**Section 1)** **History of C++**

- C++ was designed by Bjarne Stroustrup while he was working for AT&T Bell Labs.

- Initial versions of the language were made available internally at AT&T beginning in 1981

- The C++ Programming Language, was published in early 1986.

**==================================================================**

**Section 2**) **Why C++ over C**

**Difference between C and C++ :**

The major difference between C and C++ is that C is a procedural programming language and does not support classes and objects, while C++ is a combination of both procedural and object oriented programming language, therefore C++ can be called a hybrid language.

|  |  |
| --- | --- |
| C was developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs. | C++ was developed by Bjarne Stroustrup in 1979 with C++'s predecessor "C with Classes". |
| C supports procedural programming paradigm for code development. | C++ supports both procedural and object oriented programming paradigms; therefore C++ is also called a hybrid language. |
| C does not support object oriented programming; therefore it has no support for polymorphism, encapsulation, and inheritance. | Being an object oriented programming language C++ supports polymorphism, encapsulation, and inheritance. |
| In C (because it is a procedural programming language), data and functions are separate and free entities. | In C++ (when it is used as object oriented programming language), data and functions are encapsulated together in form of an object. For creating objects class provides a blueprint of structure of the object. |
| In C, data are free entities and can be manipulated by outside code. This is because C does not support information hiding. | In C++, Encapsulation hides the data to ensure that data structures and operators are used as intended. |
| C, being a procedural programming, it is a function driven language. | While, C++, being an object oriented programming, it is an object driven language. |
| C does not support function and operator overloading. | C++ supports both function and operator overloading. |
| C does not allow functions to be defined inside structures. | In C++, functions can be used inside a structure. |
| C does not have namespace feature. | C++ uses NAMESPACE which avoid name collisions.  A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries. All identifiers at namespace scope are visible to one another without qualification. Identifiers outside the namespace can access the members by using the fully qualified name for each identifier. |
| C uses functions for input/output. For example scanf and printf. | C++ uses objects for input output. For example cin and cout. |
| C does not support reference variables. | C++ supports reference variables. |
| C has no support for virtual and friend functions. | C++ supports virtual and friend functions. |
| C provides malloc() and calloc() functions for dynamic memory allocation, and free() for memory de-allocation. | C++ provides new operator for memory allocation and delete operator for memory de-allocation. |
| C does not provide direct support for error handling (also called exception handling) | C++ provides support for exception handling. Exceptions are used for "hard" errors that make the code incorrect. |

Link : <http://cs-fundamentals.com/tech-interview/c/difference-between-c-and-cpp.php>

**===================================================================**

**Section 3**) **OOPS concepts**

**Object oriented programming :**

As the name suggests uses objects in programming. Object oriented programming aims to implement real world entities like inheritance, hiding, polymorphism etc in programming. The main aim of OOP is to bind together the data and the functions that operates on them so that no other part of code can access this data except that function.

**Objects :**

Objects are instances of a class, these are defined user defined data types.

**Class :**

Class is a blueprint of data and functions or methods.

Class is a user defined data type like structures and unions in C.

**OOPS concept.**

**Encapsulation:**

Wrapping up of data and functions into a single unit is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapping in the class can access it.

**Data abstraction** :

- Poviding only needed information to the outside world and hiding implementation detail.

Example is tv, which we can turn on, turn off,change the channel, adjust the volume and add externel component such as speakers, but we dont knoe the internal details how it recived signal over air and translates them.

- Abstraction using access specifiers.

**Refer Additional link :** [**http://www.geeksforgeeks.org/abstraction-in-c/**](http://www.geeksforgeeks.org/abstraction-in-c/)

**Inheritance :**

Inheritance provides re usability. This means that we can add additional features to an existing class without modifying it.

**Polymorphism:**

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

C++ supports operator overloading and function overloading.

**Refer** **Additional link** : <http://www.geeksforgeeks.org/polymorphism-in-c/>

**Dynamic Binding:**

In dynamic binding, the code to be executed in response to function call is decided at runtime.

Example is virtual function.

**Message Passing** :

An object-oriented program consists of a set of objects that communicate with each other.

The process of programming in an object-oriented language therefore involves the following basic steps:  
     1. Creating classes that define objects and their behavior.  
     2. Creating objects from class definitions.  
     3. Establishing communication among objects.

**Refer** **Additional link** : <https://www.howtoforge.com/learning-c-cplusplus-step-by-step-p13>

**===================================================================**

**Section 4**) **Setting up C++ Development Environment**

**Text Editor :**

- Vim editor : open the file in vim editor (on linux).

Or using eclispe and save file with **.cpp** extension.

**C++ Compiler :**

A compiler is a computer program which converts high-level language into machine understandable low-level language

- **linux installation** : Installed the command using below mentioned command.

# sudo apt-get update

# sudo apt-get install GCC ---- Install the gcc compiler.

# sudo apt-get install build-essential ---- This command will install all the libraries which are

required to compile and run a C++ program

- **Version check** : Check the compiler using below mentioned command.

# g++ --version ---- Compiler version check.

- **File compilation :** Compile the .cpp file using below mentioned command.

# g++ *filename.cpp* -o *any-name --- Compile the c++ file.*

*Filename.cpp* is the name of your source code file. In our case, the name is “helloworld.cpp” and *any-name* can be any name of your choice. This name will be assigned to the executable file which is created by compiler after compilation.

#g++ helloworld.cpp -o hello

- **Run the executable :**

**#**./hello --- Run the executable.

**==================================================================**

**Section 5)**  **Writing first C++ program**

**Simple program :**

// Simple C++ program to display "Hello World"

// Header file for input output functions

#include<iostream>

using namespace std;

// main function -

// where the execution of program begins

int main()

{

    // prints hello world

    cout<<"Hello World";

    return 0;

}

**Commented line:** As in C, C++ comments start with “//” sign

**/ / Simple C++ program to display "Hello World"**

**Preprocessor directive :**

**- #include**:In C++,  all lines that start with pound (#) sign are called directives and are processed by preprocessor which is a program invoked by the compiler

**Standered library include files** :

**#include<iostream>** . It tells the compiler to include the standard iostream file which contains declarations of all the standard input/output library functions.

**Other part of program :**

- **int main()**: This line is used to declare a function named “main” which returns data of integer type.

- **{ and }**:The opening braces ‘{‘ indicates the beginning of the main function and the closing braces ‘}’ indicates the ending of the main function.

- **std::cout<<“Hello World”; --------------------- like printf**

**===================================================================**

**Section 6 ) Basic Input / Output in C++**

**Input Stream:**

If the direction of flow of bytes is from device(for example: Keyboard) to the main memory then this process is called input.

**Standard input stream (cin)**:

Usually the input device is the keyboard. **cin** is the instance of the class **istream** and is used to read input from the standard input device which is usually keyboard.

The extraction operator(**>>**) is used along with the object **cin** for reading inputs. The extraction operator extracts the data from the object **cin** which is entered using the keboard.

Sample program

|  |
| --- |
| #include<iostream>  using namespace std;    int main()  {      int age;        cout << "Enter your age:";      cin >> age;      cout << "\nYour age is: "<<age;        return 0;  } |

**Output Stream:**

If the direction of flow of bytes is opposite, i.e. from main memory to device( display screen ) then this process is called output.

**Standard output stream (cout):**

Usually the **standard output device** is the display screen. **cout** is the instance of the **ostream class**. **cout** is used to produce output on the **standard output device** which is usually the **display screen**. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the **insertion operator (<<)**.

// 'Hello World!' program

#include <iostream>

int main()

{

std::cout << "Hello World!" << std::endl;

return 0;

}

**Un-buffered standard error stream (cerr):**

**cerr** is the **standard error stream** which is used to output the errors. This is also an instance of the **ostream class**. As cerr is un-buffered so it is used when we need to display the error message immediately. It does not have any buffer to store the error message and display later.

#include <iostream>

using namespace std;

int main( )

{

   cerr << "An error occured";

   return 0;

}

|  |
| --- |
|  |

**==================================================================**

**Section 7 )** C++ Data Types

**Primitive Data Types:**

These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char , float, bool etc. Primitive data types available in C++ are:

* Integer - int
* Character - char
* Boolean - bool
* Floating Point - float
* Double Floating Point - double
* Valueless or Void - used for function
* Wide Character

**Abstract or user defined data type :**

Apart form the c , follwing are the addistional user defined data types in c++

Class, structure, union

**References in C++**

When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting ‘&’ in the declaration.

#include<iostream>

using namespace std;

int main()

{

  int x = 10;

  // ref is a reference to x.

  int& ref = x;

  // Value of x is now changed to 20

  ref = 20;

  cout << "x = " << x << endl ;

  // Value of x is now changed to 30

  x = 30;

  cout << "ref = " << ref << endl ;

  return 0;

}

Refer additional link : <http://www.geeksforgeeks.org/references-in-c/>

**==================================================================**

**Section 8** ) **C vs C++**

**C program that won't work in c+ :**

1) In C++, it is compiler error to make a normal pointer to point a const variable, but it is allowed in C.

|  |
| --- |
| #include <stdio.h>    int main(void)  {      int const j = 20;        /\* The below assignment is invalid in C++, results in error         In C, the compiler \*may\* throw a warning, but casting is         implicitly allowed \*/      int \*ptr = &j;  // A normal pointer points to const        printf("\*ptr: %d\n", \*ptr);        return 0;  } |

2) In C, a void pointer can directly be assigned to some other pointer like int \*, char \*. But in C++, a void pointer must be explicitly typcasted.

#include <stdio.h>

int main()

{

   void \*vptr;

   int \*iptr = vptr; //In C++, it must be replaced with int \*iptr=(int \*)vptr;

   return 0;

}

3) Following program compiles & runs fine in C, but fails in compilation in C++. const variable in C++ must be initialized but in c it isn’t necessary. Thanks to Pravasi Meet for suggesting this point.

#include <stdio.h>

int main()

{

    const int a;   // LINE 4

    return 0;

}

4) C++ does more strict type checking than C. For example the following program compiles in C, but not in C++. In C++, we get compiler error “invalid conversion from ‘int’ to ‘char\*'”.

#include <stdio.h>

int main()

{

    char \*c = 333;

    printf("c = %u", c);

    return 0;

}

**Name mangling in c++ & extern “C” in C++:**

In C, names may not be mangled as C doesn’t support function overloading. So how to make sure that name of a symbol is not changed when we link a C code in C++. For example, see the following C++ program that uses printf() function of C.

// Save file as .cpp and use C++ compiler to compile it

int printf(const char \*format,...);

int main()

{

    printf("Hello world!");

    return 0;

}

The reason for compiler error is simple, name of *printf* is changed by C++ compiler and it doesn’t find definition of the function with new name.

The solution of problem is extern “C” in C++. When some code is put in extern “C” block, the C++ compiler ensures that the function names are unmangled – that the compiler emits a binary file with their names unchanged, as a C compiler would do.

If we change the above program to following, the program works fine and prints “Hello world!” on console.

// Save file as .cpp and use C++ compiler to compile it

extern "C"

{

    int printf(const char \*format,...);

}

int main()

{

    printf("Hello world!");

    return 0;

}

# **How does “void \*” differ in C and C++?**

C allows a void\* pointer to be assigned to any pointer type without a cast, whereas C++ does not; this idiom appears often in C code using malloc memory allocation. For example, the following is valid in C but not C++:

void\* ptr;

int \*i = ptr; /\* Implicit conversion from void\* to int\* \*/

**Difference between C structures and C++ structures.**

**Member functions inside structure**: Structures in C cannot have member functions inside structure but Structures in C++ can have member functions along with data members.

**Direct Initialization:** We cannot directly initialize structure data members in C but we can do it in C++

**Using struct keyword:** In C, we need to use struct to declare a struct variable. In C++, struct is not necessary.

**Static Members:** C structures cannot have static members but is allowed in C++.

**sizeof operator:** This operator will generate **0** for an empty structure in C whereas **1** for an empty structure in C++.

**Data Hiding:** C structures does not allow concept of Data hiding but is permitted in C++ as C++ is an object oriented language whereas C is not.

