**Programming Paradigm and OOP Concept:**

1. **What is Programming Paradigm?**

Answer) Programming paradigm is a fundamental style of computer programming. It is a way of building the structures and functions around/of the program.

1. **Mention some points about Object Oriented Programming Paradigm:**

Answer)

* OOP treats data as a critical element and **does not allow data to flow freely around the system.**
* OOP ties data close to the function which operates on it
* OOP allows decomposition of a function into a set of entities called objects(runtime instance of class) and builds data around them
* The data of an object can be accessed only by the function associated with the objects.
* One object can communicate with other objects using function of that object.

1. **Describe four key concepts of object oriented programming.**

Answer) The four key concepts of object oriented programming is 1) Encapsulation 2) Abstraction 3) Inheritance 4) Polymorphism

**1) Encapsulation:** The wrapping up of data and methods into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world. Only those methods, which are wrapped in the class, can access it.

So, these methods (which are wrapped in the class) provides the object's data and the program. This insulation of the data from direct access by the program is called 'data hiding'. (Private data members are example of data encapsulation)

**(data hiding, not implementation hiding)**

**2) Abstraction:** Abstraction refers to the act of representing essential features of a class and omit the unnecessary details about it. For instance, when we think about a car, we don't consider all the irrelevant details like how the break actually works or how the Bluetooth device is installed in the

car, how it provides all the cool features. We think it as a transport medium which could take us

from one place to another place and has some cool features. That is abstraction.

(abstraction is just that. It does not hide data. Instead, it hides unnecessary implementation details. Like, In a car, you can steerWheel() or stopCar(). But, you need not to know the implementation of how a car stops.

**Now, the difference between abstraction and encapsulation:**

One is mechanism hiding and another is data hiding.

**3) Inheritance:** Inheritance is a way by which a newly defined class inherits attributes and

behaviour of an existing class along with its own properties.

Using inheritance the hierarchical relationships are established.

Inheritance allows the re usability of an existing operations and extending the basic unit of the a class without creating it from the scratch.

Some more points about Inheritance:

Inheritance is a “is a” relationship (not a “has a” relationship)

Like: we can say “A four-wheeler is a car” (It's a real “is a” relationship)

But, we cannot say, “A steering wheel is a car” (It's a “has a” relationship)

(A has a relationship is used in composition. An example of composition: A structure with in a structure as a member)

**4) Polymorphism:** Polymorphism is sharing a common interface for multiple types but having different implementation for different types.

In OOP, polymorphism is a technique where objects of classes belonging to

the same hierarchical tree may posses interface bearing the same name but each having different behaviors.

It is the way of inheriting when useful, overriding when not useful.

It allows automatically do the current behaviour even if we are working with

many different forms

**4.Define class.**

Answer) Class is a static definition of new type as a collection of data and associated operations from which runtime instances called objects can be created.

**5.Define object.**

Answer) Object is runtime instance of a conceptual framework encapsulating typed data and typed operations that correspond to a real world entity or thing for the purpose of computer modeling.

**Memory Layout:**

**6.Memory Layout/Different Memory Segment In C++:**

A typical memory representation of C program consists of following sections.

(this is allocation of RAM from the c++ language perspective)

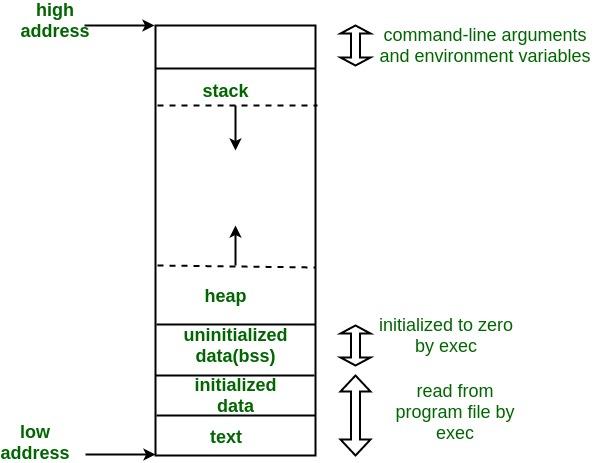
1. Text segment/code segment

2. Initialized data segment

3. Uninitialized data segment

4. Stack

5. Heap



**Code segment:**

* Code segment is fixed
* Code segment is read only.

When an executable file is bought into main memory (by long term scheduler and short term scheduler. Long term scheduler does the mix and put into the ready queue and short term chooses a process from the ready queue and bought it into main memory for execution) it is loaded into this part. The processor will access the machine instructions to perform specified actions.

**Initialized Data Segment:**

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer. **(not the system to the default value)**

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into **initialized read-only area and initialized read-write area.**

**For instance the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char\* string = “hello world” makes the string literal “hello world” to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.**

Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

**Uninitialized Data Segment:**

Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared static int i; would be contained in the BSS segment.

For instance a global variable declared int j; would be contained in the BSS segment.

**Heap:**

Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

**Stack:**

The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)

**(So, when std::bad\_alloc is generated, it’s not that heap memory is exhausted, it’s that the combination of stack memory and heap memory got exhausted)**

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

Now, some crucial point about memory location:

A static variable’s memory is allocated from data segment.

**Concepts Like namespace and inline functions:**

**7.What is namespace?**

Answer) Consider a situation, when we have two persons with the same name, Zara, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area if they live in different area or their mother or father name, etc.

Same situation can arise in your C++ applications. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A namespace is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

**Now, some details about namespace:**

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.  
    
  **(what does it necessarily mean?**

**You cannot declare it withing a function or class)**

* 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.

In the below, we provide an example which shows that multiple namespaces (since, at the end namespace is a scope/ declarative region for identifiers) can be defined in a program, and both contain the same function. But, since, their scope will be difference, it wont make any problem.

**// 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;**

**}**

**Note** How the namespace ns1 and ns2’s value function is called.

**ns1::value()**

There is another example provided here, to explain how to declare a class inside a namespace and define it, and how to use it:

**// 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. Check the below example.

**// 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;**

**}**

Classes can defined within the namespace, whereas, functions of a class can be defined outside the namespace

**// 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;**

**}**

1. **What is inline function?**

Answer) An inline function is a function that is expanded in line when it is invoked (called). That is, the the compiler replaces the function call with the corresponding function code (something similar to macro expansion)

* Inline functions are reduced to overhead of normal functional call. (Every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the functions (the called function), saving registers (for current register, because, suppose, in a paging environment, the pages of that particular function is not there in the main memory. (main memory is RAM, p**rimary** storage, also known as **main** storage or **memory**, is the area in a computer in which data is stored for quick access by the computer's processor. The terms random access **memory** (**RAM**) and **memory** are often as synonyms for **primary** or **main** storage. )pushing arguments into the stacks and returning

to the calling function. Now, all of these, (i.e. jumping to function, saving

resisters, pushing arguments into the stack and returning to the calling function)

causes a huge time overhead. **Because, there will be at least two context**

**switch in terms of operating system (that is what I understand)**.

When a function is small and calls repetitively a substantial percentage of

execution time may be spent in such overloads.

