UNIT II

What is Function?

Function is a group of logically related statements that is used to perform specific task.
Function offers several advantages to the programmer:
There is no need to write same block of statements again and again. Thus it will reduce the length of the program.
Error handling is easy because we have to check only function body instead of same block of statement again and again.

main () Function in C++

The main function is the starting point for the execution of the program. The execution of every c++ program starts from main function.
The general syntax of main function in c++ is given below:
int main ()
Here,
main function returns the value of type integer while in C main function does not return any value. Hence main function in C++ returns value of type integer we must have to use return statement at the end of main function as shown below:

```
int main()
{
Block of statements
return 0;
}
```

If you don't specify the return type for main function then default return type is considered of type integer.

Function Prototype/Function Declaration

The declaration of function in the program is known as Function Prototype. The general syntax of function prototype is given below:
Return-type Function_ Name (Argument_List);

Example:

```
int sum (int a, int b);
```

Function Prototype provides following information to the compiler:

(1) Name of the Function

(2) Return Type of the Function
(3) Number of arguments and their Data Type
Whenever compiler founds any function call statement, first it will check function prototype to ensure following things:
(1) Weather the function that is called is declared or not.
(2) Weather proper number of arguments is passed while calling the function or not.
(3) Weather the data type of the arguments that are passed is corresponds to the arguments specified in the function prototype or not.
(4) If the function returns any value then it is corresponds to the return type specified in the function prototype or not.
If all the above mentioned criteria are satisfied then control of the program is transferred to the function definition otherwise compiler will generate error.

While declaring the function you should keep following points in the mind:
(1) You must specify data types for each argument separately.

```
Example:
int sum (int a, int b); // valid
int sum (int a, b); // Not Valid
```

(2) It is not compulsory to specify name of the arguments in the function declaration.

```
Example:
int sum (int, int); // valid
```

(3)If function does not accept any argument then you can leave the parenthesis empty.

```
Example:
int sum (); // valid
```

Types of User-defined Functions in C++

In this tutorial, you will learn about different approaches you can take to solve a single problem using functions. For better understanding of arguments and return in functions, user-defined functions can be categorised as:

- Function with no argument and no return value
- Function with no argument but return value
- Function with argument but no return value
- Function with argument and return value

Consider a situation in which you have to check prime number. This problem is solved below by making user-defined function in 4 different ways as mentioned above.

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

void prime();

int main()
{
   // No argument is passed to prime()
   prime();
   return 0;
}


// Return type of function is void because value is not returned.
void prime()
{

   int num, i, flag = 0;

   cout << "Enter a positive integer enter to check: ";
   cin >> num;

   for(i = 2; i <= num/2; ++i)
   {
      if(num % i == 0)
      {
         flag = 1;
         break;
      }
   }

   if (flag == 1)
   {
```

```
        cout << num << " is not a prime number.";
    }
    else
    {
        cout << num << " is a prime number.";
    }
}
```

In the above program, `prime()` is called from the main() with no arguments.

`prime()` takes the positive number from the user and checks whether the number is a prime number or not.

Since, return type of `prime()` is `void`, no value is returned from the function.

Example 2: No arguments passed but a return value

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

int prime();

int main()
{
    int num, i, flag = 0;

    // No argument is passed to prime()
    num = prime();
    for (i = 2; i <= num/2; ++i)
    {
        if (num%i == 0)
        {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
    {
        cout<<num<<" is not a prime number.";
    }
    else
    {
        cout<<num<<" is a prime number.";
    }
    return 0;
```

```
}

// Return type of function is int
int prime()
{
   int n;

   printf("Enter a positive integer to check: ");
   cin >> n;

   return n;
}
```

In the above program, `prime()` function is called from the `main()` with no arguments. `prime()` takes a positive integer from the user. Since, return type of the function is an `int`, it returns the inputted number from the user back to the calling `main()` function. Then, whether the number is prime or not is checked in the main() itself and printed onto the screen.