**Access Modifiers:** C structures does not have access modifiers as these modifiers are not suppoted by the language. C++ structures can have this concept as it is inbuilt in the language.

=======================================================================

**Section 9** ) **Input and output**

**endl vs \n in C++**

Difference between endl and '\n'

cout << endl**; -------------------------** Inserts a new line and flushes the stream

cout << "\n" ; ------------------------- Only inserts a new line.

cout << ‘\n’ << flush;

=======================================================================

**Section 10) Operators**

# **Scope Resolution Operator Versus this pointer in C++**

Consider below C++ program:

// C++ program to show that local parameters hide

// class members

#include<iostream>

using namespace std;

class Test

{

    int a;

public:

    Test() { a = 1; }

    // Local parameter 'a' hides class member 'a'

    void func(int a)  { cout << a; }

};

int main()

{

    Test obj;

    int k = 3 ;

    obj.func(k);

    return 0;

}

The output for the above program is 3 since the “a” passed as argument to the “func” shadows the “a” of the class .i.e 1  
Then how to output the class’s ‘a’. This is where **this pointer** comes in handy. A statement like “cout << this->a” instead of “cout << a" can simply output the value 1 as this pointer points to the object from whom func is called.

// C++ program to show use of this to access member when// there is a local variable with same name.

#include<iostream>

using namespace std;

class Test

{

    int a;

public:

    Test() { a = 1; }

    // Local parameter ‘a’ hides object’s member

    // ‘a’, but we can access it using this.

    void func(int a)  { cout << this->a; }

};

int main()

{

    Test obj;

    int k = 3 ;

    obj.func(k);

    return 0;

}

**new and delete operators in C++ for dynamic memory**

The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

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

// Pointer initialized with NULL

// Then request memory for the variable

int \*p = NULL;

p = new int;

**Initialize memory:**

int \*p = new int(25);

float \*q = new float(75.25);

**Allocate block of memory:**

int \*p = new int[10]

**delete operator:**

// Release memory pointed by pointer-variable

**delete** pointer-variable;

delete p;

delete q;

//free the entire array pointed by p.

delete[] p;

<http://www.geeksforgeeks.org/new-and-delete-operators-in-cpp-for-dynamic-memory/>

# **Casting operators in C++**

C++ supports following 4 types of casting operators:

1. const\_cast  
2. static\_cast  
3. dynamic\_cast  
4. reinterpret\_cast

**Refer Additional link** : <http://www.cplusplus.com/doc/tutorial/operators/>

============================================================================

**Section 11) Functions**

# **Default Arguments in C++**

A default argument is a value provided in function declaration that is automatically assigned by the compiler if caller of the function doesn’t provide a value for the argument with default value.

**Note** : Default value to the function parameter only at the point of function declaration not at defination in case if function is defined and declared seprartely.

**#include<iostream>**

using namespace std;

// A function with default arguments, it can be called with

// 2 arguments or 3 arguments or 4 arguments.

int sum(int x, int y, int z=0, int w=0)

{

    return (x + y + z + w);

}

/\* Drier program to test above function\*/

int main()

{

    cout << sum(10, 15) << endl;

    cout << sum(10, 15, 25) << endl;

    cout << sum(10, 15, 25, 30) << endl;

    return 0;

}

- Once default value is used for an argument, all subsequent arguments must have default value.

# **Inline Functions in C++**

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

**Prototype**

inline return-type function-name(parameters)

{

// function code

}

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

1) If a function contains a loop. (for, while, do-while)  
2) If a function contains static variables.  
3) If a function is recursive.  
4) If a function return type is other than void, and the return statement doesn’t exist in function body.  
5) If a function contains switch or goto statement.

**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 inside the class**

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.

class S

{

public:

    inline int square(int s) // redundant use of inline

    {

        // this function is automatically inline

        // function body

    }

};

**For example:**

|  |
| --- |
|  |

# **Return from void functions in C++**

Void functions are “void” due to the fact that they are not supposed to return values. True, but not completely. We cannot return values but there is something we can surely return from void functions. Some of cases are listed below.

- We can simply write return statement in a void fun().

- In-fact it is considered a good practice (for readability of code) to write return; statement to indicate end of function.

#include <iostream>

using namespace std;

void fun()

{

   cout << "Hello";

   // We can write return in void

   return;

}

int main()

{

   fun();

   return 0;

}

**Refer Additional Link** : <http://www.geeksforgeeks.org/return-void-functions-c/>

======================================================================

**Section 12)** **Pointers and references**

# **References in C++**

- When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting ‘&’ in the declaration.

- When a variable is declared as reference, it becomes an alternative name for an existing variable.

- A variable can be declared as reference by putting ‘&’ in the declaration.

#include<iostream>

using namespace std;

int main()

{

  int x = 10;

  // ref is a reference to x.

  int& ref = x;

  // Value of x is now changed to 20

  ref = 20;

  cout << "x = " << x << endl ;

  // Value of x is now changed to 30

  x = 30;

  cout << "ref = " << ref << endl ;

  return 0;

}

**References vs Pointers**

- A pointer can be declared as void but a reference can never be void

int a = 10;

void\* aa = &a;. //it is valid

void &ar = a; // it is not valid

- Once a reference is created, it cannot be later made to reference another object; it cannot be reseated. This is often done with pointers.

- References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.

- A reference must be initialized when declared. There is no such restriction with pointers

# **When do we pass arguments by reference or pointer**

*To modify local variables of the caller function:*

A reference (or pointer) allows called function to modify a local variable of the caller function. For example, consider the following example program where *fun()* is able to modify local variable *x* of *main()*.

void fun(int &x) {

    x = 20;

}

int main() {

    int x = 10;

    fun(x);

    cout<<"New value of x is "<<x;

    return 0;

}

*For passing large sized arguments:*

If an argument is large, passing by reference (or pointer) is more efficient because only an address is really passed, not the entire object.

*To avoid Object Slicing:*If we pass an object of subclass to a function that expects an object of superclass then the passed object is sliced if it is pass by value. For example, consider the following program, it prints “This is Pet Class”.

#include <iostream>

#include<string>

using namespace std;

class Pet {

public:

    virtual string getDescription() const {

        return "This is Pet class";

    }

};

class Dog : public Pet {

public:

    virtual string getDescription() const {

        return "This is Dog class";

    }

};

void describe(Pet p) { // Slices the derived class object

    cout<<p.getDescription()<<endl;

}

int main() {

    Dog d;

    describe(d);

    return 0;

}

if we use pass by reference in the above program then it correctly prints “This is Dog Class”. See the following modified program.

|  |
| --- |
|  |

**4)** *To achieve Run Time Polymorphism in a function*  
We can make a function polymorphic by passing objects as reference (or pointer) to it.

# Refer Aditional link : <http://www.geeksforgeeks.org/passing-by-pointer-vs-passing-by-reference-in-c/>

# **‘this’ pointer in C++**

The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. ‘this’ pointer is a constant pointer that holds the memory address of the current object.

Following are the situations where ‘this’ pointer is used:

**1) When local variable’s name is same as member’s name**

#include<iostream>

using namespace std;

/\* local variable is same as a member's name \*/

class Test

{

private:

   int x;

public:

   void setX (int x)

   {

       // The 'this' pointer is used to retrieve the object's x

       // hidden by the local variable 'x'

       this->x = x;

   }

   void print() { cout << "x = " << x << endl; }

};

int main()

{

   Test obj;

   int x = 20;

   obj.setX(x);

   obj.print();

   return 0;

}

**2) To return reference to the calling object**

/\* Reference to the calling object can be returned \*/

Test& Test::func ()

{

   // Some processing

   return \*this;

}

#include<iostream>

using namespace std;

class Test

{

private:

  int x;

  int y;

public:

  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }

  Test &setX(int a) { x = a; return \*this; }

  Test &setY(int b) { y = b; return \*this; }

  void print() { cout << "x = " << x << " y = " << y << endl; }

};

int main()

{

  Test obj1(5, 5);

  // Chained function calls.  All calls modify the same object

  // as the same object is returned by reference

  obj1.setX(10).setY(20);

  obj1.print();

  return 0;

}

**Additional link**: <http://www.geeksforgeeks.org/this-pointer-in-c/>

# **Dangling Pointer**

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.

/ Deallocating a memory pointed by ptr causes

// dangling pointer

#include <stdlib.h>

#include <stdio.h>

int main()

{

    int \*ptr = (int \*)malloc(sizeof(int));

    // After below free call, ptr becomes a

    // dangling pointer

    free(ptr);

    // No more a dangling pointer

    ptr = NULL;

}

===============================================================

**Section 13) Dynamic memory allocation**

# **new and delete operators in C++**

**- new operator :** The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

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

**- delete operator**

**delete** pointer-variable;

delete p;

delete q;

# **malloc() vs new**

**Calling Constructors:** new calls constructors, while malloc() does not. In fact primitive data types (char, int, float.. etc) can also be initialized with new. For example, below program prints 10.

#include<iostream>

 using namespace std;

 int main()

{

   int \*n = new int(10); // initialization with new()

   cout << \*n;

   getchar();

   return 0;

}

**operator vs function:** new is an operator, while malloc() is a function

**return type:** new returns exact data type, while malloc() returns void \*.

**Failure Condition:** On failure, malloc() returns NULL where as new Throws.

**Memory:** In case of new, memory is allocated from free store where as in malloc() memory allocation is done from heap.

**Overriding:** We are allowed to override new operator where as we can not override the malloc() function legally.

**Size:** Required size of memory is calculated by compiler for new, where as we have to manually calculate size for malloc().

**Additional link** : <http://www.geeksforgeeks.org/malloc-vs-new/>

**delete and free() in C++**

In C++, delete operator should only be used either for the pointers pointing to the memory allocated using new operator or for a NULL pointer, and free() should only be used either for the pointers pointing to the memory allocated using malloc() or for a NULL pointer.

#include<stdio.h>

#include<stdlib.h>

int main()

{

    int x;

    int \*ptr1 = &x;

    int \*ptr2 = (int \*)malloc(sizeof(int));

    int \*ptr3 = new int;

    int \*ptr4 = NULL;

    /\* delete Should NOT be used like below because x is allocated

        on stack frame \*/

    delete ptr1;

    /\* delete Should NOT be used like below because x is allocated

        using malloc() \*/

    delete ptr2;

    /\* Correct uses of delete \*/

    delete ptr3;

    delete ptr4;

    getchar();

    return 0;

}

===============================================================

**Section 14) Classes and Objects**

**Class**

The building block of C++ that leads to Object Oriented programming is a **Class**. It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

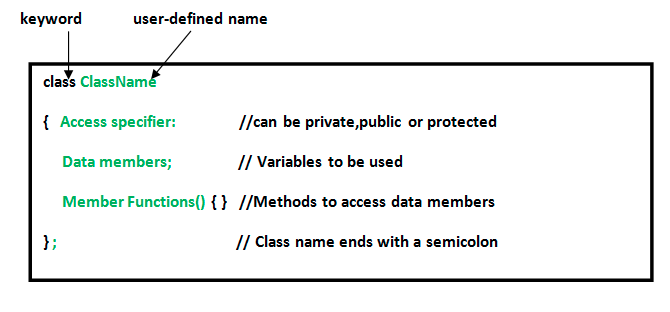
**-** A Class is a user defined data-type which have data members and member functions

**-** Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.

**-** In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

**Object**

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

**Declaring Objects:** When a 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.

**ClassName ObjectName;**

**Accessing data members and member functions:**

This access control is given by Access modifiers in C++. There are three access modifiers : **public, private and protected**.