It's not a command. (i.e. Inline is not a command) It's rather a request to the c++ compiler that the substitution of the function body is to be preferred to the usual function call implementation.

Suggestion/Request of expanding a function inline could be ignored based on the compiler's discretion.

**Advantages :-**

1) It does not require function calling overhead.

2) It also save overhead of variables push/pop on the stack, while function calling.

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

4) It increases locality of reference by utilizing instruction cache. **(one concept could be that the function which is called frequently is stored on TLB (translation look aside buffer) a form of cache. However, if function is not big enough, the paging performance cannot be improved. Since, there will be lots of cache miss.)**

(Translation Look Aside Buffer.

Now, this optimizes the performance.

TLB hit, TLB miss

TLB is a fast cache.

In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random. Furthermore, some TLBs allow certain entries to be wired down, meaning that they cannot be removed from TLB. Typically TLB entries for kernel codes are wired down.

Some TLBs store ASIDs (Address Space Identifier) in each TLB entry. An ASID uniquely identifies each process, and is used to provide address space protection for this process. (I can understand that ASID can uniquely identifies a process, but how it is used for address space protection for each process). When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously.

If the TLB does not support separate ASIDs, then every time a new page table is selected, (for instance, with each context switch) The TLB must be flushed (or erased) to ensure that the next executing process does not use wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.**)**

(Now, you can clearly understand the introduction of TLB concept here)

(But, is this same as instruction cache?

(an **instruction cache** is a cache to speed up executable instruction fetch)

1. After in-lining compiler can also apply intraprocedural optimization if specified. This is the most important one, in this way compiler can now focus on dead code elimination, can give more stress on branch prediction, induction variable elimination etc
2. inline is mandatory if a function (no matter how complex or "linear") is defined in a header file, to allow multiple sources to include it without getting a "multiple definition" error by the linker.

**Disadvantages:**

1) May increase function size so that it may not fit on the cache, causing lots of cache miss.

2) After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization.

3) It may cause compilation overhead as if some body changes code inside inline function than all calling location will also be compiled. **(I kind of know what is static linking and what is dynamic linking. In dynamic linking, the function call statement is replaced with a pointer to the memory location in which the actual function definition can be found, Now, in case of static linking, the calls are directly replaced with function defintion. So, if the actual function definition changes, then compilation will take more time.)**

4) If used in header file, it will make your header file size large and may also make it unreadable.

5) If somebody used too many inline function resultant in a larger code size than it may cause thrashing in memory. More and more number of page fault bringing down your program performance.

**(Thrashing:**

If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process's execution. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.

In fact, look at any process that does not have "enough" frames. If the

process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must back in immediately.

This high paging activity is called A process is thrashing if it is spending more time paging than executing.

)

(Now, how thrashing occurs here?

Because, if the executable program needs more pages but it not allocated such a huge number of pages.)

1. Its not useful for embedded system where large binary size is not preferred at all due to memory size constraints.

**9.The Cases When inline request is generally ignored:**Some of the situations where inline expansion may not work:

1. Suppose an inline function is returning values, If a loop, a switch or a goto exists
2. For functions not returning values, if a return statement exists  
     
   void func\_name(data\_type1 arg1, data\_type2 arg2)  
   {  
    return;

}

1. If function contain static variables
2. If inline functions are recursive

**10.Example of one particular case when inline function is mandatory?**

Answer) inline is mandatory if a function (no matter how complex or "linear") is defined in a

header file, to allow multiple sources to include it without getting a **"multiple definition" error by the linker.**

**11.Difference Between Inline function And Macro Substitution:**

Inline function requests compiler to make a function expand inline. Whereas, macro substitution, substitution is guaranteed. It’s not a request. It will happen

Inline function is a compilation step. Macro substitution is a preprocessing step done by preprocessor before compilation.

There is always some side effects of just textual substitution:

**1.#include <stdio.h>**

**2.int foo(int, int);**

**3.#define foo(x, y) x / y + x**

**4.int main()**

**5.{**

1. **int i = -6, j = 3;**
2. **printf("%d ",foo(i + j, 3));**
3. **#undef foo**
4. **printf("%d\n",foo(i + j, 3));**

**10.}**

**11.int foo(int x, int y)**

**12.{**

1. **return x / y + x;**
2. **}**

This will generate -8 when foo acts as macro substitution. Generates -4 When function is called.

**Pointers And References:**

1. **What is importance of pointer in c++?**

Answer) Pointer concept is useful in many contexts. Some of them are following:

1) Pointer helps us to allocate memory dynamically. Static memory allocation often leads to wastage of memory space. So, dynamically memory allocation is important

2) Pointers allow us to resize the data structure whenever needed. For example, if you have an array of size 10, it cannot be resized. But, an array created out of malloc and assigned to a pointer can be resized easily by using realloc.

3) There is a concept of function pointer. A function pointer can point to a function. Now, function pointers are also useful in many contexts. Like,

• Function pointers can be useful when you want to create callback mechanism, and need to pass address of an function to another function. Like, when we use qsort function of stdlib.h header file we send a pointer to a function as the fourth argument of the function. The callback mechanism provided by function pointer is also used for event handlers, parser specialization, comparator function passing. In fact, C-language GUI toolkits and application frameworks (e.g. Gnome/Gtk+/Glib) often accept function pointers as “callbacks” for timer or user interface events. (EG: “call this function whenever this button is clicked” or “…whenever this timer expires”)

• **Plugins and extensions -** the pointers to functions provided by plugins or library extensions are gathered by a standard function GetProcAddress, dlsym or similar, which take the function identifier as name and return a function pointer. Absolutely vital for APIs like OpenGL.

• **Finite State Machines:** Where the elements of (multi-dimensional) arrays indicate the routine that processes/handles the next state. This keeps the definition of the FSM in one place (the array).

• **Enabling features:** and disabling of features can be done using function pointers. You may have features that you wish to enable or disable that do similar yet distinct things. Instead of populating and cluttering your code with if-else constructs testing variables, you can code it so that it uses a function pointer, and then you can enable/disable features by changing/assigning the function pointer. If you add new variants, you don't have to track down all your if-else or switch cases (and risk missing one); instead you just update your function pointer to enable the new feature, or disable the old one.

In c++, base class pointers are type compatible with derived class pointers. This is also another usefulness of pointer concept. Now, suppose, you create a class named Bank account for a bank.

Now, suppose the bank can create atmost 10000 different bank accounts in a moth. Now, different bank users have different demand in mind when it comes about bank account. Like: One user may want to create a recurring bank account. Another user may want to create a savings bank account.

Now, there's a requirement of dynamically allocating bank account to the users according to their demand among those 10000 bank accounts (the upper limit of bank accounts the bank can create in a month). In these types of cases, the compatibility of base class pointers to derived class pointers can be very useful.

**13.What is callback Function In C/C++?**A callback is any executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at a given time [Source : Wiki]. In simple language, If a reference of a function is passed to another function as an argument to call it, then it it will be called as a Callback function.

