is exited. Destructors for local objects are not called if either of `exit()` or `abort()` call is made.

5.7.2 Static Local Objects

Constructors are called only once at the beginning when execution or control reaches the point of object creation. Similarly destructors are called when `main()` ends or `exit()` is called.

Abnormal program termination with calls to `abort()`, static local object's destructors are not called. Let us now go through a simple example similar to the one in chapter . We have a class called `EXAMPLE`, containing a data member `i` to distinguish objects from one another. The constructors and destructors of the class are defined to have corresponding display statements with the object number, purely for the purpose of identifying the ordering of constructor and destructor calls. The code looks as shown in Figure 5.4.

```
"example.h"
Class EXAMPLE
{
private : int i;
public: EXAMPLE(int = 0);
~EXAMPLE();
};
"example.cpp"
EXAMPLE::EXAMPLE(void)
{
if (val>=0)
i = val;
else
cout<< "Invalid intialiser";
cout<< "constructor called for object" <<i<<endl;
};
EXAMPLE::~EXAMPLE(void)
{
cout << "Destructor called for object" <<i<<endl;
};
"main.cpp"
#include "example.h"
void standalonefunction(void);
EXAMPLE one(1); //global object
int main()
{
EXAMPLE two(2); // Auto local object
Static EXAMPLE three(3); //static local object in main
Standalonefunction(); //standalone function that again creates objects
of class EXAMPLE;
EXAMPLE four(4);
return 0;
};
void standalonefunction()
{
EXAMPLE five(5); //automatic object local to function stanalonefunction
Static EXAMPLE six(6);// Static object local to function
stanalonefunction
EXAMPLE seven(7); //automatic object local to function standalonefunction
}
```

Figure 5.4: C++ code illustrating constructor/destructor calls.

The output of the above code is shown in Figure 5.5. The order in which the constructor and destructor calls are as shown in the above output.

```
Constructor called for object 1
Constructor called for object 2
Constructor called for object 3
Constructor called for object 5
Constructor called for object 6
Constructor called for object 7
Destructor called for object 7
Destructor called for object 5
Constructor called for object 4
Destructor called for object 4
```

Figure 5.5: Output for code in Figure 5.4.

Note: The assignment operator when used with objects performs default member wise copy, where each member of the RHS object is assigned individually to the same member of the LHS object. This is similar to the application of = operator on structure variables.

Objects should be passed to functions as constant references and thereby avoid the duplicating overhead associated with pass by value and the accidental modification or side effect overhead associated with pass by reference. Constant reference is the best way of passing objects to functions.

5.7.3 A Subtle Trap

Returning a reference to a private data member: As we have already seen a reference to an object is an alias for the name of the object and hence can be used LHS of an assignment statement. Thus, the reference makes a perfectly acceptable lvalue that can receive a value. But then this can be exploited by having a public member function of a class returning a non-constant reference to a private data member of that class. Let us consider the following example for better understanding of these concepts, code for which is shown in Figure 5.6. Note that set value that returns a reference to one of its private data members can be modified from outside the class using an assignment statement or as a part of an assignment statement as an `lvalue`.

5.8 Constant Member Functions and Objects

The principle of least privilege is one of the key features of object-oriented programming. The constant qualifier to a great extent is helpful in implementing the principle of least privilege, which is one of the most fundamental principles of good software engineering. Certain objects require being modifiable while certain other objects require only read access. An object that is declared to be constant and further attempts to modify it is a syntax error. Declaring variables and objects as constant can improve performance since constants are interpreted in optimized forms by certain compilers.

Only constant member functions can be invoked on constant objects. Constant member functions cannot modify the objects state or to be precise, they cannot modify the data

```
class Test {
public: Test (int =0, int =0);
void setvalue(int,int);
int getval();
int &set value(int);
private : int a;
int b;
};
Test :: Test (int v1,int v2)
{
a=v1;
b=v2;
}
int & setvalue (int v3)
{
a = v3;
return a;
}
int main()
{
Test O1;
int &aref=O1.setvalue(50);
aref=30;//undesired modification
O1.sevalue(50)=70;//returned reference used as lvalue
return 0;
}
```

Figure 5.6: Member functions returning references.

members of the objects. The **constant** keyword is mentioned in both functions prototype and definition after the function header. Member functions that require only read permission can be declared to be as constant functions. The **constant** keyword should be mentioned at the end of both function declaration and definition.