// C++ program to demonstrate

// accessing of data members

 #include <bits/stdc++.h>

using namespace std;

class Geeks

{

    // Access specifier

    public:

    // Data Members

    string geekname;

    // Member Functions()

    void printname()

    {

       cout << "Geekname is: " << geekname;

    }

};

int main() {

    // Declare an object of class geeks

    Geeks obj1;

    // accessing data member

    obj1.geekname = "Abhi";

    // accessing member function

    obj1.printname();

    return 0;

}

**Member Functions in Classes**

There are 2 ways to define a member function:

* Inside class definition
* Outside class definition (Using scope resolution operator)

// C++ program to demonstrate function

// declaration outside class

#include <bits/stdc++.h>

using namespace std;

class Geeks

{

    public:

    string geekname;

    int id;

    // printname is not defined inside class defination

    void printname();

    // printid is defined inside class defination

    void printid()

    {

        cout << "Geek id is: " << id;

    }

};

// Definition of printname using scope resolution operator ::

void Geeks::printname()

{

    cout << "Geekname is: " << geekname;

}

int main() {

    Geeks obj1;

    obj1.geekname = "xyz";

    obj1.id=15;

    // call printname()

    obj1.printname();

    cout << endl;

    // call printid()

    obj1.printid();

    return 0;

}

**Special member function.**

**Constructor** : Constructors are special class member function, which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition.  
There are 3 types of constructors:

* Default constructors
* Parametrized constructors
* Copy constructors

// C++ program to demonstrate constructors

#include <bits/stdc++.h>

using namespace std;

class Geeks

{

    public:

    int id;

    //Default Constructor

    Geeks()

    {

        cout << "Default Constructor called" << endl;

        id=-1;

    }

    //Parametrized Constructor

    Geeks(int x)

    {

        cout << "Parametrized Constructor called" << endl;

        id=x;

    }

};

int main() {

    // obj1 will call Default Constructor

    Geeks obj1;

    cout << "Geek id is: " <<obj1.id << endl;

    // obj1 will call Parametrized Constructor

    Geeks obj2(21);

    cout << "Geek id is: " <<obj2.id << endl;

    return 0;

}

A **Copy Constructor** creates a new object, which is exact copy of the existing copy. The compiler provides a default Copy Constructor to all the classes.

class-name (class-name &){}

[**Destructors**](http://www.geeksforgeeks.org/destructors-c/) **:** Destructor isanother special member function that is called by the compiler when the scope of the object ends.

// C++ program to explain destructors

#include <bits/stdc++.h>

using namespace std;

class Geeks

{

    public:

    int id;

    //Definition for Destructor

    ~Geeks()

    {

        cout << "Destructor called for id: " << id <<endl;

    }

};

int main()

  {

    Geeks obj1;

    obj1.id=7;

    int i = 0;

    while ( i < 5 )

    {

        Geeks obj2;

        obj2.id=i;

        i++;

    } // Scope for obj2 ends here

    return 0;

  } // Scope for obj1 ends here

**Structure vs class.**

Members of a class are private by default and members of struct are public by default.

#include <stdio.h>

struct Test {

    int x; // x is public

};

int main()

{

  Test t;

  t.x = 20; // works fine because x is public

  getchar();

  return 0;

}

**Additional link** : <http://www.geeksforgeeks.org/g-fact-76/>

**Local Classes in C++**

Class declared inside a function becomes local to that function and is called Local Class in C++. For example, in the following program, Test is a local class in fun().

#include<iostream>

using namespace std;

void fun()

{

      class Test  // local to fun

      {

        /\* members of Test class \*/

      };

}

int main()

{

    return 0;

}

# **Nested Classes**

A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

#include<iostream>

using namespace std;

 /\* start of Enclosing class declaration \*/

class Enclosing {

   int x;

   /\* start of Nested class declaration \*/

   class Nested {

      int y;

      void NestedFun(Enclosing \*e) {

        cout<<e->x;  // works fine: nested class can access

                     // private members of Enclosing class

      }

   }; // declaration Nested class ends here

}; // declaration Enclosing class ends here

int main()

{

}

**Access Modifiers (Specifier)**

**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.

// 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;

}

**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 which gives compilation error

    obj.radius = 1.5;

    cout << "Area is:" << obj.compute\_area();

    return 0;

}

**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.

// 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 derived class can

    // access the protected data members of base class

    obj1.setId(81);

    obj1.displayId();

    return 0;

}

**=====================================================================**

**Section 15) Constructor and Destructor**

**Constructor**

A constructor is a member function of a class which initializes objects of a class. In C++,Constructor is automatically called when object(instance of class) create.It is special member function of the class.

A constructor is different from normal functions in following ways:

* Constructor has same name as the class itself
* Constructors don’t have return type
* A constructor is automatically called when an object is created.
* If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

**Types of Constructors**

**Default Constructors** :

Default constructor is the constructor which doesn’t take any argument. It has no parameters.

// Cpp program to illustrate the

// concept of Constructors

#include <iostream>

using namespace std;

class construct

{

public:

    int a, b;

    // Default Constructor

    construct()

    {

        a = 10;

        b = 20;

    }

};

int main()

{

    // Default constructor called automatically

    // when the object is created

    construct c;

    cout << "a: "<< c.a << endl << "b: "<< c.b;

    return 1;

}

- Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

**Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.

// CPP program to illustrate parameterized constructors

#include<iostream>

using namespace std;

 class Point

{

    private:

        int x, y;

    public:

        // Parameterized Constructor

        Point(int x1, int y1)

        {

            x = x1;

            y = y1;

        }

        int getX()

        {

            return x;

        }

        int getY()

        {

            return y;

        }

    };

int main()

{

    // Constructor called

    Point p1(10, 15);

    // Access values assigned by constructor

    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;

}

**Copy constructor :** A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

#include<iostream>

using namespace std;

class Point

{

private:

    int x, y;

public:

    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor

    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()            {  return x; }

    int getY()            {  return y; }

};

int main()

{

    Point p1(10, 15); // Normal constructor is called here

    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors

    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;

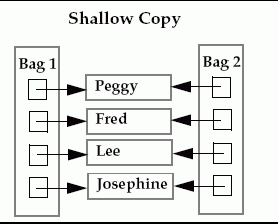
}

In C++, a Copy Constructor may be called in following cases:  
1. When an object of the class is returned by value.  
2. When an object of the class is passed (to a function) by value as an argument.  
3. When an object is constructed based on another object of the same class.  
4. When compiler generates a temporary object.

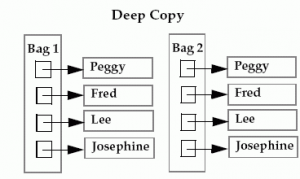
**When is user defined copy constructor needed?**

If we don’t define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member wise copy between objects. The compiler created copy constructor works fine in general. We need to define ohanur own copy constructor only if an object has pointers or any run time allocation of resource like file dle, a network connection..etc.

**Default constructor does only shallow copy.**



**Deep copy is possible only with user defined copy constructor.** In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.



Aditional link : <http://www.cpphub.com/2014/11/copy-constructor-shallow-and-deep.html>

**Destructor :**

Destructor is a member function which destructs or deletes an object.

**When is destructor called?**  
A destructor function is called automatically when the object goes out of scope:  
(1) the function ends  
(2) the program ends  
(3) a block containing local variables ends  
(4) a delete operator is called

**How destructors are different from a normal member function?**  
Destructors have same name as the class preceded by a tilde (~)  
Destructors don’t take any argument and don’t return anything

**When do we need to write a user-defined destructor**

If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

// Program 2

#include <stdio.h>

struct Test {

    int x; // x is public

};

int main()

{

  Test t;

  t.x = 20; // works fine because x is public

  getchar();

  return 0;

}

class String

{

private:

    char \*s;

    int size;

public:

    String(char \*); // constructor

    ~String();      // destructor

};

String::String(char \*c)

{

    size = strlen(c);

    s = new char[size+1];

    strcpy(s,c);

}

String::~String()

{

    delete []s;

}

# **Initialization of data members**

In C++, class variables are initialized in the same order as they appear in the class declaration.

Consider the below code.

#include<iostream>

using namespace std;

class Test {

  private:

    int y;

    int x;

  public:

    Test() : x(10), y(x + 10) {}

    void print();

};

void Test::print()

{

   cout<<"x = "<<x<<" y = "<<y;

}

int main()

{

    Test t;

    t.print();

    getchar();

    return 0;

}

# **When do we use Initializer List in C++**

**For initialization of non-static const data members:**  
const data members must be initialized using Initializer List. In the following example, “t” is a const data member of Test class and is initialized using Initializer List.

#include<iostream>

using namespace std;

class Test {

    const int t;

public:

    Test(int t):t(t) {}  //Initializer list must be used

    int getT() { return t; }

};

int main() {

    Test t1(10);

    cout<<t1.getT();

    return 0;

}

/\* OUTPUT:

   10

\*/

**For initialization of reference members:**  
Reference members must be initialized using Initializer List. In the following example, “t” is a reference member of Test class and is initialized using Initializer List.

// Initialization of reference data members

#include<iostream>

using namespace std;

 class Test {

    int &t;

public:

    Test(int &t):t(t) {}  //Initializer list must be used

    int getT() { return t; }

};

 int main() {

    int x = 20;

    Test t1(x);

    cout<<t1.getT()<<endl;

    x = 30;

    cout<<t1.getT()<<endl;

    return 0;

}

/\* OUTPUT:

    20

    30

 \*/

**3) For initialization of member objects which do not have default constructor:**  
In the following example, an object “a” of class “A” is data member of class “B”, and “A” doesn’t have default constructor. Initializer List must be used to initialize “a”.

#include <iostream>

using namespace std;

class A {

    int i;

public:

    A(int );

};

 A::A(int arg) {

    i = arg;

    cout << "A's Constructor called: Value of i: " << i << endl;

}

 // Class B contains object of A

class B {

    A a;

public:

    B(int );

};

 B::B(int x):a(x) {  //Initializer list must be used

    cout << "B's Constructor called";

}

 int main() {

    B obj(10);

    return 0;

}

/\* OUTPUT:

    A's Constructor called: Value of i: 10

    B's Constructor called

\*/

**For initialization of base class members :** Like point 3, parameterized constructor of base class can only be called using Initializer List.

#include <iostream>

using namespace std;

class A {

    int i;

public:

    A(int );

};

A::A(int arg) {

    i = arg;

    cout << "A's Constructor called: Value of i: " << i << endl;

}

// Class B is derived from A

class B: A {

public:

    B(int );

};

B::B(int x):A(x) { //Initializer list must be used

    cout << "B's Constructor called";

}

int main() {

    B obj(10);

    return 0;

}

**When constructor’s parameter name is same as data member**  
If constructor’s parameter name is same as data member name then the data member must be initialized either using [this pointer](http://msdn.microsoft.com/en-us/library/y0dddwwd.aspx) or Initializer List. In the following example, both member name and parameter name for A() is “i”.

#include <iostream>

using namespace std;

class A {

    int i;

public:

    A(int );

    int getI() const { return i; }

};

A::A(int i):i(i) { }  // Either Initializer list or this pointer must be used

/\* The above constructor can also be written as

A::A(int i) {

    this->i = i;

}

\*/

 int main() {

    A a(10);

    cout<<a.getI();

    return 0;

}

/\* OUTPUT:

    10

\*/

**For Performance reasons:**  
It is better to initialize all class variables in Initializer List instead of assigning values inside body. Consider the following example:

// Without Initializer List

class MyClass {

    Type variable;

public:

    MyClass(Type a) {  // Assume that Type is an already

                     // declared class and it has appropriate

                     // constructors and operators

      variable = a;

    }

};

**Use of explicit keyword in C++**

Refer link : <http://www.geeksforgeeks.org/g-fact-93/>

=================================================================

**Section 16 ) Static data member and member function**

**Static Data member :**

When a **data member** is declared as **static**, only one copy of the **data** is maintained for all objects of the class. Static data members are not part of objects of a given class type. As a result, the declaration of a static data member is not considered a definition. The data member is declared in class scope, but definition is performed at file scope. These static members have external linkage.

#include <iostream>

using namespace std;

class A

{

    int x;

public:

    A() { cout << "A's constructor called " << endl;  }

};

class B

{

    static A a;

public:

    B() { cout << "B's constructor called " << endl; }

    static A getA() { return a; }

};

int main()

{

    B b;

    A a = b.getA();

    return 0;

}

Compiler Error: undefined reference to `B::a'

If we add definition of a, the program will works fine and will call A’s constructor. See the following program.