In C, a callback function is a function that is called through a function pointer.

Below is a simple example in C to illustrate the above definition to make it more clear:

**// A simple C program to demonstrate callback**

**#include<stdio.h>**

**void A()**

**{**

**printf("I am function A\n");**

**}**

**// callback function**

**void B(void (\*ptr)())**

**{**

**(\*ptr) (); // callback to A**

**}**

**int main()**

**{**

**void (\*ptr)() = &A;**

**// calling function B and passing**

**// address of the function A as argument**

**B(ptr);**

**return 0;**

**}**

**It is used on so many cases in functions like qsort and sort.**

**14.Why pass by reference concept is added in c++?**

Answer) Pass by reference concept in added in c++, because of the following reasons:

• they are necessary to define copy constructors

• they are necessary for operator overloads

• const references allow you to have pass-by-value semantics while avoiding a copy

**15.Some notes about reference:**

**Constant reference:**

void foo(const int &x) // x is a const reference

{

x = 6; // compile error: a const reference cannot have its value changed!

}

**References to pointers:**

It’s possible to pass a pointer by reference, and have the function change the address of the pointer entirely:

**#include <iostream>**

**void foo(int \*&ptr) // pass pointer by reference**

**{**

**ptr = nullptr; // this changes the actual ptr argument passed in, not a copy**

**}**

**int main()**

**{**

**int x = 5;**

**int \*ptr = &x;**

**std::cout << "ptr is: " << (ptr ? "non-null" : "null") << '\n'; // prints non-null**

**foo(ptr);**

**std::cout << "ptr is: " << (ptr ? "non-null" : "null") << '\n'; // prints null**

**return 0;**

**}**

**16.Pros and Cons About Passing By Reference:**

**Advantages of passing by reference:**

References allow a function to change the value of the argument, which is sometimes useful. Otherwise, const references can be used to guarantee the function won’t change the argument.

**Because a copy of the argument is not made, pass by reference is fast, even when used with large structs or classes.**

References can be used to return multiple values from a function (via out parameters).

References must be initialized, so there’s no worry about null values.

**Disadvantages of passing by reference:**

Because a non-const reference cannot be initialized with an const l-value or an r-value (e.g. a literal or an expression), arguments to reference parameters must be normal variables.  
  
**What is the meaning of this:**

**#include<cstdio>**

**using namespace std;**

**int main()**

**{**

**int &x=6;**

**//x is a non const reference variable**

**//6 is an integral constant**

**return 0;**

**}**

This will generate a compilation error.

Because, a non const reference cannot be initialized with an const l value (a literal: like: 6 is a literal, an integral constant)

The compilation error will say:

**invalid initialization of non-const reference of type ‘int&’ from a temporary of type ‘int’**

Similarly:

**#include<cstdio>**

**int main()**

**{**

**int x=6,y=7;**

**int &ref=x\*y;**

**//x\*y is an expression: r value**

**//ref is a non const reference**

**return 0;**

**}**

This will generate compilation error , too.

It can be hard to tell whether a parameter passed by non-const reference is meant to be input, output, or both. Judicious use of const and a naming suffix for out variables can help.

It’s impossible to tell from the function call whether the argument may change. An argument passed by value and passed by reference looks the same. We can only tell whether an argument is passed by value or reference by looking at the function declaration. This can lead to situations where the programmer does not realize a function will change the value of the argument.

**17.Difference Between Pointer And Reference**

**A pointer is the memory address of an object. A reference is an alias for an object.** References are often implemented using pointers under the covers, and the distinction can be tricky. However I don't recommend thinking of references as "safe pointers".

Some ways to consider the difference (using C++ syntax):

A reference is treated \*exactly\* as if you had used the original variable in its place. For example, if you assign to a reference, it is as if you assigned to the original variable:

int x = 5;

int &y = x; // y is an alias for x

y = 6; // now x == 6

When you pass a parameter by reference, the parameter inside the function is an alias to the variable you passed from the outside. When you pass a variable by a pointer, you take the address of the variable and pass the address into the function. The main difference is that you can pass values without an address (like a number) into a function which takes a const reference, while you can't pass address-less values into a function which takes const pointers.

**(this is from stackoverflow)**

References are generally implemented using pointers. A reference is same object, just with a different name and reference must refer to an object. Since references can’t be NULL, they are safer to use.

* A pointer can be re-assigned while reference cannot, and must be assigned at initialization only.
* Pointer can be assigned NULL directly, whereas reference cannot.
* Pointers can iterate over an array, we can use ++ to go to the next item that a pointer is pointing to.
* A pointer is a variable that holds a memory address. A reference has the same memory address as the item it references.
* A pointer to a class/struct uses ‘->'(arrow operator) to access it’s members whereas a reference uses a ‘.'(dot operator)
* A pointer needs to be dereferenced with \* to access the memory location it points to, whereas a reference can be used directly.

Now, what do you mean by the fact that a pointer can be re assigned while reference cannot, and reference must be assigned at initialization only.

**#include<cstdio>**

**#include<iostream>**

**using namespace std;**

**int main()**

**{**

**int &x;**

**&x=6;**

**cout<<"x is: "<<x<<endl;**

**return 0;**

**}**

You cannot do that.

It will generate compilation error. Compiler will say:

**‘x’ declared as reference but not initialized**

**lvalue required as left operand of assignment**

Now, the second error could be due to the fact that a non const reference cannot be assigned with a const l value.

**#include<cstdio>**

**#include<iostream>**

**using namespace std;**

**int main()**

**{**

**int y=6;**

**int z=7;**

**int &x=y;**

**&x=z;**

**//you cannot reassign it**

**x=z;**

**//now, this is different. This will change y's value to 7**

**return 0;**

**}**

You cannot do that, either.

**Type Qualifiers in c++**

**18.What is “const” keyword in c++? What is the use of it?**

**(short description)**

It can be used with variables.

It can be used with function.

It can be used with a reference.

**19. What is “volatile” keyword in c++? What is the use of it?**1) Global variables modified by an interrupt service routine outside the scope: For example, a global variable can represent a data port (usually global pointer referred as memory mapped IO) which will be updated dynamically. The code reading data port must be declared as volatile in order to fetch latest data available at the port. Failing to declare variable as volatile, the compiler will optimize the code in such a way that it will read the port only once and keeps using the same value in a temporary register to speed up the program (speed optimization). In general, an ISR (Interrupt Service Routine) used to update these data port when there is an interrupt due to availability of new data

1. Global variables within a multi-threaded application: There are multiple ways for threads communication, viz, message passing, shared memory, mail boxes, etc. A global variable is weak form of shared memory. When two threads sharing information via global variable, they need to be qualified with volatile. Since threads run asynchronously, any update of global variable due to one thread should be fetched freshly by another consumer thread. Compiler can read the global variable and can place them in temporary variable of current thread context. To nullify the effect of compiler optimizations, such global variables to be qualified as volatile.  
     