Note: A member function that has been declared to as constant and modifies the object state or its data members is a syntax error. Also, invoking non-constant member function on a constant object is a syntax error. As normal functions can be overloaded, even non-constant member functions can be overloaded with their non-constant counterparts. The compiler resolves overloaded member function calls based on the objects nature, since only constant member functions can be invoked on constant objects.

The only exception with constant objects with respect to constant member functions being called on them is that constructors and destructors of constant objects cannot be declared as constant since constructor's very purpose is to initialize objects data members and destructors purpose is to perform termination clean up; both of which require write or modification permissions. Thus constant declaration is not allowed for constant objects. Thus even though constructors or destructors are non-constant member functions, they can still be invoked on constant objects. Possible combinations of member functions type and object types are as shown in Table 5.1.

Table 5.1: Member function types and objects

Object type	*Member function type*	*Allowed/disallowed*
Constant	Constant	Allowed
Constant	Non-constant	Disallowed
Non-constant	Constant	Allowed
Non-constant	Non-constant	Allowed

Readers need to differentiate the fact that with constant objects [data members cannot be modified] while with constant functions no modifying logic can be specified within its definition. Only read type operation or code can be specified within the function definition of constant functions. Constant data members of a class cannot be initialized using an assignment statement. Constant members of a class are initialized using member initializer syntax.

Non-constant data members can also initialized using the member initializer syntax. In case there are multiple constant class members, the member initializer list is comma separate. Composition is another feature of OOPS, wherein objects of other classes are by themselves members of a class or objects of classes are contained or composed within a class definition also makes use of the member initializer syntax to initialize the composed object data members, [indirectly or actually] calling the constructor of the contained - composed object. Let us now look at a programming example to understand the member initializer syntax.

```
class Test {
private :  int data;
const int value;
public:  Test (int=0,int=1);
void addval()  data + = value;
void display () const;
};
Test::Test (int d,int v):value(v)
{
data = d;
}
void Test::display() const
cout << "data is" << data << "Value is" << value << endl;
}
```

Note that in the above code the constant value data member is initialized in the constructor header using a member initializer operator (: colon). Trying to initialize a constant data member of a class using an assignment statement is a syntax error.

Not providing member initializer syntax for constant data member is also a syntax error. All member functions that do not modify the object should be declared as constant. This allows such functions to be invoked on constant objects and also prevents any object modifying (accidental side effect) statement within the definition. Attempts to do so will be caught at compilation stage itself.

5.9 Composition—Objects as Members of Classes

Composition is another key feature of OOP wherein a class has objects of other classes as its members. Realistically, the software engineering observation has been that end users or clients require objects of already defined classes as its members. For better understanding, let us consider the following example. An organization is wishing to create employee information details. Details of employees include first name, last name, hire data and birth date. We shall incorporate composition concept of OOP to achieve the task. We shall have two classes employee and date whose data members are as follows: **Date : month, day** and **year**—all of type integers. **Employee: fname, lname, Birth_Date, Join_Date**. The declaration of class **Date** is as shown in Figure 5.7.

```
"date.h"
class Date {
private : int month;
int day;
int year;
int check day(int);
public: Date (int =1,int =1,int=2003);
void display() const;
~Date() {cout<<"Destructor for date object"<<endl; }
};
"date.cpp"
#include"date.h"
Date::Date (it m,int y, int d)
{
cout<<"constructor for date object"<<endl;
if(m>= 0 && m<=12)
month = m;
else
month =1;
year =( (y>1900 && y<2075)?y:0);
day = checkday(d);
}
int Date:: checkdate(int today)
{
static constint daysofmonth[13]=0,31,28,31,30,31,30,31,31,30,31,30,31;
if(today >0 && today <=daysofmonth[moth])
return today;
if(month ==2 && today ==29 && (year % 400 ==0) ||( year % 4==0 && year
% 100 !=0))
return today;
count <<"Invalid Day Value"<<endl;
return 1;
}
"employee.h"
#include "date.h"
class Employee {
private : char fname [25];
```