#include <iostream>

using namespace std;

class A

{

    int x;

public:

    A() { cout << "A's constructor called " << endl;  }

};

class B

{

    static A a;

public:

    B() { cout << "B's constructor called " << endl; }

    static A getA() { return a; }

};

A B::a;  // definition of a

int main()

{

    B b1, b2, b3;

    A a = b1.getA();

    return 0;

}

- The static keyword is only used with the declaration of a static member, inside the class definition, but not with the definition of that static member:

**Static Member function**

**Static member functions do not have this pointer.**  
For example following program fails in compilation with error *“`this’ is unavailable for static member functions “*

#include<iostream>

class Test {

   static Test \* fun() {

     return this; // compiler error

   }

};

int main()

{

   getchar();

   return 0;

}

**- A static member function cannot be virtual**

- **Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.**  
For example, following program fails in compilation with error “*‘void Test::fun()’ and `static void Test::fun()’ cannot be overloaded* ”

#include<iostream>

class Test {

   static void fun() {}

   void fun() {} // compiler error

};

int main()

{

   getchar();

   return 0;

}

- **A static member function can not be declared *const*, *volatile*, or *const volatile***

***- A static member can not access non static data member because they does not have this pointer.***

***================================================================***

**Section 17 ) Constant data member and member function**

**Constant data memeber**

**const** data members are not assigned values during its declaration. Const data members are assigned values in the constructor.

#include <iostream>

using namespace std;

class A

{

const int x;

public:

A(int y)

{

x = y;

}

};

int main()

{

A a(5);

return 0;

}

**Initialization of constant data member**

Constant data member need to be intialized using constructor initialization list and those value should not be change.

*//Example of constant member*

*struct* A{

*const* *int* a;

A(*int* b):a(b){}

};

**Constant Member Function of Class**

A **const** member function cannot change the value of any data member of the class and cannot call any member function which is not constant.

class A

{

public:

int x;

void func() const

{

x = 0; // this will give compilation error

}

};

- A const object can only call a const member function

- We cannot make constructors const

**=================================================================**

**Section 18 ) Friend class and function**

**Friend Class**

A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

class Node

{

private:

  int key;

  Node \*next;

  /\* Other members of Node Class \*/

  friend class LinkedList; // Now class  LinkedList can

                           // access private members of Node

};

**Friend Function**

Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:  
a) A method of another class  
b) A global function

class Node

{

private:

  int key;

  Node \*next;

  /\* Other members of Node Class \*/

  friend int LinkedList::search(); // Only search() of linkedList

                                  // can access internal members

};

Following are some important points about friend functions and classes:  
**1)** Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

**2)** Friendship is not mutual. If a class A is friend of B, then B doesn’t become friend of A automatically.

**3)** Friendship is not inherited

**A simple and complete C++ program to demonstrate friend Class**

#include <iostream>

class A {

private:

    int a;

public:

    A() { a=0; }

    friend class B;     // Friend Class

};

class B {

private:

    int b;

public:

    void showA(A& x) {

        // Since B is friend of A, it can access

        // private members of A

        std::cout << "A::a=" << x.a;

    }

};

int main() {

   A a;

   B b;

   b.showA(a);

   return 0;

}

**Friend function is member of another class**

**A simple and complete C++ program to demonstrate friend function of another class**

#include <iostream>

class B;

class A

{

public:

    void showB(B& );

};

class B

{

private:

    int b;

public:

    B()  {  b = 0; }

    friend void A::showB(B& x); // Friend function

};

void A::showB(B &x)

{

    // Since show() is friend of B, it can

    // access private members of B

    std::cout << "B::b = " << x.b;

}

int main()

{

    A a;

    B x;

    a.showB(x);

    return 0;

}

**Global friend function**

**A simple and complete C++ program to demonstrate global friend**

#include <iostream>

class A

{

    int a;

public:

    A() {a = 0;}

    friend void showA(A&); // global friend function

};

void showA(A& x) {

    // Since showA() is a friend, it can access

    // private members of A

    std::cout << "A::a=" << x.a;

}

int main()

{

    A a;

    showA(a);

    return 0;

}

**Additional link** : <http://www.geeksforgeeks.org/friend-class-function-cpp/>

**=================================================================**

**Section 19) Function overloading & Operator overloading**

**Function Overloading**

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.

Function overloading can be considered as an example of polymorphism feature in C++.

Following is a simple C++ example to demonstrate function overloading.

#include <iostream>

using namespace std;

void print(int i) {

  cout << " Here is int " << i << endl;

}

void print(double  f) {

  cout << " Here is float " << f << endl;

}

void print(char\* c) {

  cout << " Here is char\* " << c << endl;

}

int main() {

  print(10);

  print(10.10);

  print("ten");

  return 0;

}

**Only those function can be overloaded which has same name but different by**

- Order of arguments.

- Types of aruments.

- Number of argument.

Example is overload constructor.

# **Functions that cannot be overloaded in C++**

1) Function declarations that differ only in the return type

2) Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration. For example, following program fails in compilation.

|  |
| --- |
|  |

3) Parameter declarations that differ only in a pointer \* versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration. Only the second and subsequent array dimensions are significant in parameter types. For example, following two function declarations are equivalent.

int fun(int \*ptr);

int fun(int ptr[]); // redeclaration of fun(int \*ptr)

4) Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.

void h(int ());

void h(int (\*)()); // redeclaration of h(int())

5) Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the const and volatile type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. For example, following program fails in compilation with error *“redefinition of `int f(int)’ “*

Example:

#include<iostream>

#include<stdio.h>

using namespace std;

int f ( int x) {

    return x+10;

}

int f ( const int x) {

    return x+10;

}

int main() {

  getchar();

  return 0;

}

6) Two parameter declarations that differ only in their default arguments are equivalent. For example, following program fails in compilation with error *“redefinition of `int f(int, int)’ “*

#include<iostream>

#include<stdio.h>

using namespace std;

int f ( int x, int y) {

    return x+10;

}

int f ( int x, int y = 10) {

    return x+y;

}

int main() {

  getchar();

  return 0;

}

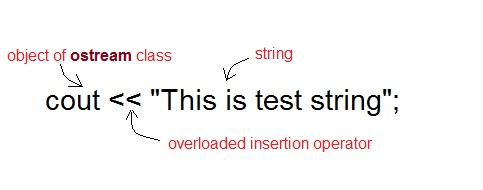
**Operator Overloading**

- It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it.

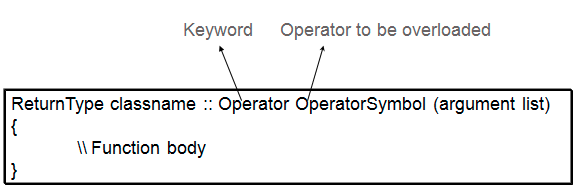
- In C++, we can make operators to work for user defined classes.

- For example, we can overload an operator ‘+’ in a class like String so that we can concatenate two strings by just using +.

- Classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer.



#### Operator Overloading Syntax



#include<iostream>

using namespace std;

class Complex {

private:

    int real, imag;

public:

    Complex(int r = 0, int i =0)  {real = r;   imag = i;}

    // This is automatically called when '+' is used with

    // between two Complex objects

    Complex operator + (Complex const &obj) {

         Complex res;

         res.real = real + obj.real;

         res.imag = imag + obj.imag;

         return res;

    }

    void print() { cout << real << " + i" << imag << endl; }

};

int main()

{

    Complex c1(10, 5), c2(2, 4);

    Complex c3 = c1 + c2; // An example call to "operator+"

    c3.print();

}

**Implementing Operator Overloading**

Operator overloading can be done by implementing a function which can be :

Member Function

1. Non-Member Function
2. Friend Function

**Operator Overloading using member function:**

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.

**For Unary operator** : No parameters are required

**For Binary operator** : One parameter is required

**Operator Overloading using friend function.**

Operator overloading function can be made friend function if it needs access to the private and protected members of class.

**For Unary operator** : One parameter is required

**For Binary operator** : Two parameters are required

#### Example: overloading '+' Operator to add two time object

#include< iostream.h>

#include< conio.h>

class time

{

int h,m,s;

public:

time()

{

h=0, m=0; s=0;

}

void getTime();

void show()

{

cout<< h<< ":"<< m<< ":"<< s;

}

**time operator+(time);** //overloading '+' operator

};

**time time::operator+(time t1)** //operator function

{

time t;

int a,b;

a=s+t1.s;

t.s=a%60;

b=(a/60)+m+t1.m;

t.m=b%60;

t.h=(b/60)+h+t1.h;

t.h=t.h%12;

return t;

}

void time::getTime()

{

cout<<"\n Enter the hour(0-11) ";

cin>>h;

cout<<"\n Enter the minute(0-59) ";

cin>>m;

cout<<"\n Enter the second(0-59) ";

cin>>s;

}

void main()

{

clrscr();

time t1,t2,t3;

cout<<"\n Enter the first time ";

t1.getTime();

cout<<"\n Enter the second time ";

t2.getTime();

t3=t1+t2; //adding of two time object using '+' operator

cout<<"\n First time ";

t1.show();

cout<<"\n Second time ";

t2.show();

cout<<"\n Sum of times ";

t3.show();

getch();

}

# **Overloading I/O operator ('<<' & '>>')**

#include <iostream>

using namespace std;

class Complex

{

private:

    int real, imag;

public:

    Complex(int r = 0, int i =0)

    {  real = r;   imag = i; }

    friend ostream & operator << (ostream &out, const Complex &c);

    friend istream & operator >> (istream &in,  Complex &c);

};

ostream & operator << (ostream &out, const Complex &c)

{

    out << c.real;

    out << "+i" << c.imag << endl;

    return out;

}

istream & operator >> (istream &in,  Complex &c)

{

    cout << "Enter Real Part ";

    in >> c.real;

    cout << "Enter Imagenory Part ";

    in >> c.imag;

    return in;

}

int main()

{

   Complex c1;

   cin >> c1;

   cout << "The complex object is ";

   cout << c1;

   return 0;

}

**Overloading Subscript or array index operator []**

using namespace std;

// A class to represent an integer array

class Array

{

private:

    int \*ptr;

    int size;

public:

    Array(int \*, int);

    // Overloading [] operator to access elements in array style

    int &operator[] (int);

    // Utility function to print contents

    void print() const;

};

// Implementation of [] operator.  This function must return a

// refernce as array element can be put on left side

int &Array::operator[](int index)

{

    if (index >= size)

    {

        cout << "Array index out of bound, exiting";

        exit(0);

    }

    return ptr[index];

}

// constructor for array class

Array::Array(int \*p = NULL, int s = 0)

{

    size = s;

    ptr = NULL;

    if (s != 0)

    {

        ptr = new int[s];

        for (int i = 0; i < s; i++)

            ptr[i] = p[i];

    }

}

void Array::print() const

{

    for(int i = 0; i < size; i++)

        cout<<ptr[i]<<" ";

    cout<<endl;

}

// Driver program to test above methods

int main()

{

    int a[] = {1, 2, 4, 5};

    Array arr1(a, 4);

    arr1[2] = 6;

    arr1.print();

    arr1[8] = 6;

    return 0;

}

**Example : Overload “=”(Assignment) operator**

#include <iostream>

using namespace std;

class Distance {

private:

int feet; // 0 to infinite

int inches; // 0 to 12

public:

// required constructors

Distance() {

feet = 0;

inches = 0;

}

Distance(int f, int i) {

feet = f;

inches = i;

}

void operator = (const Distance &D ) {

feet = D.feet;

inches = D.inches;

}

// method to display distance

void displayDistance() {

cout << "F: " << feet << " I:" << inches << endl;

}

};

int main() {

Distance D1(11, 10), D2(5, 11);

cout << "First Distance : ";

D1.displayDistance();

cout << "Second Distance :";

D2.displayDistance();

// use assignment operator

D1 = D2;

cout << "First Distance :";

D1.displayDistance();

return 0;

}

**Copy constructor Vs. Assignment operator**

**Assignment operator** is used to copy the values from one object to another **already existing object**.

time tm(3,15,45); //**tm** object created and initialized

time t1; //**t1** object created

t1 = tm; //initializing **t1** using **tm**

**Copy constructor** is a special constructor that initializes a **new object** from an existing object.

time tm(3,15,45); //**tm** object created and initialized

time t1(tm); //**t1** object created and initialized using **tm** object

**Operator's which can not be overloaded**.

In C++, following operators can not be overloaded:

. (Member Access or Dot operator)  
?: (Ternary or Conditional Operator )  
:: (Scope Resolution Operator)  
.\* (Pointer-to-member Operator )  
[sizeof](http://en.wikipedia.org/wiki/Sizeof) (Object size Operator)  
[typeid](http://msdn.microsoft.com/en-us/library/fyf39xec(VS.80).aspx) (Object type Operator)

**Restrictions on Operator Overloading**

Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and Associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of a procedure. You cannot change how integers are added.