   (Because, compiler optimization will make a variable cached thread locally. This could be a huge problem in multicore CPU. Because, a thread might not get the updated value of the shared variable If it is not volatile)

**If we do not use volatile qualifier, the following problems may arise**

1) Code may not work as expected when optimization is turned on.

2) Code may not work as expected when interrupts are enabled and used.  
  
  
**20.const volatile Qualifier:**#include <iostream>

#include <typeinfo>

using namespace std;

int main(void)

{

int a1 = 40;

const volatile int\* b1 = &a1;

cout << "typeid of b1 " << typeid(b1).name() << '\n';

int\* c1 = const\_cast <int \*> (b1);

cout << "typeid of c1 " << typeid(c1).name() << '\n';

return 0;

}

Now, it will not generate compilation error.

So, const and volatile can be mixed.

This is possible and mostly used in embedded system.The example is Interrupt Status Register as it is a status register , in the program we should not modify this Variable so it should be a constant.But this variable can be changed by the processor or hardware based on the interrupt condition. So when in the program ,we want to read the value of this variable , it should read the actual value with out any optimization.For this reason ,the variable can be declared as volatile too.

(so, current program cannot modify it. Hence, const. But, interrupt routine can change it. (imagine the scenario in a multicore CPU. So, when in the program, we want to read the value of this variable, it should read the actual value of this variable. Hence, volatile too)

Const qualifier makes sure the variable declared isn’t changed by the code, say assignment or something similar. This doesn’t restrict the variable from being modified external to the code. Take a peripheral’s status register as an example. If we don’t want the user to modify the contents of the register (Read-only mode), it is made const. Now if there is a chance for this register to be updated outside the code, the variable shouldn’t be cached. Make it volatile.

**Storage Class Specifiers In C++**

There are five of them in c++.

* Auto
* Register
* Static
* Extern
* Mutable

Now, you can find some decent amount of mcqs solved about the first four storage class specifiers in my repository C-learning-and-MCQ. As these four storage class specifiers are also in c.

So, here we will only discuss about mutable storage class specifier.

**21.What is mutable storage class specifier in c++?**

Sometimes there is requirement to modify one or more data members of class / struct through const function even though you don’t want the function to update other members of class / struct. This task can be easily performed by using mutable keyword. Consider this example where use of mutable can be useful. Suppose you go to hotel and you give the order to waiter to bring some food dish. After giving order, you suddenly decide to change the order of food. Assume that hotel provides facility to change the ordered food and again take the order of new food within 10 minutes after giving the 1st order. After 10 minutes order can’t be cancelled and old order can’t be replaced by new order. See the following code for details.

#include <iostream>

#include <string.h>

using std::cout;

using std::endl;

class Customer

{

char name[25];

mutable char placedorder[50];

int tableno;

mutable int bill;

public:

Customer(char\* s, char\* m, int a, int p)

{

strcpy(name, s);

strcpy(placedorder, m);

tableno = a;

bill = p;

}

void changePlacedOrder(char\* p) const

{

strcpy(placedorder, p);

}

void changeBill(int s) const

{

bill = s;

}

void display() const

{

cout << "Customer name is: " << name << endl;

cout << "Food ordered by customer is: " << placedorder << endl;

cout << "table no is: " << tableno << endl;

cout << "Total payable amount: " << bill << endl;

}

};

int main()

{

const Customer c1("Pravasi Meet", "Ice Cream", 3, 100);

c1.display();

c1.changePlacedOrder("GulabJammuns");

c1.changeBill(150);

c1.display();

return 0;

}

**Closely observe the output of above program. The values of placedorder and bill data members are changed from const function because they are declared as mutable.**

(example is taken from geeksforgeeks.  
<https://www.geeksforgeeks.org/c-mutable-keyword/>

)

Now, this is not the best of examples.

**The two cases where generally mutable is used:**

synchronization objects (mutexes, semaphores, atomic spinlocks, etc) -- this is the most common case in my practice, I am very used to writing mutable std::mutex whatever;. If a const-qualified member function has to acquire a mutex, the state of the mutex is changed, so it must be mutable.

memoiziation caches, for when something is referentially-transparent, but expensive to calculate, the first call to the (const-qualified) accessor calculates the value and stores it in a mutable member hash table, second and subsequent calls fetch the value from the table instead.

**Default Arguments:**

**22.What is default argument in c++? What is the use of it?**

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call. **Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default**

**values.**

**A default argument is checked for type at the time of the declaration and**

**evaluated at the time of call.**

**(So, it is a compile time phenomenon first. When compiling, compiler first checks whether the number of arguments passed to a function call is the number of parameters a function accept. If both are same, no concept of default argument will be applied. If both are not same, it will check if by using default arguments we can match the function call with the function declaration. During this check, it will also follow the rules of default argument. From right to left.**

**Note:** Only trailing arguments can have default values and therefore we must add defaults from right to left. We cannot provide a default value to a particular value to a particular argument in the middle of an argument list.

**Basics Of Derived Datatypes:**

**23.What is the difference of class and structure in c++ context?**

Answer) The conventional structure in c cannot contain functions (Functions which operate on data items contained by the function). But that is not the problem with c++ structures. In case of c++, structures can hold both data members and the functions operating on them. But, in c++, the members of a structure are public (by default). So, it violates one of the most important OOP characteristics-Encapsulation. Whereas, the members of a class are by default private.

Point wise difference:

1) c++ structs have public members by default and classes have private members by default.

2) Bases of a struct are inherited publicly by default, whereas bases of a class are inherited privately by default.

**Below is the example of a struct having member functions:**

using std::cout;

using std::cin;

using std::endl;

struct person\_str

{

string name;

int age;

//constructor

person\_str()

{

name = "default";

age = 77;

}

void print();

};

void person\_str::print()

{

/\*

we don't have to mention what "name" and "age" are,

because it automatically refers back to the

member variables.

The "this" variable is the implied first argument,

which is a pointer to the object that the member it was called on.

\*/

cout << name << " " << this->age << endl;

}

int main()

{

person\_str p1;

p1.name = "Chrys";

p1.age = 26;

p1.print();

}

**Static Member Function:**

**24.Mention Some Properties Of Static Members Of A Class:**

The common properties of static member of a class are given below:

* It is initialised to zero(if no value is given i.e. Uninitialized by the programmer) or to the given value when the first object of its class is created. No other initialization is permitted. (Note: even for normal static variables, it is true. In case of c, same static variable can be declared many times, but we can initialize it only once. Whereas, in case of c++, we can declare a static variable only once and we can also initialize it only once). Also, it is not mandatory that the initialization statement has to be present before the creation of first object. It can be present at anywhere. Compiler will find it. (if there is any: Otherwise compilation error will happen). But, the initialization statement can be present at anywhere that does not mean it could be present in some function's scope. The initialization function could be present anywhere in the global scope.
* Only one copy of that function is created for the entire class and is shared by all the objects of that class, no matter how many objects are created. **(hence, it is class variable or class member, instead of being instance variable or instance members)**
* It is visible within the class, but **its lifetime is the entire program.**
* A static function member operate on non static members of a class.
* For static member functions, this pointer does not exist.
* It can only call other static members of the class.
* For static variables of a class, the memory is allocated from data segment.