Figure 5.7: Continued

```
char lname [25];
const Date Birth Date;
const Date Join Date;
public: Employee(char *, char *, int,int,int,int,int,int);
void display() const;
~Employee();
};
"employee.cpp"
#include"employee.h"
#include"date.h"
Employee::Employee (char *fn,char *ln,int bm,int bd,int by,int jd,int
jm,int
jy):Birth Date(bm,bd,by),Join Date(jm,jd,jy)
{
int length = strlen(fn);
strncpy(fname,fn,length);
fname[length]='\0';
Continued on Next Page
5.9. COMPOSITION-OBJECTS AS MEMBERS OF CLASSES 141
int length = strlen(ln);
strncpy(lname,ln,length);
lname[length]='\0';
cout << "constructor for employee object" <<fname<<lname<<endl;
}
void Employee::display () const
{
cout <<lastname<<firstname;
Join Date.print();
Bith Date.print();
cout<<endl;
}
Employee::~Employee()
{
cout<< "destructor called for employee" <<fname<<lname<<endl;
}
"main.cpp"
#include"employee.h"
#include<iostream.h>
int main()
{
Employee e1("Siva","Selvan",23,7,1978,16,7,2003);
e1.display();
}
```

Figure 5.7: C++ programming example for class composition.

When an object is created its constructor is called automatically. In cases of classes con
ing or containing objects of other class care should be taken to pass arguments to me
object constructors and to invoke member object constructors properly. Member objec
constructed in the order in which they are declared within the class definition and not
order in which they are listed in the constructor's member intializer list. In fact, th
structors of the member objects are invoked before the composing object or containing
object constructor.

Observe that the constructor of the enclosing class (employee) accepts arguments inclusive of the member objects constructors. In case of the member initializer list missing, the default constructors of the member objects are invoked. Again if the default constructors are not provided and the member initializer list is missing as well, this leads to syntax error. A class that has as its member another object with public access specifier encapsulation and hiding concepts of that composed objects private members remain intact. The output of the above example illustrates the order in which constructors and destructors calls are made.

5.10 this Pointer

Every object has access to its own address through a pointer called `this` pointer. The `this` pointer is not reflected in the size of operation on the object since the `this` pointer. The `this` pointer is passed onto the object implicitly by the compiler every time a non-static member function call is made to the object.

The `this` pointer can be used to reference both data members and member functions of an object. It can be used both implicitly as well as explicitly. The type of `this` pointer depends on type of object and the member Function where `this` pointer is referenced is a constant function or not. In a non-constant member function of class `Test`, the `this` pointer is of type `Test * const`. In a constant member function of the class `Test`, the type of the `this` pointer is const `Test * const`. Every non-static member function has access to `this` pointer to the object for which the member function is being invoked. Let us now look at a simple example for the understanding of the concepts explored with respect to the `this` pointer, code for which is shown in Figure 5.8.

```
class Test
{
private : int x;
Test (int =0);
void printf() const;
};
Test :: Test (int a) {x = a;}
void Test::print () cosnt
{
Cout << x << this→x<<(*this).x;
}
int main ()
{
Test O1(2);
O1.print();
return 0;
}
```

Figure 5.8: C++ code to illustrate the use of `this` pointer.

The parenthesizing in line is required because of operator precedence rules. If the expression is not parenthesized, since . carries higher precedence over *, the expression would be interpreted as `*(this.x)`, however, the dot operator cannot be used with a pointer and hence it leads to a syntax error.

Two main applications of `this` pointers are to enable cascaded member function class and prevent object self assignment as will be explained in the later sections. In fact, the cascaded member function call will be explained in the following section.