Example 3: Arguments passed but no return value

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

void prime(int n);

int main()
{
   int num;
   cout << "Enter a positive integer to check: ";
   cin >> num;

   // Argument num is passed to the function prime()
   prime(num);
   return 0;
}

// There is no return value to calling function. Hence, return type of function is void. */
void prime(int n)
{
   int i, flag = 0;
   for (i = 2; i <= n/2; ++i)
   {
```

```
        if (n%i == 0)
        {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
    {
        cout << n << " is not a prime number.";
    }
    else {
        cout << n << " is a prime number.";
    }
}
```

In the above program, positive number is first asked from the user which is stored in the variable `num`.

Then, `num` is passed to the `prime()` function where, whether the number is prime or not is checked and printed.

Since, the return type of `prime()` is a `void`, no value is returned from the function.

Example 4: Arguments passed and a return value.

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

int prime(int n);

int main()
{
    int num, flag = 0;
    cout << "Enter positive integer to check: ";
    cin >> num;

    // Argument num is passed to check() function
    flag = prime(num);

    if(flag == 1)
        cout << num << " is not a prime number.";
    else
        cout<< num << " is a prime number.";
    return 0;
}
```

```
/* This function returns integer value.  */
int prime(int n)
{
   int i;
   for(i = 2; i <= n/2; ++i)
   {
      if(n % i == 0)
         return 1;
   }

   return 0;
}
```

In the above program, a positive integer is asked from the user and stored in the variable `num`.

Then, `num` is passed to the function `prime()` where, whether the number is prime or not is checked.

Since, the return type of `prime()` is an `int`, 1 or 0 is returned to the `main()` calling function. If the number is a prime number, 1 is returned. If not, 0 is returned.

Back in the `main()` function, the returned 1 or 0 is stored in the variable `flag`, and the corresponding text is printed onto the screen.

C++ Function Overloading

Two or more functions having same name but different argument(s) are known as overloaded functions. In this article, you will learn about function overloading with examples.

Function refers to a segment that groups code to perform a specific task.

In C++ programming, two functions can have same name if number and/or type of arguments passed are different.

These functions having different number or type (or both) of parameters are known as overloaded functions. For example:

```
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.

Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

```
// Error code
int test(int a) { }
double test(int b){ }
```

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

Example 1: Function Overloading

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

void display(int);
void display(float);
void display(int, float);

int main() {

   int a = 5;
   float b = 5.5;

   display(a);
   display(b);
   display(a, b);

   return 0;
}

void display(int var) {
   cout << "Integer number: " << var << endl;
}
```

```
void display(float var) {
   cout << "Float number: " << var << endl;
}

void display(int var1, float var2) {
   cout << "Integer number: " << var1;
   cout << " and float number:" << var2;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Here, the `display()` function is called three times with different type or number of arguments.

The return type of all these functions are same but it's not necessary.

Example 2: Function Overloading

```
// Program to compute absolute value
// Works both for integer and float

#include <iostream>
using namespace std;

int absolute(int);
float absolute(float);

int main() {
   int a = -5;
   float b = 5.5;

   cout << "Absolute value of " << a << " = " << absolute(a) << endl;
   cout << "Absolute value of " << b << " = " << absolute(b);
   return 0;
}

int absolute(int var) {
    if (var < 0)
        var = -var;
   return var;
```

```
}

float absolute(float var){
   if (var < 0.0)
      var = -var;
   return var;
}
```

Output

```
Absolute value of -5 = 5
Absolute value of 5.5 = 5.5
```

In the above example, two functions `absolute()` are overloaded.

Both functions take single argument. However, one function takes integer as an argument and other takes float as an argument.

When `absolute()` function is called with integer as an argument, this function is called:

```
int absolute(int var) {
    if (var < 0)
        var = -var;
    return var;
}
```

When `absolute()` function is called with float as an argument, this function is called:

```
float absolute(float var){
   if (var < 0.0)
      var = -var;
   return var;
}
```

Inline Function

One disadvantage of using normal function is that it will decrease the execution speed of the program. Because every time a function is called the control of the program is transferred to the function body, after executing all the statements inside function the control of the program is transferred immediately after the statement from which the function is called. Thus it will take lots of extra time when function is small and it needs to be called repeatedly. This problem can solved using the concept of inline function. In order to make a normal function inline you just have to precede the function definition with inline keyword. The inline function must be defined before it is called in the program. It means at the starting of the program.

When the function is defined as inline the function call is replaced by function definition at the time of function call. So there it will increase the execution speed of the program.

The general form of inline function is given below:

```
inline Return-type Function_ Name (Argument_List)
{
Block of statement
}
```

Advantage of inline function is that it will increase the speed of program execution. You should declare the function as inline only when the function contains two or three statements. **Disadvantage** of inline function is that each time a function is called it will replace by function definition so extra memory space is occupied at the time of executing the program.

Inline functions provide following advantages:

1) Function call overhead doesn't occur.

2) It also saves the overhead of push/pop variables on the stack when function is called.

3) It also saves overhead of a return call from a function.

4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.

5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Inline function disadvantages:

1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

The following program demonstrates the use of use of inline function.

```
#include <iostream>

using namespace std;

inline int cube(int s)

{

        return s*s*s;

}

int main()

{

        cout << "The cube of 3 is: " << cube(3) << "\n";

        return 0;

} //Output: The cube of 3 is: 27
```

Inline function and classes:
It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.
For example:

```
class S

{

public:
```

```
        inline int square(int s) // redundant use of inline
        {
                // this function is automatically inline
                // function body
        }
};
#include <iostream>
using namespace std;
class operation
{
        int a,b,add,sub,mul;
        float div;
public:
        void get();
        void sum();
        void difference();
        void product();
        void division();
};
inline void operation :: get()
{
        cout << "Enter first value:";
        cin >> a;
        cout << "Enter second value:";
        cin >> b;
```

```
}

inline void operation :: sum()
{
        add = a+b;
        cout << "Addition of two numbers: " << a+b << "\n";
}

inline void operation :: difference()
{
        sub = a-b;
        cout << "Difference of two numbers: " << a-b << "\n";
}

inline void operation :: product()
{
        mul = a*b;
        cout << "Product of two numbers: " << a*b << "\n";
}

inline void operation ::division()
{
        div=a/b;
        cout<<"Division of two numbers: "<<a/b<<"\n" ;
}
```

```
int main()
{
        cout << "Program using inline function\n";
        operation s;
        s.get();
        s.sum();
        s.difference();
        s.product();
        s.division();
        return 0;
}
```

Output:

```
Enter first value: 45
Enter second value: 15
Addition of two numbers: 60
Difference of two numbers: 30
Product of two numbers: 675
Division of two numbers: 3
```

Library Functions In C++

Library functions which are also called as "built-in" functions are the functions that are already available and implemented in C++.

We can directly call these functions in our program as per our requirements. Library functions in C++ are declared and defined in special files called "Header Files" which we can reference in our C++ programs using the "include" directive.

Headers	Description
iostream	This header contains the prototype for standard input and output functions used in C++ like cin, cout, etc.

Headers	Description
cmath	This is the header containing various math library functions.
iomanip	This header contains stream manipulator functions that allow us to format the stream of data.
cstdlib	The header cstdlib contains various functions related to conversion between text and numbers, memory allocation, random numbers, and other utility functions.
ctime	ctime contains function prototypes related to date and time manipulations in C++.
cctype	This header includes function prototypes that test the type of characters (digit, punctuation, etc.). It also has prototypes that are used to convert from uppercase to lowercase and another way around.
cstring	cstring header includes function prototypes for C-style string-processing functions.
cstdio	This header contains function prototypes for the C-style standard input/output library functions which we included initially in stdio.h
fstream	Function prototypes for functions that perform input/output from/to files on disk are included in fstream header.
climits	climits header has the integral size limits of the system.
cassert	cassert header contains macros and variables for adding diagnostics that help us in program debugging.
cfloat	This header file contains the size limits for floating-point numbers on the system.
string	The header string defines the class string of the C++ Standard Library.
list, vector, stack, queue, deque, map, set, bitset	All these headers are used for Standard Template Library (STL) implementation. Each of these headers contains the respective class definition and function prototypes.
typeinfo	This header contains various classes for Runtime Type Identification (RTTI).
exception, stdexcept	All the classes and functions used for exception handling in C++ are included in these two headers.
memory	This header is used by the C++ standard library to allocate memory.
sstream	Functions that read input from strings in memory and output to strings in memory require functions prototypes from sstream header to implement

Headers	Description
	the functionality.
functional	Used by C++ standard library algorithms.
iterator	Function prototypes and classes in this header are used by Standard Template Library to traverse through or iterate through the data inside containers.
algorithm	Methods that act on STL container data are included in this header algorithm
locale	To process data in the original natural form for different languages or locales (currencies, character presentation, etc.), the locale header definitions are used.
limits	This header defines the data type limit for Numbers on each platform.
utility	This header contains utility functions and classes used by the Standard C++ library.

We have already used most of these headers throughout our tutorial so far. Notable is <iostream>, <string>, <ctime> headers that we have used from time to time.

Function	Description
sqrt(x)	Accepts any non-negative numeric parameter x and returns the square root of this number x
pow(base,exponent)	Raises the 'base' value to the power specified by the exponent. Returns base^exponent.
exp(x)	Takes any number (positive, negative or zero) as a parameter and returns exponential (Euler's number) e raised to the given parameter
fabs(x)	Returns absolute value of an argument.
log(x)	Returns the natural logarithm (to the base e) of value x
log 10(x)	Return the logarithm (to the base 10) of value x
sin(x)	Returns sine of the angle x (in radians)
cos(x)	Returns cosine of angle x (in radians)
tan(x)	Returns tangent of angle x (in radians)
asin(x)	Returns inverse sine (in radians) of number x
acos(x)	Returns inverse cosine (in radians) of number x

Function	Description
atan(x)	Returns inverse tangent (in radians) of number x
Function	Description
toupper(ch)	Takes in character 'ch' as an argument and returns the uppercase equivalent of ch if it's present otherwise returns ch.
tolower(ch)	Takes in character 'ch' as an argument and returns the lowercase equivalent of ch if it's present otherwise returns ch.
isalpha(ch)	Returns non-zero if ch is alphabet otherwise 0.
isalnum(ch)	Returns non-zero if ch is alphanumeric (alphabet or number) otherwise 0.
isupper(ch)	Returns non-zero value if ch is uppercase otherwise 0.
isdigit(ch)	Returns non-zero value if ch is a number otherwise 0.
islower()	Returns non-zero value if ch is lowercase otherwise 0.
Function	Description
abs(x)	Returns absolute value of an integral argument x
atof(const char* str)	Converts string to double; returns double
atoi(const char* str)	Converts string to int; returns an int
atol(const char* str)	Converts string to long int; returns a long int
atoll(const char* str)	Converts string to long long int; returns a long long int
strtod	Converts string to double
strtol	Converts string to long int
strtoul	Converts string to unsigned long integer
strtof	Converts string to float
strtold	Converts string to long double
strtoull	Converts string to unsigned long long integer
strtoll	Converts string to long long integer
srand(int seed)	This is a pseudo-random generator that is initialized to argument 'seed'

Function	Description
qsort	Sorts elements of the array in ascending order(internally uses quick sort method)
abort	Aborts the process resulting in abnormal program termination
atexit	Has function passed as an argument which is executed resulting in normal program termination.
malloc(size_t size)	Used to allocate memory specified by size and return a pointer to it
calloc (size_t num, size_t size)	Allocates memory of (num*size) bytes with all bits initialized to zero
free(void* ptr)	Deallocates memory block allocated by malloc, calloc or realloc function call.
realloc (void* ptr, size_t size)	Resizes the memory block pointed to by ptr that was initially allocated using malloc or calloc function call.
quick_exit	Normal termination of the process after returning control to the host environment.
system	Invokes command processor to execute system command passed as an argument
getenv	Retrieves the value of environment string passed as an argument to the function
wctomb	Convert the wide character to a multibyte sequence
wcstombs	Convert wide character string to multibyte string

C++ Classes and Objects

C++ is a multi-paradigm programming language. Meaning, it supports different programming styles.

One of the popular ways to solve a programming problem is by creating objects, known as object-oriented style of programming.

C++ supports object-oriented (OO) style of programming which allows you to divide complex problems into smaller sets by creating objects.

Object is simply a collection of data and functions that act on those data.

C++ Class

Before you create an object in C++, you need to define a class.

A class is a blueprint for the object.

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

As, many houses can be made from the same description, we can create many objects from a class.

How to define a class in C++?

A class is defined in C++ using keyword `class` followed by the name of class.
The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