**Smart Pointers :**

Using smart pointers, we can make pointers to work in way that we don’t need to explicitly call delete. Smart pointer is a wrapper class over a pointer with operator like \* and -> overloaded. The objects of smart pointer class look like pointer, but can do many things that a normal pointer can’t like automatic destruction (yes, we don’t have to explicitly use delete), reference counting and more.

**Refer link** : <https://www.codeproject.com/Articles/15351/Implementing-a-simple-smart-pointer-in-c>

**========================================================**

**Section 20) Inheritance**

**What is inheritance**

Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

**Base and Derived Classes**

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes.

Syntex :

class **derived-class**: access-specifier **base-class**

#include <iostream>

using namespace std;

// Base class

class Shape {

public:

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

// Derived class

class Rectangle: public Shape {

public:

int getArea() {

return (width \* height);

}

};

int main(void) {

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

return 0;

}

**Access Control and Inheritance**

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way −

|  |  |  |  |
| --- | --- | --- | --- |
| **Access** | **public** | **protected** | **private** |
| Same class | yes | yes | yes |
| Derived classes | yes | yes | no |
| Outside classes | yes | no | no |

**Mode of Inheritance**

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier

**Public Inheritance** − When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.

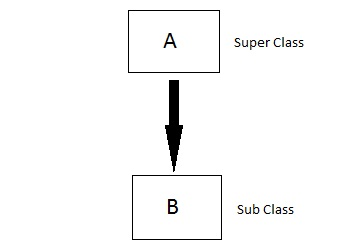
**Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.

**Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

**Types of inhertance**

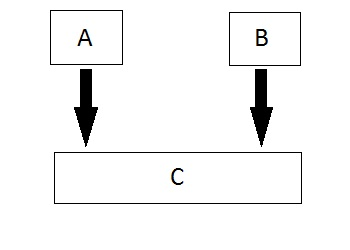
**1) Single level inheritance**

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



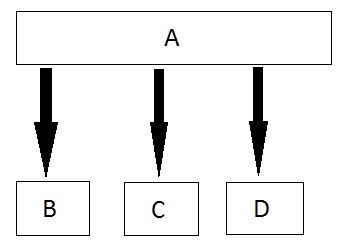
**2) Multiple Inheritance**

In this type of inheritance a single derived class may inherit from two or more than two base classes.



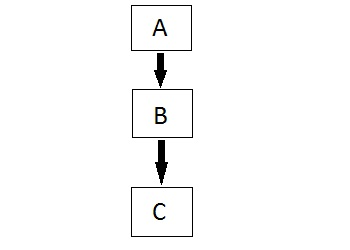
3) **Hierrarchical Inhertance**

In this type of inheritance, multiple derived classes inherits from a single base class.

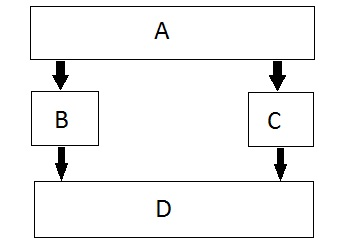


4) **Multilevel Inheritance**

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



**5) Hybrid (Virtual) Inheritance**



**Order of Constructor and destructor function call.**

Invocation of constructors and destructors depends on the type of inheritance being implemented. We have presented you the sequence in which constructors and destructors get called in single and multiple inheritance.

Base class constructors are called first and the derived class constructors are called next in single inheritance

Destructor is called in reverse sequence of constructor invocation i.e. The destructor of the derived class is called first and the destructor of the base is called next.

//example of constructor and destructor in multiple inheritance

#include<iostream>

using namespace std;

class base\_one

{

public:

    base\_one()

    {

        cout<<"base\_one class constructor"<<endl;

    }

    ~base\_one()

    {

        cout<<"base\_one class destructor"<<endl;

    }

};

class base\_two

{

public:

    base\_two()

    {

        cout<<"base\_two class constructor"<<endl;

    }

    ~base\_two()

    {

        cout<<"base\_two class destructor"<<endl;

    }

};

class derived:public base\_one, public base\_two

{

public:

    derived()

    {

        cout<<"derived class constructor"<<endl;

    }

    ~derived()

    {

        cout<<"derived class destructor"<<endl;

    }

};

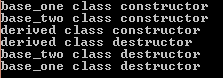
int main()

{

    derived d;

    return 0;

}



The output of the program is:

**Constructor initializer list**

class A

{

protected:

A(int a, int b) : a(a), b(b) {} // Accessible to derived classes

// Change "protected" to "public" to allow others to instantiate A.

private:

int a, b; // Keep these variables private in A

};

class B : public A

{

public:

B() : A(0, 0) // Calls A's constructor, initializing a and b in A to 0.

{

}

};

# **Object Slicing:**

Object slicing happens when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off to form the base class object.

#include <iostream>

using namespace std;

class Base

{

protected:

    int i;

public:

    Base(int a)     { i = a; }

    virtual void display()

    { cout << "I am Base class object, i = " << i << endl; }

};

class Derived : public Base

{

    int j;

public:

    Derived(int a, int b) : Base(a) { j = b; }

    virtual void display()

    { cout << "I am Derived class object, i = "

           << i << ", j = " << j << endl;  }

};

// Global method, Base class object is passed by value

void somefunc (Base obj)

{

    obj.display();

}

int main()

{

    Base b(33);

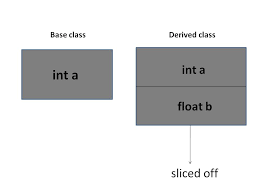
    Derived d(45, 54);

    somefunc(b);

    somefunc(d);  // Object Slicing, the member j of d is sliced off

    return 0;

}



**Inheritance and friendship**

In C++, friendship is not inherited. If a base class has a friend function, then the function doesn’t become a friend of the derived class.

For example, following program prints error because *show()* which is a friend of base class *A* tries to access private data of derived class *B*.

#include <iostream>

using namespace std;

class A

{

  protected:

    int x;

  public:

    A() { x = 0;}

  friend void show();

};

class B: public A

{

  public:

    B() : y (0) {}

  private:

    int y;

};

void show()

{

  B b;

  cout << "The default value of A::x = " << b.x;

  // Can't access private member declared in class 'B'

  cout << "The default value of B::y = " << b.y;

}

int main()

{

  show();

  getchar();

  return 0;

}

=======================================================================

**Section 21) Virtual function**

**Virtual Functions and Runtime Polymorphism**

Consider the following simple program which is an example of runtime polymorphism.  
The main thing to note about the program is, derived class function is called using a base class pointer. The idea is, virtual functions are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.

#include<iostream>

using namespace std;

class Base

{

public:

    virtual void show() { cout<<" In Base \n"; }

};

class Derived: public Base

{

public:

    void show() { cout<<"In Derived \n"; }

};

int main(void)

{

    Base \*bp = new Derived;

    bp->show();  // RUN-TIME POLYMORPHISM

    return 0;

}

**What is use of virtual function**

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class *Employee* , the class contains virtual functions like *raiseSalary()*, *transfer()*, *promote()*,.. etc. Different types of employees like *Manager*, *Engineer*, ..etc may have their own implementations of the virtual functions present in base class *Employee*. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise salary of all employees by iterating through list of employees. Every type of employee may have its own logic in its class, we don’t need to worry because if *raiseSalary()* is present for a specific employee type, only that function would be called.

class Employee

{

public:

    virtual void raiseSalary()

    {  /\* common raise salary code \*/  }

    virtual void promote()

    { /\* common promote code \*/ }

};

class Manager: public Employee {

    virtual void raiseSalary()

    {  /\* Manager specific raise salary code, may contain

          increment of manager specific incentives\*/  }

    virtual void promote()

    { /\* Manager specific promote \*/ }

};

// Similarly, there may be other types of employees

// We need a very simple function to increment salary of all employees

// Note that emp[] is an array of pointers and actual pointed objects can

// be any type of employees. This function should ideally be in a class

// like Organization, we have made it global to keep things simple

void globalRaiseSalary(Employee \*emp[], int n)

{

    for (int i = 0; i < n; i++)

        emp[i]->raiseSalary(); // Polymorphic Call: Calls raiseSalary()

                               // according to the actual object, not

                               // according to the type of pointer

}

**VTABLE**

- Every class that uses virtual functions (or is derived from a class that uses virtual functions) is given it's own virtual table as a secret data member.  
  
- This table is set up by the compiler at compile time.  
  
- A virtual table contains one entry as a function pointer for each virtual function that can be called by objects of the class.  
  
- Virtual table stores NULL pointer to pure virtual functions in ABC.  
  
- Virtual Table is created even for classes that have virtual base classes. In this case, the vtable has pointer to the shared instance of the base class along with the pointers to the classe's virtual functions if any.

**Vptr**

- This vtable pointer or \_vptr, is a hidden pointer added by the Compiler to the base class. And this pointer is pointing to the virtual table of that particular class.

- This \_vptr is inherited to all the derived classes.

- Each object of a class with virtual functions transparently stores this \_vptr.

- Call to a virtual function by an object is resolved by following this hidden \_vptr.

**Example** :-

#include<iostream.h>

class Base

{

public:

virtual void function1() {cout<<"Base :: function1()\n";};

virtual void function2() {cout<<"Base :: function2()\n";};

virtual ~Base(){};

};

class D1: public Base

{

public:

~D1(){};

virtual void function1() { cout<<"D1 :: function1()\n";};

};

class D2: public Base

{

public:

~D2(){};

virtual void function2() { cout<< "D2 :: function2\n";};

};

int main()

{

D1 \*d = new D1;;

Base \*b = d;

b->function1();

b->function2();

delete (b);

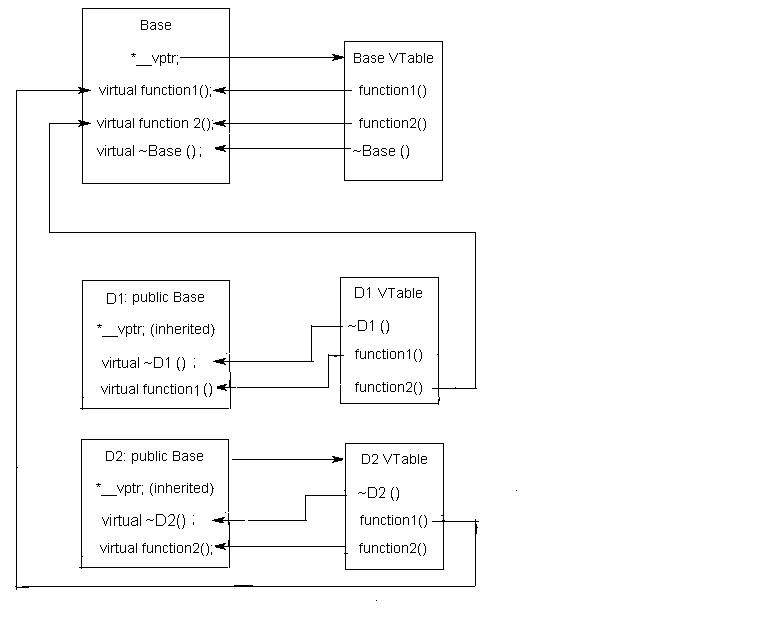
return (0);

}

output:

D1 :: function1()

Base :: function2()



**Explanation:**Here in function main b pointer gets assigned to D1's \_vptr and now starts pointing to D1's vtable. Then calling to a function1(), makes it's \_vptr startightway calls D1's vtable function1() and so in turn calls D1's method i.e. function1() as D1 has it's own function1() defined it's class.  
  