5.11 Cascaded Member Function Calls

Cascading of member function calls is nothing but the process of invoking several member functions of a class with respect to a single object and in a single line statement of the form `o1.input ().output ().exit ()`. Here `input`, `output` and `exit` are three member functions of some class and should get executed or invoked with object `o1` of the respective class. This feature is also called chaining of member function calls. Let us now look at an example that implements the cascading of member functions, code for which is shown in Figure 5.9.

```
class Test {
private: int mem1;
int mem2;
int mem3;
public: Test (int =0,int=0,int=0);
Test & setmember1(int);
Test & setmember2(int);
Test & setmember3(int);
Test & setall(int,int,int);
};
Test::Test (int v1,int v2,int v3)
{
setall (v1,v2,v3);
}
Test & Test::setall(int val1,int val2,int val3)
{
setmember1(val1);
setmember2(val2);
setmember3(val3);
}
return *this;
}
Test & setmember1(int val1)
{
mem1=val1> 0?val1:0;
return * this;
}
Test & setmember2(int val2)
{
mem2=val2> 0?val2:0;
return * this;
}
Test & setmember3(int val3)
{
mem3=val3> 0?val3:0;
return * this;
}
int main ()
{
Test O1,O2;
O1.setmember1(5);
O1.setmember2(10);
O1.setmember3(15);
O2.setmember1(20).setmember2(25).setmember3(30);
return 0;
}
```

Figure 5.9: C++ code supporting cascaded member function calls.

The above example has three integer data members and four public interfaces, three of which set the individual data members and one among them (int,int,int) sets all the three data members at one stretch. All the four interfaces are scheduled to return a reference to class Test, so as to support cascading.

The cascaded member function call in line of main() is parsed left to right (since the dot operator associates left to right). First, the member function setmember1 is executed with respect to object O2. At the end of its call it returns the this pointer or the address of the object with respect to which the member function was invoked. Thus, the next function call is executed with respect to this returned address. Thus, this pointers have a crucial role to play in supporting cascading of member function. We have just illustrated with one instance of cascaded member function calls. Readers are advised to try all possible combinations of calls to setmember1, setmember2, setmember3 and set all in a cascaded fashion. For better understanding of the cascaded member function call, the parsing syntax is shown below

```
(((O2.setmember1(20)).setmember2(25)).setmember3(30)); 1 2 3
```

The parentheses and the numbers below them indicated the order in which the entire cascaded call is processed. At the end of each call, as we have said earlier the address of the object with respect to which the member function was called is returned. Thus, the object with respect to which the remaining function calls should proceed is available. This avoids repetition of object name with each function call(s).

5.12 Dynamic Memory Allocation

C vs. C++

The new and delete Operators available in C++ are used to allocated memory dynamically at run-time. C's mechanism of allocating memory dynamically is using the malloc and free function calls. Static allocation of memory [at compiler time] may cause wastage of memory space. The statements in C to allocate memory are as follows:

```
int * eptr;
eptr= malloc (sizeof(int));
```

The above statement dynamically creates an object of the type int. The corresponding statement to release the memory allocated using malloc is free(eptr); or in general free(ptrname). Memory allocation in C requires an explicit call of function malloc and within it use of the size of operator. Certain early versions of C would also require the pointer returned by malloc to be casted to the type of ptr on the LHS side of the assignment statement. And also the allocated block of memory using malloc call is not initialized for which calloc is to be used. In C++, the equivalent statement to allocate memory dynamically for the above type is eptr=new int; assuming that eptr has been suitably declared to be a pointer to an integer.

The keyword new is an operator that automatically creates an object of the correct size, then automatically calls the constructor for the object and returns a pointer of the type for which memory has been created. In cases of memory allocation being most successful, the new operator automatically throws an exception indicating the failure of memory allocation. Exception handling another feature of OOP can be used to handle exceptions as we will discuss

in the chapter - on Exception Handling. To reclaim memory or destroy the object for which memory has been allocated using `new` operator, the `delete` operator is used as `delete eptr`. C++ allows initializers to be provided for a newly created object as follows:

```
Double * eptr = new double (9.67);
```

Array declaration (dynamic allocation) in C++ is as follows:

```
int * a = new int [20];
```

Reclaiming the space allocated for `a` is by `delete [ ] a`. The main advantage of using `new` and `delete` operators is that the constructors and destructors of the class get called automatically. However, `new` and `delete` operators when mixed with `malloc/free` function calls cases logical error. Space that has been created by `malloc/new` must be released by `delete/free` operator only. Also an array that has allocated space dynamically should be deleted using `delete []`. Application of just delete on an array causes logical error.