```
class className
  {
  // some data
  // some functions
  };
```

Example: Class in C++

```
class Test
{
   private:
      int data1;
      float data2;
```

```
public:
   void function1()
   {  data1 = 2;  }

   float function2()
   {
      data2 = 3.5;
      return data2;
   }
};
```

Here, we defined a class named `Test`
This class has two data members: `data1` and `data2` and two member functions: `function1()` and `function2()`.

Keywords: private and public

You may have noticed two keywords: private and public in the above example.

The private keyword makes data and functions private. Private data and functions can be accessed only from inside the same class.

The public keyword makes data and functions public. Public data and functions can be accessed out of the class.

Here, `data1` and `data2` are private members where as `function1()` and `function2() are public members`.
If you try to access private data from outside of the class, compiler throws error. This feature in OOP is known as data hiding.

C++ Objects

When class is defined, only the specification for the object is defined; no memory or storage is allocated.

To use the data and access functions defined in the class, you need to create objects.

Syntax to Define Object in C++

```
className objectVariableName;
```

You can create objects of `Test` class (defined in above example) as follows:

```
class Test
{
  private:
    int data1;
    float data2;

  public:
    void function1()
    {  data1 = 2;  }

    float function2()
    {
        data2 = 3.5;
        return data2;
    }
};

int main()
{
  Test o1, o2;
}
```

Here, two objects `o1` and `o2` of `Test` class are created.

In the above class `Test`, `data1` and `data2` are data members and `function1()` and `function2()` are member functions.

How to access data member and member function in C++?

You can access the data members and member functions by using a . (dot) operator.

For example,

```
o2.function1();
```

This will call the function1() function inside the Test class for objects o2.

Similarly, the data member can be accessed as:

```
o1.data2 = 5.5;
```

It is important to note that, the private members can be accessed only from inside the class.

So, you can use `o2.function1();` from any function or class in the above example.

However, the code `o1.data2 = 5.5;` should always be inside the class `Test`.

Example: Object and Class in C++ Programming

```
// Program to illustrate the working of objects and class in C++ Programming
#include <iostream>
using namespace std;

class Test
{
   private:
      int data1;
      float data2;

   public:

      void insertIntegerData(int d)
      {
         data1 = d;
         cout << "Number: " << data1;
      }

      float insertFloatData()
      {
         cout << "\nEnter data: ";
         cin >> data2;
         return data2;
      }
};

int main()
{
    Test o1, o2;
    float secondDataOfObject2;
```

```
    o1.insertIntegerData(12);
    secondDataOfObject2 = o2.insertFloatData();

    cout << "You entered " << secondDataOfObject2;
    return 0;
}
```

Output

```
Number: 12
Enter data: 23.3
You entered 23.3
```

In this program, two data members `data1` and `data2` and two member functions `insertIntegerData()` and `insertFloatData()` are defined under `Test` class.

Two objects `o1` and `o2` of the same class are declared.

The `insertIntegerData()` function is called for the `o1` object using:

```
o1.insertIntegerData(12);
```

This sets the value of data1 for object `o1` to 12.

Then, the `insertFloatData()` function for object `o2` is called and the return value from the function is stored in variable `secondDataOfObject2` using:

```
secondDataOfObject2 = o2.insertFloatData();
```

In this program, `data2` of `o1` and `data1` of `o2` are not used and contains garbage value.

Friend Function

A friend function is a function that is not a member of a class but it can access private and protected member of the class in which it is declared as friend.

Since friend function is not a member of class it can not be accessed using object of the class. It is called in the same way as normal external function is called.

It works same as your real life friend. Your friend is not a member of your family but still he knows about you and your family.

Sometimes it is required that private member of the class can be accessed outside the class at that time we

have to use friend function.

A function can be declared as a friend by preceding function declaration with friend keyword as shown below:

friend Return_Type Function_Name (Argument List);

Example:

```
#include<iostream.h>
class Circle
{
int r;
public:
void input()
{
cout<<"Enter Radius:";
cin>>r;
}
friend float area(Circle C);
};
float area(Circle C)
{
return (3.14*C.r*C.r);
}
int main()
{
Circle C1;
C1.input();
cout<<"Area of Circle is:"<<area(C1);
return 0;
}
```

Friend function having following characteristics:

(1) A friend function can be declared inside class but it is not member of the class.

(2) It can be declared either public or private without affecting its meaning.

(3) A friend function is not a member of class so it is not called using object of the class. It is called like normal external function.

(4) A friend function accepts object as an argument to access private or public member of the class.

(5) A friend function can be declared as friend in any number of classes.

Access Modifiers in C++

Access modifiers are used to implement an important feature of Object-Oriented Programming known as **Data Hiding**. Consider a real-life example:
The Indian secret informatic system having 10 senior members have some top secret regarding national security. So we can think that 10 people as class data members or member functions who can directly access secret information from each other but anyone can't access this information other than these 10 members i.e. outside people can't access information directly without having any privileges. This is what data hiding is.
Access Modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.
There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Private**.
Let us now look at each one these access modifiers in details:

1. **Public**: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.
   Example:

```
// C++ program to demonstrate public
// access modifier

#include<iostream>
using namespace std;

// class definition
class Circle
```

```
{
    public:
        double radius;

        double compute_area()
        {
            return 3.14*radius*radius;
        }

};

// main function
int main()
{
    Circle obj;

    // accessing public datamember outside class
    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```

Output:

```
Radius is: 5.5
Area is: 94.985
```

In the above program the data member *radius* is public so we are allowed to access it outside the class.

2.Private: The class members declared as *private* can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to

access the private data members of a class.

Example:

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
	// private data member
	private:
		double radius;

	// public member function
	public:
		double compute_area()
		{ // member function can access private
			// data member radius
			return 3.14*radius*radius;
		}

};

// main function
int main()
{
	// creating object of the class
	Circle obj;

	// trying to access private data member
	// directly outside the class
	obj.radius = 1.5;

	cout << "Area is:" << obj.compute_area();
	return 0;
}
```

The output of above program will be a compile time error because we are not allowed to access the private data members of a class directly outside the class.

Output:

```
 In function 'int main()':

11:16: error: 'double Circle::radius' is private

        double radius;

               ^

31:9: error: within this context

    obj.radius = 1.5;
```

```
    ^
```

However, we can access the private data members of a class indirectly using the public member functions of the class. Below program explains how to do this:

```
// C++ program to demonstrate private
// access modifier

#include<iostream>
using namespace std;

class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        void compute_area(double r)
        { // member function can access private
            // data member radius
            radius = r;

            double area = 3.14*radius*radius;

            cout << "Radius is: " << radius << endl;
            cout << "Area is: " << area;
        }

};

// main function
int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class
    obj.compute_area(1.5);


    return 0;
}
```

Output:

```
Radius is: 1.5

Area is: 7.065
```

3.Protected: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

Example:

```
// C++ program to demonstrate
// protected access modifier
#include <bits/stdc++.h>
using namespace std;

// base class
class Parent
{
	// protected data members
	protected:
	int id_protected;

};

// sub class or derived class
class Child : public Parent
{


	public:
	void setId(int id)
	{

		// Child class is able to access the inherited
		// protected data members of base class

		id_protected = id;

	}

	void displayId()
	{
		cout << "id_protected is: " << id_protected << endl;
	}
};

// main function
int main() {

	Child obj1;

	// member function of the derived class can
	// access the protected data members of the base class

	obj1.setId(81);
	obj1.displayId();
```

```
        return 0;
    }
```

Output:

```
id_protected is: 81
```