**First Example:**

This example describes how static variable of a class can be initialized:

**#include<iostream>**

**using namespace std;**

**class item**

**{**

**private:**

**static int count;**

**int number;**

**public:**

**void getdata(int);**

**void getcount(void);**

**};**

**void item::getdata(int a)**

**{**

**number=a;**

**count++;**

**}**

**void item::getcount(void)**

**{**

**cout<<"Count:";**

**cout<<count<<"\n";**

**}**

**int item::count;**

**int main()**

**{**

**item a,b,c;**

**a.getcount();**

**b.getcount();**

**c.getcount();**

**a.getdata(100);**

**b.getdata(200);**

**c.getdata(300);**

**cout<<"After reading data"<<"\n";**

**a.getcount();**

**b.getcount();**

**c.getcount();**

**return 0;**

**}**

Now, check the declaration of the static member.

**int item::count;**

Datatype class name :: variable name;

Now, the static member is not initialized. Now, depending on the initialization, it’s memory is allocated from initialized data segment or uninitialized data segment.

Now, here, it is not initialized. Hence, it’s memory will be allocated from uninitialized data segment. Now, all members of uninitialized data segment or bsss segment is initialized with 0.

**Proof that we can write the initialization statement anywhere where the scope is global:**

**Second Example:**

**#include<iostream>**

**using namespace std;**

**class item**

**{**

**private:**

**static int count;**

**int number;**

**public:**

**void getdata(int);**

**void getcount(void);**

**};**

**void item::getdata(int a)**

**{**

**number=a;**

**count++;**

**}**

**void item::getcount(void)**

**{**

**cout<<"Count:";**

**cout<<count<<"\n";**

**}**

**int main()**

**{**

**item a,b,c;**

**a.getcount();**

**b.getcount();**

**c.getcount();**

**a.getdata(100);**

**b.getdata(200);**

**c.getdata(300);**

**cout<<"After reading data"<<"\n";**

**a.getcount();**

**b.getcount();**

**c.getcount();**

**return 0;**

**}**

**int item::count;**

Now, you can see, it can be actually declared at any portion of class scope.

**Third Example:**

**#include<iostream>**

**using namespace std;**

**class test**

**{**

**private:**

**int code;**

**static int count;**

**public:**

**void setcode (void);**

**void showcode(void);**

**static void showcount(void);**

**};**

**void test::setcode(void)**

**{**

**code=++count;**

**}**

**void test::showcode(void)**

**{**

**cout<<"Object number: "<<code<<"\n";**

**}**

**void test::showcount(void)**

**{**

**cout<<"Count:"<<count<<"\n";**

**}**

**int test::count;**

**//initialization of static variable**

**int main()**

**{**

**test t1,t2;**

**t1.setcode();**

**t2.setcode();**

**test::showcount();**

**test t3;**

**t3.setcode();**

**test::showcount();**

**t1.showcode();**

**t2.showcode();**

**t3.showcode();**

**return 0;**

**}**

Notice, how the static function is called:  
  
**test::showcount();**

**Using the class name and scope operator.**

**Note:** During definition of the static function showcount outside the class declaration, we cannot define it like:

**static void test::showcount(void)**

**{**

**cout<<"Count:"<<count<<"\n";**

**}**

That will result in compilation error. The error will say:

cannot declare member function ‘static void test::showcount()’ to have static linkage [-fpermissive]

**static void test::showcount(void)**

We have to define it like:

(by omitting static keyword)

**void test::showcount(void)**

**{**

**cout<<"Count:"<<count<<"\n";**

**}**

**Fourth Example:**

You can not access non static members of a class from a static function of a class.

#include<iostream>

using namespace std;

class test

{

private:

int code;

static int count;

public:

void setcode (void);

void showcode(void);

static void showcount(void);

};

void test::setcode(void)

{

code=++count;

}

void test::showcode(void)

{

cout<<"Object number: "<<code<<"\n";

}

void test::showcount(void)

{

cout<<"Count:"<<count<<"\n";

cout<<"Code :"<<code<<"\n";

}

int test::count;

//initialization of static variable

int main()

{

test t1,t2;

t1.setcode();

t2.setcode();

test::showcount();

test t3;

t3.setcode();

test::showcount();

t1.showcode();

t2.showcode();

t3.showcode();

return 0;

}

**This will generate compilation error.**

**cpptrial10.cpp: In static member function ‘static void test::showcount()’:**

**cpptrial10.cpp:6: error: invalid use of member ‘test::code’ in static member function**

**cpptrial10.cpp:24: error: from this location**

**(read the first answer: <https://www.quora.com/Why-cant-a-static-method-directly-access-non-static-members-What-is-the-logic-behind-it-Is-it-because-they-will-have-different-values-for-each-instance)>**

**25.Can a static member of a class be overridden in the derived class?**No. Static methods can not be overridden. But can be overloaded since they are resolved using static binding by compiler at compile time.

**Friend Function And Class:**

**26.Mention some properties of friend function in c++.**A friend function possesses certain special characteristics:

• It is not the scope of the class to which it has been declared as friend.

• Since, it is not in scope of the class, it cannot be called using the object in the class.

• It can be invoked like a normal function without the help of any object.

• Unlike member functions, it cannot access the function names directly and has to use an object name and membership operator (accordingly) with each member name.

• It can be declared either in the public or private part of a class without affecting its meaning.

• Usually, it has objects as arguments.

Following is just a naive example of friend function:

**#include<iostream>**

**using namespace std;**

**class sample**

**{**

**int a;**

**int b;**

**public:**

**void setValue() {a=25,b=40;}**

**friend float mean(sample s);**

**};**

//notice that it is defined like a normal function

**float mean(sample s)**

**{**

**return float(s.a+s.b)/2.0;**

**}**

//notice that a friend function can access the private elements of an object.

//here, it is accessing s.a and s.b

**int main()**

**{**

**Sample X;**

**X.setValue();**

**cout<<”Mean value =”mean(x)<<”\n”;**

//notice that it is called without object

**return 0;**

**}**

Now, check the first point, it is not the scope of the class, to which a function is declared friend.