Memory created as an array should be deleted with `delete []` operator and memory created for an individual element should be deleted with the `delete` operator. After our discussion on static members of a class, we shall develop a program that involves dynamic memory allocation. The application of size of operator on an object reflects only the size occupied by the data members of the object and not the member functions size. This was because only the data members uniquely distinguish objects and require explicitly memory allocations. Thus, all data members are allocated separate memory allocations for each instance of the class. One exception from this is static class members.

Static members of a class are not allocated individual memory locations for each object but they are shared across objects of the class. There is only one copy of static members that is maintained in memory. In fact, static members are class-wide property or information and not object-specific or object-wide. Static members are created on a per class basis rather than on a per object-basis. The declaration of static class members begins with the keyword `static`. Thus, static members are a property of the class and not a property of the specific objects of the class. Static members have class scope. The access specifiers of public or protected or private are applicable with static members as well.

A class's public static members can be accessed through any objects of the class or when no objects of the class are existing. They can be accessed using the class name and the binary scope resolution operator. A class's private or protected static members can be accessed only through public member function or through public member function or friends of the class. To access such members when no objects of the class are existing, a public static member function must be provided and this function must be called with the class name and the scope resolution operator when objects are existing, a static member can be referenced via any of the objects of the class. All reference refers to the shared copy of the static member. The student example shall be modified to allocated space dynamically and count the number of students that are created at any point of time, code for which is shown in Figure 5.10.

The count data member which is a static class should reflect the total number of student objects (students) that have been created. Each time an object of class student is created (at which point constructor of the student class is called), the count should be incremented. Hence this incrementing logic is placed in the constructor definition so that each time an object is created the static (common across objects) member count is suitably incremented. Since count is a private static data member, a static function called `readcount ()` that returns the value of count is provided.

```
"student.h"
class Student {
private: char * firstname;
char * lastname;
static int count;
public: Student(const char *, const char *);
~Student();
static int readcount();
void print() const;
};
"student.cpp"
#include "student.h"
int Student::count = 0;
int Student :: readcount () {return count ; }
Student :: Student (const char * fn, const char * ln)
{
firstname = new char [ strlen (fn) + 1 ];
strcpy(firstname,fn);
lastname = new char [ strlen (ln) + 1 ];
strcpy(lastname,ln);
++count;
cout << "Constructor called for"<< firstname << lastname;
}
Student :: ~ Student ()
{
cout <<"Destructor being called for"<<firstname<<lastname;
cout<<endl;
delete [] firstname;
delete [] lastname;
- -count;
}
"main.cpp"
int main ()
{
cout<<"Count of students before any student object is created \n";
cout << Student :: readcount;
Student * s1 = new Student ("Siva","Selvan");
Student * s2 = new Student("Ganesh","Ramani");
cout<<"Count of students after student objects have been created \n";
cout<< s1→readcount();
cout<<"Student 1 details \n";
s1→print();
cout<<"Student 2 details \n";
s2→print();
delete s1;
delete s2;
s1=0;
s2=0;
cout<<"Count of students after student objects have been deleted \n";
cout << Student :: readcount();
}
void Student : printf () const
{
cout <<"First name is" << firstname <<endl;
cout <<"Last name is" << lastname <<endl;
}
```

Figure 5.10: C++ code illustrating the use of **new** operator.

Note that when there are no objects of the class student existing, the static member is referred to using the class name and the scope resolution operator. When objects of the class are existing, static data members can be referred to using one of the objects or using the class name with the :: operator. Member functions can be declared to be static if only they do not refer to non-static class members. As we have already stated, static member functions do not have `this` pointer associated with them because static data members and static member functions exist independent of the objects of the class. Note that static members [data or functions] of a class exist even when there are no objects of the class existing and can be used or referred to.

5.13 Friends

The dictionary or real time meaning of the word friend is somebody who can be trusted or confided in (expressing confidence). It is based on this trust that we have the concept of friends supported in C++. The access specifiers private or public or protected provided varying levels of access to data member or member functions to the external or the outside world. As it is only public members of a class can be accessed from outside.