Where as pointer b calling to a function2(), makes it's \_vptr points to D1's vatble which in-turn pointing to Base class's vtable function2 () as shown in the diagram (as D1 class does not have it's own definition or function2()).  
  
So, now calling delete on pointer b follows the \_vptr - which is pointing to D1's vtable calls it's own class's destructor i.e. D1 class's destructor and then calls the destrcutor of Base class - this as part of when dervied object gets deleted it turn deletes it's emebeded base object. Thats why we must always make Base class's destrcutor as virtual if it has any virtual functions in it.

**Default arguments and virtual function**

**#include <iostream>**

using namespace std;

class Base

{

public:

    virtual void fun ( int x = 0 )

    {

        cout << "Base::fun(), x = " << x << endl;

    }

};

class Derived : public Base

{

public:

    virtual void fun ( int x )

    {

        cout << "Derived::fun(), x = " << x << endl;

    }

};

int main()

{

    Derived d1;

    Base \*bp = &d1;

    bp->fun();

    return 0;

}

Default arguments do not participate in signature of functions. So signatures of fun() in base class and derived class are considered same, hence the fun() of base class is overridden.

**Virtual functions in derived classes**

In C++, once a member function is declared as a virtual function in a base class, it becomes virtual in every class derived from that base

#include<iostream>

using namespace std;

class A {

  public:

    virtual void fun()

    { cout<<"\n A::fun() called ";}

};

class B: public A {

  public:

    void fun()

    { cout<<"\n B::fun() called "; }

};

class C: public B {

  public:

    void fun()

    { cout<<"\n C::fun() called "; }

};

int main()

{

   C c; // An object of class C

   B \*b = &c; // A pointer of type B\* pointing to c

   b->fun();  // this line prints "C::fun() called"

   getchar();

   return 0;

}

**Abstract class :**

An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier ( = 0 ) in the declaration of a virtual member function in the class declaration.

**#include<iostream>**

using namespace std;

class Base

{

   int x;

public:

    virtual void fun() = 0;

    int getX() { return x; }

};

// This class ingerits from Base and implements fun()

class Derived: public Base

{

    int y;

public:

    void fun() { cout << "fun() called"; }

};

int main(void)

{

    Derived d;

    d.fun();

    return 0;

}

**-** *A class is abstract if it has at least one pure virtual function.*

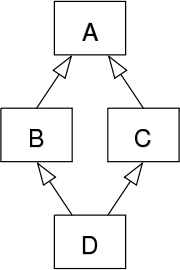
*- If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.*

*- We can have pointers and references of abstract class type.*

*- An abstract class can have constructors.*

**Why we require abstract class :**

Sometimes implementation of all function cannot be provided in a base **class** because **we** don't know the implementation. ... **We** cannot create objects of **abstract classes**. A pure virtual function (or **abstract** function) in **C++** is a virtual function for which **we** don't have implementation, **we** only declare it.

**Hybried inheritence (diamond problem)**

Suppose we have 2 classes B and C that derive from the *same* class – in our example above it would be class A. We also have class D that derives from *both* B and C by using multiple inheritance. You can see in the figure above that the classes essentially form the shape of a diamond – which is why this problem is called the diamond problem

/\*

The Animal class below corresponds to class

A in our graphic above

\*/

class Animal { /\* ... \*/ }; // base class

{

int weight;

public:

int getWeight() { return weight;};

};

class Tiger : public Animal { /\* ... \*/ };

class Lion : public Animal { /\* ... \*/ }

class Liger : public Tiger, public Lion { /\* ... \*/ };

In the code above, we’ve given a more concrete example of the diamond problem. The Animal class corresponds to the topmost class in the hierarchy (A in our graphic above), Tiger and Lion respectively correspond to B and C in the graphic, and the Liger class corresponds to D.

Now, the question is what is the problem with having an inheritance hierarcy like this. Take a look at the code below so that we can best answer that question:

In our inheritance hierarchy, we can see that both the Tiger and Lion classes derive from the Animal base class. And here is the problem: **because Liger derives from both the Tiger and Lion classes – which each have their own copy of the data members and methods of the Animal class- the Liger object "lg" will contain** *two* **subobjects of the** *Animal* **base class.**

|  |
| --- |
|  |

So, you ask, what’s the problem with a Liger object having 2 **sub**-objects of the Animal class? Take another look at the code above – the call "lg.getWeight()" will result in a compiler error. This is because the compiler **does not know whether the call to getWeight refers to the copy of getWeight that the Liger object lg inherited through the Lion class or the copy that lg inherited through the Tiger class**. So, the call to getWeight in the code above is ambiguous and will not get past the compiler.

## Solution to the Diamond Problem

We’ve given an explanation of the diamond problem, but now we want to give you a solution to the diamond problem. If the inheritance from the Animal class to both the Lion class and the Tiger class is marked as virtual, then C++ will ensure that only one subobject of the Animal class will be created for every Liger object. This is what the code for that would look like:

class Tiger : virtual public Animal { /\* ... \*/ };

class Lion : virtual public Animal { /\* ... \*/ }

int main( )

{

Liger lg ;

/\*THIS CODE WILL NOW COMPILE OK NOW THAT WE'VE

USED THE VIRTUAL KEYWORD IN THE TIGER AND LION

CLASS DECLARATIONS \*/

int weight = lg.getWeight();

}

===================================================================

**Section 22) RTTI (Run-time type Information) & Type conversions**

RTTI (Run-time type information) is a mechanism that exposes information about an object’s data type at runtime and is available only for the classes which have at least one virtual function. It allows the type of an object to be determined during program execution

**dynamic\_cast :**

dynamic\_cast can only be used with pointers and references to classes (or with void\*). Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.

This naturally includes *pointer upcast* (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an *implicit conversion*.  
But dynamic\_cast can also *downcast* (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if -and only if- the pointed object is a valid complete object of the target type.

*// dynamic\_cast*

#include <iostream>

#include <exception>

*using* *namespace* std;

*class* Base { *virtual* *void* dummy() {} };

*class* Derived: *public* Base { *int* a; };

*int* main () {

*try* {

Base \* pba = *new* Derived;

Base \* pbb = *new* Base;

Derived \* pd;

pd = *dynamic\_cast*<Derived\*>(pba);

*if* (pd==0) cout << "Null pointer on first type-cast.\n";

pd = *dynamic\_cast*<Derived\*>(pbb);

*if* (pd==0) cout << "Null pointer on second type-cast.\n";

} *catch* (exception& e) {cout << "Exception: " << e.what();}

*return* 0;

}

When dynamic\_cast cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a *null pointer* to indicate the failure. If dynamic\_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad\_cast is thrown instead.

<http://www.cplusplus.com/doc/tutorial/typecasting/>

**static\_cast :**

static\_cast can perform conversions between pointers to related classes, not only *upcasts* (from pointer-to-derived to pointer-to-base), but also *downcasts* (from pointer-to-base to pointer-to-derived). No checks are performed during runtime to guarantee that the object being converted is in fact a full object of the destination type

Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, it does not incur the overhead of the type-safety checks of dynamic\_cast.

***class* Base {};**

*class* Derived: *public* Base {};

Base \* a = *new* Base;

Derived \* b = *static\_cast*<Derived\*>(a);

Therefore, static\_cast is able to perform with pointers to classes not only the conversions allowed implicitly, but also their opposite conversions.

static\_cast is also able to perform all conversions allowed implicitly (not only those with pointers to classes), and is also able to perform the opposite of these. It can:

* Convert from void\* to any pointer type. In this case, it guarantees that if the void\* value was obtained by converting from that same pointer type, the resulting pointer value is the same.
* Convert integers, floating-point values and enum types to enum types.

Additionally, static\_cast can also perform the following:

* Explicitly call a single-argument constructor or a conversion operator.
* Convert to *rvalue references*.
* Convert enum class values into integers or floating-point values.
* Convert any type to void, evaluating and discarding the value.

**reinterpret\_cast**

reinterpret\_cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it (such as [intptr\_t](http://www.cplusplus.com/intptr_t)), is guaranteed to be able to be cast back to a valid pointer.

The conversions that can be performed by reinterpret\_cast but not by static\_cast are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable. For example:

***class* A { */\* ... \*/* };**

*class* B { */\* ... \*/* };

A \* a = *new* A;

B \* b = *reinterpret\_cast*<B\*>(a);

**const\_cast**

This type of casting manipulates the constness of the object pointed by a pointer, either to be set or to be removed. For example, in order to pass a const pointer to a function that expects a non-const argument:

*// const\_cast*

#include <iostream>

*using* *namespace* std;

*void* print (*char* \* str)

{

cout << str << '\n';

}

*int* main () {

*const* *char* \* c = "sample text";

print ( *const\_cast*<*char* \*> (c) );

*return* 0;

}

**typeid**

typeid allows to check the type of an expression:

typeid (expression)

This operator returns a reference to a constant object of type type\_info that is defined in the standard header <typeinfo>. A value returned by typeid can be compared with another value returned by typeid using operators == and != or can serve to obtain a null-terminated character sequence representing the data type or class name by using its name() member.

*// typeid*

#include <iostream>

#include <typeinfo>

*using* *namespace* std;

*int* main () {

*int* \* a,b;

a=0; b=0;

*if* (*typeid*(a) != *typeid*(b))

{

cout << "a and b are of different types:\n";

cout << "a is: " << *typeid*(a).name() << '\n';

cout << "b is: " << *typeid*(b).name() << '\n';

}

*return* 0;

}

When typeid is applied to classes, typeid uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

*// typeid, polymorphic class*

#include <iostream>

#include <typeinfo>

#include <exception>

*using* *namespace* std;

*class* Base { *virtual* *void* f(){} };

*class* Derived : *public* Base {};

*int* main () {

*try* {

Base\* a = *new* Base;

Base\* b = *new* Derived;

cout << "a is: " << *typeid*(a).name() << '\n';

cout << "b is: " << *typeid*(b).name() << '\n';

cout << "\*a is: " << *typeid*(\*a).name() << '\n';

cout << "\*b is: " << *typeid*(\*b).name() << '\n';

} *catch* (exception& e) { cout << "Exception: " << e.what() << '\n'; }

*return* 0;

}

=========================================================================

**Section 23) Namespace**

**Defination** : Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

- Namespace is a feature added in C++ and not present in C.

- A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.

- Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

namespace namespace\_name

{

int x, y; // code declarations where

// x and y are declared in

// namespace\_name's scope

}

- Namespace declarations appear only at global scope.

- Namespace declarations can be nested within another namespace.

- Namespace declarations don’t have access specifiers. (Public or private)

- No need to give semicolon after the closing brace of definition of namespace.

- We can split the definition of namespace over several units.

// Creating namespaces

#include <iostream>

using namespace std;

namespace ns1

{

    int value()    { return 5; }

}

namespace ns2

{

    const double x = 100;

    double value() {  return 2\*x; }

}

int main()

{

    // Access value function within ns1

    cout << ns1::value() << '\n';

    // Access value function within ns2

    cout << ns2::value() << '\n';

    // Access variable x directly

    cout << ns2::x << '\n';

    return 0;

}

**Classes and Namespace:**

// A C++ program to demonstrate use of class

// in a namespace

#include <iostream>

using namespace std;

namespace ns

{

    // A Class in a namespace

    class geek

    {

    public:

        void display()

        {

            cout << "ns::geek::display()\n";

        }

    };

}

int main()

{

    // Creating Object of student Class

    ns::geek obj;

    obj.display();

    return 0;

}

**Class can also be declared inside namespace and defined outside namespace** using following syntax

// A C++ program to demonstrate use of class

// in a namespace

#include <iostream>

using namespace std;

namespace ns

{

    // Only declaring class here

    class geek;

}

// Defining class outside

class ns::geek

{

public:

    void display()

    {

        cout << "ns::geek::display()\n";

    }

};

int main()

{

    //Creating Object of student Class

    ns::geek obj;

    obj.display();

    return 0;

}

**Function within Namespace:**

We can **define methods also outside the namespace**. Following is an example code.