Hence, friend function **is not defined as:**

**float sample::mean(sample s)**

**{**

**//code**

**}**

It is defined as:

**float mean(sample s)**

**{**

**return float(s.a+s.b)/2.0;**

**}**

However, check the following examples to learn new things:

**Using friend function to add data objects of two different class:**   
  
**#include<iostream>**

**using namespace std;**

**class ABC;**

//this forward declaration is to be done

**class xyz  
{**

**int data;**

**public:**

**void setValue(int value)**

**{**

**data=value;**

**}**

**friend void add(XYZ,ABC);**

//for this line, we need that forward declaration

//class ABC;

**};**

**class ABC**

**{**

**Int data;**

**public:**

**void setValue(int value)**

**{**

**data=value;**

**}**

**friend void add(XYZ,ABC);**

//XYZ does not need forward declaration

**};**

//since, this friend function is going to add objects of two different class

**void add(XYZ obj1,ABC obj2)**

**{**

**cout<<”Sum of data volume of xyz and abc objeects using friend function: “<<obj1.data+obj2.data;**

**}**

//note that, during definition we do not use the friend keyword

**int main()**

**{**

**XYZ x;**

**ABC a;**

**x.setValue(5);**

**a.setValue(50);**

**add(x,a);**

**return 0;**

**}**

Now, generally friend function is used for operator overloading.-

**Overloading Operator using friend:**

Suppose, you are overloading the binary ‘+’ operator for complex number. For complex number we have written a separate class, which has two data members: real and imaginary.

Now, overloading it using operator overloading function: thee syntax should be like the following:

Declaration:

**friend Complex operator+(Complex a,Complex b);**

Definition:

**Complex operator+(Complex a,Complex b)**

**{**

**return Complex((a.x+b.x),(a.y+b.y));**

**}**

**27.Where A Friend Cannot Be Used For Operator Overloading:  
  
=** Assignment operator.

**()** Function call operator.

**[]** Subscripting operator.

**->** Class member access operator.  
  
(class member access operator -> cannot be overloaded with friend function. But, can . (class member operator) be overloaded with friend operator??)

**28.Example of One Case When Friend is Necessary:**

**#include<bits/stdc++.h>**

**using namespace std;**

**class myString**

**{**

**private:**

**char \*str;**

**int curr\_str\_size;**

**int curr\_position;**

**public:**

**myString();**

**myString(char \*str);**

**myString(const myString &);**

**myString& operator=(const myString &);**

**myString& operator=(const char \*);**

**friend ostream& operator<<(ostream &,const myString &);**

**//**it will return a reference to the ostream(output stream) and it is overriding <<.

**//it cannot be part of this**

**//hence, friend function**

**//subscript operator overloading:**

**//gives direct access to the ith character of a string**

**char operator[](int);**

**int length();**

**void push\_back(char);**

**char pop\_back();**

**};**

**myString::myString()**

**{**

**//if no size is specified**

**//one default size can be given**

**//let's suppose that default size is 32**

**cout<<"Default constructor is called"<<endl;**

**try**

**{**

**str=new char[32];**

**curr\_str\_size=32;**

**curr\_position=0;**

**}**

**catch(bad\_alloc e)**

**{**

**throw ("Further string can not be created. As further heap memory cannpot be allocated");**

**}**

**}**

**myString::myString(char \*str)**

**{**

**cout<<"parametric constructor is called"<<endl;**

**if(str==NULL)**

**{**

**throw ("The str which is passed as argument to this function is NULL");**

**}**

**//however, it does not handle if the str is not null terminated**

**this->curr\_str\_size=strlen(str)+1;**

**this->curr\_position=strlen(str);**

**this->str=new char[this->curr\_str\_size];**

**strcpy(this->str,str);**

**}**

**myString::myString(const myString &myStr)**

**{**

**cout<<"Copy constructor is called"<<endl;**

**//self reference checking**

**if(this==&myStr)**

**{**

**return;**

**}**

**//why const reference sending is important**

**//const because, we need not the argument to be modified**

**//reference because, otherwise copy constructor will be invoked for the argument itself**

**//delete str;**

**//here, you cannot delete it, because, it is an uninitialized pointer**

**//uninitialized pointer derefenecning or deleting can cause segmentation fault**

**//hence, this will does not cause segmentation fault**

**this->curr\_str\_size=strlen(myStr.str)+1;**

**this->curr\_position=strlen(myStr.str);**

**str=new char[this->curr\_str\_size];**

**strcpy(str,myStr.str);**

**//in copy constructor, we can access other object's private member. But, the object**

**//should be of same class**

**}**

**//assignment operator overloading**

**myString& myString::operator=(const myString &myStr)**

**{**

**//self reference checking**

**cout<<"Assignment operator is overloaded"<<endl;**

**if(this==&myStr)**

**{**

**return \*this;**

**}**

**delete str;**

**//it was never initialized to null**

**//hence, this will does not cause segmentation fault**

**/\*Important for this kind of stupid operation we do every day**

**myString str;**

**str="Sayak Haldar";**

**\*/**

**this->curr\_str\_size=strlen(myStr.str)+1;**

**str=new char[this->curr\_position];**

**strcpy(str,myStr.str);**

**return \*this;**

**}**

**myString& myString::operator=(const char\* str)**

**{**

**cout<<"Assignment operator is overloaded"<<endl;**

**//no need to self reference checking**

**//because, it is not accepting a string as an argument. Rather, it will accept a const char as an argument**

**delete this->str;**

**this->curr\_position=strlen(str);**

**this->curr\_str\_size=this->curr\_position;**

**strcpy(this->str,str);**

**return \*this;**

**}**

**char myString::operator[](int index)**

**{**

**if(index<0||index>=curr\_position)**

**{**

**throw ("Index out of bound exception");**

**}**

**return str[index];**

**}**

**int myString::length()**

**{**

**return this->curr\_position;**

**}**

**void myString::push\_back(char c)**

**{**

**if(this->curr\_position+1>=this->curr\_str\_size)**

**{**

**//we need to realloc**

**char \*tempStr=new char[this->curr\_str\_size+1];**

**strcpy(tempStr,str);**

**delete str;**

**//now, tempStr is not a pointer alias**

**//hence, it will remain fine even if we delete str**

**str=new char[this->curr\_str\_size+1+32];**

**//allocating some extra space to make push\_back operation little less time consuming**

**this->curr\_str\_size=this->curr\_str\_size+32;**

**strcpy(str,tempStr);**

**str[this->curr\_position]=c;**

**str[this->curr\_position+1]='\0';**

**//very important, otherwise it wont be null terminated**

**this->curr\_position=this->curr\_position+1;**

**delete tempStr;**

**//otherwise, there will be a memory leak**

**}**

**//now, it good for checking**

**//however, we still need to remember that curr\_position is in 1 indexing**

**else**

**{**

**str[this->curr\_position]=c;**

**str[this->curr\_position+1]='\0';**

**this->curr\_position=this->curr\_position+1;**

**}**

**}**

**char myString::pop\_back()**

**{**

**char c=str[curr\_position-1];**

**//remember it is a null terminated string and curr\_postion is in 1 indexing**

**//hence,str[curr\_position] will actually contain '\0';**

**str[curr\_position-1]='\0';**

**return c;**

**}**

**ostream& operator<<(ostream &os,const myString &myStr)**

**{**

**return os<<myStr.str;**

**}**

**int main()**

**{**

**myString str1;**

**//this will invoke default constructor**

**str1="Sayak Haldar";**

**//this will invoke assginment operator**

**cout<<str1<<endl;**

**myString str2("Surekha Haldar");**

**//this will invoke parametric constructor**

**cout<<str2<<endl;**

**myString str3(str2);**

**//this will invoke copy constructor**

**cout<<str3<<endl;**

**myString str4;**

**str4=str1;**

**//this will invoke assignment operator**

**cout<<str4<<endl;**

**}**

This is one such case. If you want to enable your own designed string class to be compatible with function cout, you have to declare a friend function:

**friend ostream& operator<<(ostream &,const myString &);**

And, you have to define the function like this:

**ostream& operator<<(ostream &os,const myString &myStr)**

**{**

**return os<<myStr.str;**

**}**

You cannot do it by overloading the operator << as a member function:

**//Implementing my own string class**

**#include<bits/stdc++.h>**

**using namespace std;**

**class myString**

**{**

**private:**

**char \*str;**

**int curr\_str\_size;**

**int curr\_position;**

**public:**

**myString();**

**myString(char \*str);**

**myString(const myString &);**

**myString& operator=(const myString &);**

**myString& operator=(const char \*);**

**ostream& operator<<(ostream &);**

**//it cannot be part of this**

**//hence, friend function**

**//subscript operator overloading:**

**//gives direct access to the ith character of a string**

**char operator[](int);**

**int length();**

**void push\_back(char);**

**char pop\_back();**

**};**

**myString::myString()**

**{**

**//if no size is specified**

**//one default size can be given**

**//let's suppose that default size is 32**

**cout<<"Default constructor is called"<<endl;**

**try**

**{**

**str=new char[32];**

**curr\_str\_size=32;**

**curr\_position=0;**

**}**

**catch(bad\_alloc e)**

**{**

**throw ("Further string can not be created. As further heap memory cannpot be allocated");**

**}**

**}**

**myString::myString(char \*str)**

**{**

**cout<<"parametric constructor is called"<<endl;**

**if(str==NULL)**

**{**

**throw ("The str which is passed as argument to this function is NULL");**

**}**

**//however, it does not handle if the str is not null terminated**

**this->curr\_str\_size=strlen(str)+1;**

**this->curr\_position=strlen(str);**

**this->str=new char[this->curr\_str\_size];**

**strcpy(this->str,str);**

**}**

**myString::myString(const myString &myStr)**

**{**

**cout<<"Copy constructor is called"<<endl;**

**//self reference checking**

**if(this==&myStr)**

**{**

**return;**

**}**

**//why const reference sending is important**

**//const because, we need not the argument to be modified**

**//reference because, otherwise copy constructor will be invoked for the argument itself**

**//delete str;**

**//here, you cannot delete it, because, it is an uninitialized pointer**

**//uninitialized pointer derefenecning or deleting can caus segmentation fault**

**//hence, this will does not cause segmentation fault**

**this->curr\_str\_size=strlen(myStr.str)+1;**

**this->curr\_position=strlen(myStr.str);**

**str=new char[this->curr\_str\_size];**

**strcpy(str,myStr.str);**

**//in copy constructor, we can access other object's private member. But, the object**

**//should be of same class**

**}**

**//assignment operator overloading**

**myString& myString::operator=(const myString &myStr)**

**{**

**//self reference checking**

**cout<<"Assignment operator is overloaded"<<endl;**

**if(this==&myStr)**

**{**

**return \*this;**

**}**

**delete str;**

**//it was never initialized to null**

**//hence, this will does not cause segmentation fault**

**/\*Important for this kind of stupid operation we do every day**

**myString str;**

**str="Sayak Haldar";**

**\*/**

**this->curr\_str\_size=strlen(myStr.str)+1;**

**str=new char[this->curr\_position];**

**strcpy(str,myStr.str);**

**return \*this;**

**}**

**myString& myString::operator=(const char\* str)**

**{**

**cout<<"Assignment operator is overloaded"<<endl;**

**//no need to self reference checking**

**delete this->str;**

**this->curr\_position=strlen(str);**

**this->curr\_str\_size=this->curr\_position;**

**strcpy(this->str,str);**

**return \*this;**

**}**

**char myString::operator[](int index)**

**{**

**if(index<0||index>=curr\_position)**

**{**

**throw ("Index out of bound exception");**

**}**

**return str[index];**

**}**

**int myString::length()**

**{**

**return this->curr\_position;**

**}**

**void myString::push\_back(char c)**

**{**

**if(this->curr\_position+1>=this->curr\_str\_size)**

**{**

**//we need to realloc**

**char \*tempStr=new char[this->curr\_str\_size+1];**

**strcpy(tempStr,str);**

**delete str;**

**//now, tempStr is not a pointer alias**

**//hence, it will remain fine even if we delete str**

**str=new char[this->curr\_str\_size+1+32];**

**//allocating some extra space to make push\_back operation little less time consuming**

**this->curr\_str\_size=this->curr\_str\_size+32;**

**strcpy(str,tempStr);**

**str[this->curr\_position]=c;**

**str[this->curr\_position+1]='\0';**

**//very important, otherwise it wont be null terminated**

**this->curr\_position=this->curr\_position+1;**

**delete tempStr;**

**//otherwise, there will be a memory leak**

**}**

**//now, it good for checking**

**//however, we still need to remember that curr\_position is in 1 indexing**

**else**

**{**

**str[this->curr\_position]=c;**

**str[this->curr\_position+1]='\0';**

**this->curr\_position=this->curr\_position+1;**

**}**

**}**

**char myString::pop\_back()**

**{**

**char c=str[curr\_position-1];**

**//remember it is a null terminated string and curr\_postion is in 1 indexing**

**//hence,str[curr\_position] will actually contain '\0';**

**str[curr\_position-1]='\0';**

**return c;**

**}**

**ostream& myString::operator<<(ostream &os)**

**{**

**return os<<str;**

**}**

**int main()**

**{**

**myString str1;**

**//this will invoke default constructor**

**str1="Sayak Haldar";**

**//this will invoke assginment operator**

**cout<<str1<<endl;**

**myString str2("Surekha Haldar");**

**//this will invoke parametric constructor**

**cout<<str2<<endl;**

**myString str3(str2);**

**//this will invoke copy constructor**

**cout<<str3<<endl;**

**myString str4;**

**str4=str1;**

**//this will invoke assignment operator**

**cout<<str4<<endl;**

**}**

**29.Friend Class:**

**Friend Class:  
  
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**  **};** |

**A complete example of friend class:**

#include<cstdio>

#include<iostream>

class Node

{

private:

int key;

Node \*next;

public:

Node();

Node(int);

int get\_key();

Node\* get\_node();

void set\_next(Node \*);

~Node();

friend class LinkedList;

};

class LinkedList

{

private:

Node \*head;

public:

LinkedList();

void insert\_at\_beginning(int);

void insert\_at\_end(int);

void display\_linked\_list();

~LinkedList();

};

//friend Node LinkedList;

Node::Node()

{

//an empty contructor

}

Node::Node(int key)

{

this->key=key;

this->next=NULL;

}

int Node::get\_key()

{

return this->key;

}

Node\* Node::get\_node()

{

//Now, this is a pointer

//could we return this?

return this;

}

void Node::set\_next(Node \*to\_be\_next)