Protected members of a class can be accessed from outside the class but only by derived class objects. Private members which provide the peak secured access can be accessed from only within the class or from outside via public member functions of the class. One exception to this is friends. Friends can access private members of a class directly from the external world or outside the class. A function or an entire class can be made a friend to another class. Friend function's extensive use is in operator overloading which will be explored in the next chapter. The keyword `friend` is used to declare a class or a function as a friend of another class. For example, to declare a function `f1()` or a class B as a friend of class A, the following syntax shown in Table 5.2 should be adopted.

Table 5.2: Declaration syntax for friends

`Class A`
`{`
`friend class B;`
`};`
	`Class A`
	`{`
	`friend void f1();`
	`};`

The class or function which is to act as a friend of a class, the corresponding function or the class which is to act as a friend to a class should be included in the class definition only [header file] with the keyword `friend` preceding the function or class as done in the above statement. The first code declares class B to be a friend of class A and the second code declares function `f1()` to be a friend of class A.

Note friends are neither data members or member functions of the class. It is just expressing at most confidence in providing access to all its members irrespective of the access specifier to the external world. Since access specifiers are not applicable with friends, friend declarations can be placed anywhere in the class definition. Friendship should be explicitly granted by a class and cannot be assumed or taken for granted.

For a class y to be a friend of class x, the class x should explicitly declare that y is a friend of it as was done in the example. An example involving friend functions, where the private

data member of a class is accessed and modified by a friend function is shown in Figure 5.11. Note how the data member data of Example object `exmp` is set and accessed from outside the class by non-member function `fn1`. This is possible only because the class has declared the respective function to be a friend and hence its private members are accessible to it.

```
#include<iostream.h>
class Example
{
friend void fn1(Example &,int);
void display()
{
cout<<"Data Member Value is"<<data;
}
public:Example()
{
data=0;
}
private:
int data;
};
void fn1(Example &e,int d)
{
e.data=d; //this is allowed as a result of friends feature
}
int main()
{
Example exmp;
exmp.display();
fn1(exmp,5);//new value for data member set by a nonmember
    but friend function.
exmp.display();
return 0;
}
```

Figure 5.11: C++ code to illustrate `friend` functions.

Friendship is neither symmetric nor transitive. If a class x is a friend of class y, y cannot be assumed to be a friend of class x for which class x should explicitly declare that y is a friend of it. On similar lines, if a class x is a friend of class y and y is a friend of class z, transitive inference that x is a friend of z is not valid. Such properties of binary relations does not hold true with friendship relationship. Overloaded functions can as well be declared or made friends to other classes.

Review Questions

1. List the various key object oriented programming features.
2. Explain the term data encapsulation and highlight the same with respect to the basic unit of programming in an object oriented language such as C++.

3. Justify the reasoning behind the generally used access specifiers of private for data members and public for member functions of a C++ class.
4. What is the use of the scope resolution operator? Can member functions be defined inside the class definition? Explain.
5. Differentiate the newline escape sequence from endl stream manipulator.
6. Develop a C++ program to generate the factorial of a user-specified number. Use C++ style input and output statements (use of classes is not required).
7. Develop a C++ program to matrix manipulations of addition, subtraction and multiplication. Use classes.
8. Define a constructor and identify the general responsibilities of a constructor for a C++ class.
9. Develop a C++ program using classes that implements the sorting algorithms of insertion, bubble and selection sort.
10. Explain the use of `this` pointer in C++.
11. Explain class composition and differentiate it from data encapsulation.
12. Define dynamic memory allocation and allocate an integer array of size 30 dynamically using `malloc` and new features of C and C++. Appreciate the limitations of `malloc` in comparison to the benefits offered by new operator.
13. Discuss the need for friend functions in C++ with an example.
14. Develop a Class Time package in C++ and support functions to convert time expressed in 24 hours format to 12 hour format (assume data members of hour, minute and second respectively).
15. Develop a C++ Calendar package that displays the calendar for a given month input. Also support a function to display the day given the date of the month.