// A C++ code to demonstrate that we can define

// methods outside namespace.

#include <iostream>

using namespace std;

// Creating a namespace

namespace ns

{

    void display();

    class geek

    {

    public:

       void display();

    };

}

// Defining methods of namespace

void ns::geek::display()

{

    cout << "ns::geek::display()\n";

}

void ns::display()

{

    cout << "ns::display()\n";

}

// Driver code

int main()

{

    ns::geek obj;

    ns::display();

    obj.display();

    return 0;

}

**Unnamed Namespaces**

* They are directly usable in the same program and are used for declaring unique identifiers.
* In unnamed namespaces, name of the namespace in not mentioned in the declaration of namespace.
* The name of the namespace is uniquely generated by the compiler.
* The unnamed namespaces you have created will only be accessible within the file you created it in.
* Unnamed namespaces are the replacement for the static declaration of variables.

// C++ program to demonstrate working of unnamed

// namespaces

#include <iostream>

using namespace std;

// unnamed namespace declaration

namespace

{

   int rel = 300;

}

int main()

{

   cout << rel << "\n"; // prints 300

   return 0;

}

**Accessing, creating header, nesting and aliasing**

**Normal way**

// C++ program to demonstrate accessing of variables

// in normal way, i.e., using "::"

#include <iostream>

using namespace std;

namespace geek

{

    int rel = 300;

}

int main()

{

    // variable ‘rel’ accessed

    // using scope resolution operator

    cout << geek::rel << "\n";  // prints 300

    return 0;

}

**“using” directive**

// C++ program to demonstrate accessing of variables

// in normal way, i.e., using "using" directive

#include <iostream>

using namespace std;

namespace geek

{

    int rel = 300;

}

// use of ‘using’ directive

using namespace geek;

int main()

{

   // variable ‘rel’ accessed

   // without using scope resolution variable

   cout << rel << "\n";        //prints 300

   return 0;

}

|  |
| --- |
|  |

**Using namespace in header files**

We can create namespace in one file and access contents using another program. This is done in the following manner.

* We need to create two files. One containing the namespace and all the data members and member functions we want to use later.
* And the other program can directly call the first program to use all the data members and member functions in it.

**File 1**

|  |
| --- |
| // file1.h  namespace foo  {      int value()      {         return 5;      }  } |

**File 2**

|  |
| --- |
| // file2.cpp - Not to be executed online  #include <iostream>  #include “file1.h” // Including file1  using namespace std;    int main ()  {      cout << foo::value();      return 0;  } |

**Nested Namespaces**

// C++ program to demonstrate nesting of namespaces

#include <iostream>

using namespace std;

// Nested namespace

namespace out

{

  int val = 5;

  namespace in

  {

      int val2 = val;

  }

}

// Driver code

int main()

{

  cout << out::in::val2;   // prints 5

  return 0;

}

**Namespace Aliasing**

n C++, you can use an alias name for your namespace name, for ease of use. Existing namespaces can be aliased with new names, with the following syntax:

namespace new\_name = current\_name;

#include <iostream>

namespace name1

{

    namespace name2

    {

         namespace name3

         {

             int var = 42;

         }

    }

}

// Aliasing

namespace alias = name1::name2::name3;

int main()

{

    std::cout << alias::var << '\n';

}

**Inline namespaces**

An inline namespace is a namespace that uses the optional keyword inline in its original-namespace-definition.

// C++ program to demonstrate working of

// inline namespaces

#include <iostream>

using namespace std;

namespace ns1

{

   inline namespace ns2

   {

       int var = 10;

   }

}

int main()

{

   cout << ns1::var;

   return 0;

}

// C++ program to demonstrate working of

// inline namespaces inside inline namespaces

#include <iostream>

using namespace std;

namespace ns1

{

    inline namespace ns2

    {

        inline namespace ns3

        {

            int var = 10;

        }

    }

}

int main()

{

    cout << ns1::var;

    return 0;

}

**"Using" directive**

This same behavior (same as inline namesapces) can also be achieved by using the "using" declarative inside namespaces. A using-directive that names the inline namespace is implicitly inserted in the enclosing namespace (similar to the implicit using-directive for the unnamed namespace). Consider the following C++ code:

// C++ program to demonstrate working

// of "using" to get the same effect as

// inline.

#include <iostream>

using namespace std;

namespace ns1

{

    namespace ns2

    {

        namespace ns3

        {

            int var = 10;

        }

        using namespace ns3;

    }

    using namespace ns2;

}

int main()

{

    cout << ns1::var;

    return 0;

}

===================================================================

**Section 24** ) **Exception Handling**

**Need of exception handling**

Following are main advantages of exception handling over traditional error handling.

***Separation of Error Handling code from Normal Code:***In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

***Functions/Methods can handle any exceptions they choose:***A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller. In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

***Grouping of Error Types:*** In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

**try:** represents a block of code that can throw an exception.

**catch:** represents a block of code that is executed when a particular exception is thrown.

**throw:** Used to throw an exception. Also used to list the exceptions that a function throws, but doesn’t handle itself.

#include <iostream>

using namespace std;

int main()

{

   int x = -1;

   // Some code

   cout << "Before try \n";

   try {

      cout << "Inside try \n";

      if (x < 0)

      {

         throw x;

         cout << "After throw (Never executed) \n";

      }

   }

   catch (int x ) {

      cout << "Exception Caught \n";

   }

   cout << "After catch (Will be executed) \n";

   return 0;

}

**Generic catch:**

There is a special catch block called ‘catch all’ catch(…) that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so catch(…) block will be executed.

#include <iostream>

using namespace std;

int main()

{

    try  {

       throw 10;

    }

    catch (char \*excp)  {

        cout << "Caught " << excp;

    }

    catch (...)  {

        cout << "Default Exception\n";

    }

    return 0;

}

**Implicit type conversion** doesn’t happen for primitive types. For example, in the following program ‘a’ is not implicitly converted to int

#include <iostream>

using namespace std;

int main()

{

    try  {

       throw 'a';

    }

    catch (int x)  {

        cout << "Caught " << x;

    }

    catch (...)  {

        cout << "Default Exception\n";

    }

    return 0;

}

If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch a char.

#include <iostream>

using namespace std;

int main()

{

    try  {

       throw 'a';

    }

    catch (int x)  {

        cout << "Caught ";

    }

    return 0;

}

**A derived class exception should be caught before a base class exception.**

**Nested try-catch**

In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using “throw; ”

#include <iostream>

using namespace std;

int main()

{

    try {

        try  {

            throw 20;

        }

        catch (int n) {

             cout << "Handle Partially ";

             throw;   //Re-throwing an exception

        }

    }

    catch (int n) {

        cout << "Handle remaining ";

    }

    return 0;

}

A function can also re-throw a function using same “throw; “. A function can handle a part and can ask the caller to handle remaining.

When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

**#include <iostream>**

using namespace std;

class Test {

public:

   Test() { cout << "Constructor of Test " << endl; }

  ~Test() { cout << "Destructor of Test "  << endl; }

};

int main() {

  try {

    Test t1;

    throw 10;

  } catch(int i) {

    cout << "Caught " << i << endl;

  }

}

**Stack Unwinding**

The process of removing function entries from function call stack at run time is called Stack Unwinding. Stack Unwinding is generally related to Exception Handling. In C++, when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack. So exception handling involves Stack Unwinding if exception is not handled in same function (where it is thrown).

#include <iostream>

using namespace std;

// A sample function f1() that throws an int exception

void f1() throw (int) {

  cout<<"\n f1() Start ";

  throw 100;

  cout<<"\n f1() End ";

}

// Another sample function f2() that calls f1()

void f2() throw (int) {

  cout<<"\n f2() Start ";

  f1();

  cout<<"\n f2() End ";

}

// Another sample function f3() that calls f2() and handles exception thrown by f1()

void f3() {

  cout<<"\n f3() Start ";

  try {

    f2();

  }

  catch(int i) {

   cout<<"\n Caught Exception: "<<i;

  }

  cout<<"\n f3() End";

}

// A driver function to demonstrate Stack Unwinding  process

int main() {

  f3();

  getchar();

  return 0;

}

**Catching base and derived classes as exceptions**

If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.

#include<iostream>

using namespace std;

class Base {};

class Derived: public Base {};

int main()

{

   Derived d;

   // some other stuff

   try {

       // Some monitored code

       throw d;

   }

   catch(Base b) {

        cout<<"Caught Base Exception";

   }

   catch(Derived d) {  //This catch block is NEVER executed

        cout<<"Caught Derived Exception";

   }

   getchar();

   return 0;

}

If we put base class first then the derived class catch block will never be reached. For example, following C++ code prints *“Caught Base Exception”*

**Catch block and type conversion**

**#include <iostream>**

using namespace std;

class MyExcept1 {};

class MyExcept2

{

public:

    // Conversion constructor

    MyExcept2 (const MyExcept1 &e )

    {

        cout << "Conversion constructor called";

    }

};

int main()

{

    try

    {

        MyExcept1 myexp1;

        throw myexp1;

    }

    catch(MyExcept2 e2)

    {

        cout << "Caught MyExcept2 " << endl;

    }

    catch(...)

    {

        cout << " Defaule catch block " << endl;

    }

    return 0;

}

**Exception handling and object destruction**

**#include <iostream>**

using namespace std;

class Test {

public:

  Test() { cout << "Constructing an object of Test " << endl; }

  ~Test() { cout << "Destructing an object of Test "  << endl; }

};

int main() {

  try {

    Test t1;

    throw 10;

  } catch(int i) {

    cout << "Caught " << i << endl;

  }

}

**Output:**

Constructing an object of Test

Destructing an object of Test

Caught 10

======================================================================

**Section 25** ) **Templates**

**Defination :**

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

**Function Templates:**

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

**How to declare a function template?**

A function template starts with the keyword **template** followed by template parameter/s inside  **< >** which is followed by function declaration.

**template** <**class** T>

T someFunction(T arg)

{

... .. ...

}

In the above code, *T* is a template argument that accepts different data types (int, float), and **class** is a keyword.

we can also use keyword **typename** instead of class in the above example.

When, an argument of a data type is passed to someFunction( ), compiler generates a new version of someFunction() for the given data type.

**Example 1: Function Template to find the largest number**

// If two characters are passed to function template, character with larger ASCII value is displayed.

#include <iostream>

using namespace std;

// template function

template <class T>

T Large(T n1, T n2)

{

return (n1 > n2) ? n1 : n2;

}

int main()

{

int i1, i2;

float f1, f2;

char c1, c2;

cout << "Enter two integers:\n";

cin >> i1 >> i2;

cout << Large(i1, i2) <<" is larger." << endl;

cout << "\nEnter two floating-point numbers:\n";

cin >> f1 >> f2;

cout << Large(f1, f2) <<" is larger." << endl;

cout << "\nEnter two characters:\n";

cin >> c1 >> c2;

cout << Large(c1, c2) << " has larger ASCII value.";

return 0;

}

Enter two integers:

5

10

10 is larger.

Enter two floating-point numbers:

12.4

10.2

12.4 is larger.

Enter two characters:

z

Z

z has larger ASCII value.

**Class Templates**

Like function templates, we can also create class templates for generic class operations.

Sometimes we need a class implementation that is same for all classes, only the data types used are different.

However, class templates make it easy to reuse the same code for all data types.

**How to declare a class template?**

**template** <**class** T>

class className

{

... .. ...

public:

T var;

T someOperation(T arg);

... .. ...

};

In the above declaration, T is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable *var* and a member function someOperation() are both of type T.

**How to create a class template object?**

To create a class template object, you need to define the data type inside a < > when creation.

className<dataType> classObject;

className<int> classObject;

className<float> classObject;

className<string> classObject;

**Example 3: Simple calculator using Class template**

#include <iostream>

using namespace std;

template <class T>

class Calculator

{

private:

T num1, num2;

public:

Calculator(T n1, T n2)

{

num1 = n1;

num2 = n2;

}

void displayResult()

{

cout << "Numbers are: " << num1 << " and " << num2 << "." << endl;

cout << "Addition is: " << add() << endl;

cout << "Subtraction is: " << subtract() << endl;

cout << "Product is: " << multiply() << endl;

cout << "Division is: " << divide() << endl;

}

T add() { return num1 + num2; }

T subtract() { return num1 - num2; }

T multiply() { return num1 \* num2; }

T divide() { return num1 / num2; }

};

int main()

{

Calculator<int> intCalc(2, 1);

Calculator<float> floatCalc(2.4, 1.2);

cout << "Int results:" << endl;

intCalc.displayResult();

cout << endl << "Float results:" << endl;

floatCalc.displayResult();

return 0;

}

===================================================================

**Section 26**) **C++ Standard Template Library (Containers)**

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms and iterators. It is a generalized library and so, its components are parameterized.

**Refer link :** [**http://www.geeksforgeeks.org/the-c-standard-template-library-stl/**](http://www.geeksforgeeks.org/the-c-standard-template-library-stl/)

**Containers**

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes  and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

**Sequence Containers**:  implement data structures which can be accessed in a sequential manner.

* [vector](http://quiz.geeksforgeeks.org/vector-sequence-containers-the-c-standard-template-library-stl-set-1/)
* [list](http://quiz.geeksforgeeks.org/list-sequence-containers-the-c-standard-template-library-stl/)
* [deque](http://quiz.geeksforgeeks.org/deque-sequence-containers-the-c-standard-template-library-stl/)
* [arrays](http://www.geeksforgeeks.org/array-class-c/)
* [forward\_list](http://www.geeksforgeeks.org/forward-list-c-set-1-introduction-important-functions/)

**Container Adaptors** :  provide a different interface for sequential containers.

* [queue](http://quiz.geeksforgeeks.org/queue-container-adaptors-the-c-standard-template-library-stl/)
* [priority\_queue](http://quiz.geeksforgeeks.org/priority-queue-container-adaptors-the-c-standard-template-library-stl/)
* [stack](http://quiz.geeksforgeeks.org/stack-container-adaptors-the-c-standard-template-library-stl/)

**Associative Containers** :  implement sorted data structures that can be quickly searched (O(log n) complexity).

* [set](http://quiz.geeksforgeeks.org/set-associative-containers-the-c-standard-template-library-stl/)
* [multiset](http://quiz.geeksforgeeks.org/multiset-associative-containers-the-c-standard-template-library-stl/)
* [map](http://quiz.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/)
* [multimap](http://quiz.geeksforgeeks.org/multimap-associative-containers-the-c-standard-template-library-stl/)

**Functions**

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.

**Iterators**

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

Refer link : <http://www.geeksforgeeks.org/iterators-c-stl/>

**Utility Library**

Defined under <utility header>

* [pair](http://quiz.geeksforgeeks.org/pair-simple-containers-the-c-standard-template-library-stl/)

**Std::String class**

Strings are objects that represent sequences of characters.  
  
The standard string class provides support for such objects with an interface similar to that of a [standard container](http://www.cplusplus.com/stl) of bytes, but adding features specifically designed to operate with strings of single-byte characters.

<http://www.cplusplus.com/reference/string/string/>

**================================================================**

**Section 27**)  **FILE Handling**

**Refer link :** [**http://www.csegeek.com/csegeek/view/tutorials/cpp\_lang/cpp\_file.php**](http://www.csegeek.com/csegeek/view/tutorials/cpp_lang/cpp_file.php)

**Files** are a means to store data in a storage device. C++ file handling provides a mechanism to store output of a program in a file and read from a file on the disk.

So far, we have been using **<iostream>** header file which provide functions **cin** and **cout** to take input from console and write output to a console respectively. Now, we introduce one more header file **<fstream>** which provides data types or classes ( **ifstream** , **ofstream** , **fstream** ) to read from a file and write to a file.

**File Opening Modes**

A file can be opened in different modes to perform read and write operations. Function to open a file i.e **open( )** takes two arguments : **char \*filename** and **ios :: mode**. C++ supports the following file open modes :

|  |  |
| --- | --- |
| **Mode** | **Explanation** |
| ios :: in | Open a file for reading |
| ios :: out | Open a file for writing |
| ios :: app | Appends data to the end of the file |
| ios :: ate | File pointer moves to the end of the file but allows to writes data in any location in the file |
| ios :: binary | Binary File |
| ios :: trunc | Deletes the contents of the file before opening |

**-** If a file is opened in **ios :: out** mode, then by default it is opened in **ios :: trunc** mode also i.e the contents of the opened file is overwritten.

- If we open a file using **ifstream** class, then by default it is opened in **ios :: in** mode.

- if we open a file using **ofstream** class, then by default it is opened in **ios :: out** mode.

- The **fstream** class doesn't provide any default mode.

**File Operations using ifstream and ofstream**

**Open a file for writing :**

#include<iostream>

#include<fstream>

using namespace std;

int main() {

ofstream ofile; // declaring an object of class ofstream

ofile.open("file.txt"); // open "file.txt" for writing data

/\* write to a file \*/

ofile << "This is a line in a file" << endl;

ofile << "This is another line" << endl;

/\* write to a console \*/

cout << "Data written to file" << endl;

ofile.close(); // close the file

return 0;

}

**Open a file for reading :**

#include<iostream>

#include<fstream>

using namespace std;

int main() {

char data[100]; // buffer to store a line read from file

ifstream ifile; // declaring an object of class ifstream

ifile.open("file.txt"); // open "file.txt" for reading

cout << "Reading data from a file :-" << endl << endl;

while (!ifile.eof()) { // while the end of file [ eof() ] is not reached

ifile.getline(data, 100); // read a line from file

cout << data << endl; // print the file to console

}

ifile.close(); // close the file

return 0;

}

**Open a file and append data to the end of the file :**

#include<iostream>

#include<fstream>

using namespace std;

int main() {

char line[100];

fstream file; // declare an object of fstream class

file.open("file.txt", ios :: out | ios :: app); // open file in append mode

if (file.fail()) { // check if file is opened successfully

// file opening failed

cout << "Error Opening file ... " << endl;

}

else {

// proceed with further operations

cout << "Enter a line : ";

cin.getline(line, 100);

file << line << endl; // Append the line to the file

cout << "Line written into the file" << endl;

}

return 0;

}

**Open the file for reading**

#include<iostream>

#include<fstream>

using namespace std;

int main() {

char line[100];

fstream file; // declare an object of fstream class

file.open("file.txt", ios :: out | ios :: app); // open file in append mode

if (file.fail()) { // check if file is opened successfully

// file opening failed

cout << "Error Opening file ... " << endl;

}

else {

// proceed with further operations

cout << "Enter a line : ";

cin.getline(line, 100);

file << line << endl; // Append the line to the file

cout << "Line written into the file" << endl;

}

return 0;

}

**Writing class objects to a file**

There are member functions read( ) and write( ) in the fstream class which allows reading and writing of class objects. These functions can also be used to write array elements into the file.

#include<iostream>

#include<fstream>

using namespace std;

// define a class to store student data

class student {

int roll;

char name[30];

float marks;

public:

student() { }

void getData(); // get student data from user

void displayData(); // display data

};

void student :: getData() {

cout << "\nEnter Roll No. : ";

cin >> roll;

cin.ignore(); // ignore the newline char inserted when you press enter

cout << "Enter Name : ";

cin.getline(name, 30);

cout << "Enter Marks : ";

cin >> marks;

}

void student :: displayData() {

cout << "\nRoll No. : " << roll << endl;

cout << "Name : " << name << endl;

cout << "Marks : " << marks << endl;

}

int main() {

student s[3]; // array of 3 student objects

fstream file;

int i;

file.open("objects.txt", ios :: out); // open file for writing

cout << "\nWriting Student information to the file :- " << endl;

for (i = 0; i < 3; i++) {

s[i].getData();

// write the object to a file

file.write((char \*)&s[i], sizeof(s[i]));

}

file.close(); // close the file

file.open("objects.txt", ios :: in); // open file for reading

cout << "\nReading Student information to the file :- " << endl;

for (i = 0; i < 3; i++) {

// read an object from a file

file.read((char \*)&s[i], sizeof(s[i]));

s[i].displayData();

}

file.close(); // close the file

return 0;

}

**Manipulation of file pointers**

**T**he read operation from a file involves **get** pointer. It points to a specific location in the file and reading starts from that location. Then, the **get** pointer keeps moving forward which lets us read the entire file.

- Similarly, we can start writing to a location where **put** pointer is currently pointing. The **get** and **put** are known as file position pointers and these pointers can be manipulated or repositioned to allow random access of the file.

The functions which manipulate file pointers are as follows :

|  |  |
| --- | --- |
| **Function** | **Description** |
| seekg( ) | Moves the **get** pointer to a specific location in the file |
| seekp( ) | Moves the **put** pointer to a specific location in the file |
| tellg( ) | Returns the position of **get** pointer |
| tellp( ) | Returns the position of **put** pointer |

The function **seekg( n, ref\_pos )** takes two arguments : **n** denotes the number of bytes to move and **ref\_pos** denotes the reference position relative to which the pointer moves. **ref\_pos** can take one of the three constants : **ios :: beg** moves the **get** pointer **n** bytes from the beginning of the file, **ios :: end** moves the **get** pointer **n** bytes from the end of the file and **ios :: cur** moves the **get** pointer **n** bytes from the current position. If we don't specify the second argument, then **ios :: beg** is the default reference position.  
The behaviour of **seekp( n, ref\_pos )** is same as that of **seekg( )**. Following program illustrates random access of file using file pointer manipulation functions :

#include<iostream>

#include<fstream>

using namespace std;

int main() {

fstream fp;

char buf[100];

int pos;

// open a file in write mode with 'ate' flag

fp.open("random.txt", ios :: out | ios :: ate);

cout << "\nWriting to a file ... " << endl;

fp << "This is a line" << endl; // write a line to a file

fp << "This is a another line" << endl; // write another file

pos = fp.tellp();

cout << "Current position of put pointer : " << pos << endl;

// move the pointer 10 bytes backward from current position

fp.seekp(-10, ios :: cur);

fp << endl << "Writing at a random location ";

// move the pointer 7 bytes forward from beginning of the file

fp.seekp(7, ios :: beg);

fp << " Hello World ";

fp.close(); // file write complete

cout << "Writing Complete ... " << endl;

// open a file in read mode with 'ate' flag

fp.open("random.txt", ios :: in | ios :: ate);

cout << "\nReading from the file ... " << endl;

fp.seekg(0); // move the get pointer to the beginning of the file

// read all contents till the end of file

while (!fp.eof()) {

fp.getline(buf, 100);

cout << buf << endl;

}

pos = fp.tellg();

cout << "\nCurrent Position of get pointer : " << pos << endl;

return 0;

}

=================================================================

**Section 28**)  **Design Pattern**

[**http://www.geeksforgeeks.org/singleton-design-pattern/**](http://www.geeksforgeeks.org/singleton-design-pattern/)

**Singleton pattern**

The singleton pattern is one of the simplest design patterns. Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

**Definition:**

*The singleton pattern is a design pattern that restricts the instantiation of a class to one object.*

**Refer link :** [**https://www.codeproject.com/Articles/1921/Singleton-Pattern-its-implementation-with-C**](https://www.codeproject.com/Articles/1921/Singleton-Pattern-its-implementation-with-C)

# **Factory Pattern in C++**

Basically a Factory consists of an interface class which is common to all of the implementation classes that the factory will create. Then you have the factory class which is usually a singleton class that spawns instances of these implementation classes.

**Refer link :** [**https://www.codeproject.com/Articles/363338/Factory-Pattern-in-Cplusplus**](https://www.codeproject.com/Articles/363338/Factory-Pattern-in-Cplusplus)