{

this->next=to\_be\_next;

}

Node::~Node()

{

//free next;

delete next;

//otherwise, the memory will remain allocated in case of a big program

//and we cannot reuse the heap memory

}

LinkedList::LinkedList()

{

head=NULL;

}

void LinkedList::insert\_at\_beginning(int data)

{

//now, since, LinkedList is a friend class of Node we can access the private members directly

Node \*curr=new Node();

//allocate memory for pointer to a class dynamically

curr->key=data;

curr->next=NULL;

//since, LinkedList is a friend class

if(head==NULL)

{

head=curr;

return;

}

curr->next=head;

head=curr;

//implmentation of singly linked list

}

void LinkedList::insert\_at\_end(int data)

{

Node \*curr=new Node();

//now, sincce, LinkedList is a friend class of Node

//we can access the private members directly

curr->key=data;

curr->next=NULL;

if(head==NULL)

{

head=curr;

return;

}

Node \*index\_node=head;

for(;index\_node->next!=NULL;index\_node=index\_node->next)

{

}

index\_node->next=curr;

//this is done

}

void LinkedList::display\_linked\_list()

{

printf("The linkedlist is: ");

Node \*index\_node=head;

for(;index\_node!=NULL;index\_node=index\_node->next)

{

if(index\_node==head)

{

printf("%d",index\_node->key);

}

else

{

printf(" %d",index\_node->key);

}

}

}

LinkedList::~LinkedList()

{

//free head;

delete head;

//generally memory allocated by new will be freed by delete function

}

int main()

{

LinkedList curr\_list;

//default constructor will be invoked

curr\_list.insert\_at\_end(2);

curr\_list.insert\_at\_beginning(1);

curr\_list.insert\_at\_end(3);

curr\_list.display\_linked\_list();

return 0;

//now, curr\_list goes out of scope

//so, destructor will be invoked

}

**Derived DataType Basics:**

**30.What is the difference of class and structure in c++ context?**

Answer) The conventional structure in c cannot contain functions (Functions which operate on data items contained by the function). But that is not the problem with c++ structures. In case of c++, structures can hold both data members and the functions operating on them. But, in c++, the members of a structure are public (by default). So, it violates one of the most important OOP characteristics-Encapsulation. Whereas, the members of a class are by default private.

Point wise difference:

1) c++ structs have public members by default and classes have private members by default.

2) Bases of a struct are inherited publicly by default, whereas bases of a class are inherited privately by default.

Below is the example of a struct having member functions:

using std::cout;

using std::cin;

using std::endl;

struct person\_str

{

string name;

int age;

//constructor

person\_str()

{

name = "default";

age = 77;

}

void print();

};

void person\_str::print()

{

/\*

we don't have to mention what "name" and "age" are,

because it automatically refers back to the

member variables.

The "this" variable is the implied first argument,

which is a pointer to the object that the member it was called on.

\*/

cout << name << " " << this->age << endl;

}

int main()

{

person\_str p1;

p1.name = "Chrys";

p1.age = 26;

p1.print();

}

**31.What is constructor? Mention some points about constructor.**

Answer) Constructor is a special member function of the class by which an object of the class can get initiated. It provides the same kind of initiation feature enjoyed by the in built data types to the derived data types.

* It is a member function which is used to initialize the object
* It has the same name as the class
* It is invoked in the following situations:
* when an object of the class is created
  + When an object of the class is created dynamically
  + It is invoked as many times as the number of objects in the class
* It is not invoked in the following situations:
  + when a pointer of the class is specified. (why is is not invoked is discussed later)
* It has no return type

**Consider the following:**

(1. myClass class1;

2. myClass\* class1;

3. myClass\* class1 = new myClass;

**Is Constructor invoked:**

Yes - default constructor, instance created on stack

No

Yes - default constructor, instance created on heap

The statement would instantiate an object on the stack, call constructor.

Defines only a pointer variable on the stack, no constructor is called.

The new operator would create an object in free store (usually the heap) and call constructor

)

**32.Mention Some Points About Destructors:**

Any clean up operation of an object can be done by another special function named destructor.

● It is a member function used to release the resources.

●It has same name as a class, proceeded by a '~'

● It is invoked when an object is removed or goes out of scope

●It is not invoked when a pointer to the class goes out of scope

●It is invoked as many times as the number of elements in the array

●It has no return type

●It cannot have any parameters

●It has no return type.

●It cannot be overloaded (obviously, since, destructor has no return type and cannot have any parameters

(Function Signature. A function's signature includes the function's name and the number, order and type of its formal parameters. Two overloaded functions must not have the same signature. The return value is not part of a function's signature.

Now, to overload a function, the arguments must be different. Either their order or their number. Now, since, destructor does not accept any arguments, it cannot be overloaded)

**33.What are the default members of a class?**

Answer) By default, if not implemented by the user, the compiler adds some member functions to

the class. Those are:

• Default constructor

• Destructor

• Copy constructor

• Assignment operator

However, you should note the following points:

If the constructor is overloaded (like, a parametric constructor is only defined for the class, no longer the default constructor introduced by compiler will work. Because, during compilation if compiler finds a parametric constructor, it will not generate a default constructor definition.

**For instance, the following code will work fine:**

#include<bits/stdc++.h>

using namespace std;

class Dog

{

private:

string species;

double height;

string skin\_color;

int age;

public:

Dog(string,double,string,int);

Dog(const Dog &);

//we should not have an overloaded definition of assignment operator

//since, a Dog is a living species

//Dog operator=(const Dog &);

//we need not to Because, there is no pointer variable

//hence, the default definition of destructor can do the job

//~Dog();

void display\_info();

};

//since, there is no return type for constructor

Dog::Dog(string species,double height,string skin\_color,int age)

{

if(species=="")

//cannot do null checking for string

{

throw "species of a Dog cannot be empty";

}

if(height<=0)

{

throw "The height of a Dog cannot be <=0";

}

if(skin\_color=="")

{

throw "Skin color of a Dog cannot be empty";

}

if(age<=0)

{

throw "the age of a Dog cannot be <=0";

}

this->species=species;

this->height=height;

this->skin\_color=skin\_color;

this->age=age;

}

//though conceptually, there should not be a copy constructor

Dog::Dog(const Dog &temp\_dog)

{

species=temp\_dog.species;

//reference is sent, instead of pointer. Hence, no dereferencing

height=temp\_dog.height;

skin\_color=temp\_dog.skin\_color;

age=temp\_dog.age;

}

void Dog::display\_info()

{

cout<<"The Dog's species is"<<species<<endl;

cout<<"The Dog's height is:"<<height<<" cm"<<endl;

cout<<"The Dog's skin color is: "<<skin\_color<<endl;

cout<<"The Dog's age is: "<<age<<endl;

}

int main()

{

Dog labrador("Labrador Retriever",50,"Chocolate",5);

labrador.display\_info();

//Dog pug;

return 0;

}

**This will not work fine.**

#include<bits/stdc++.h>

using namespace std;

class Dog

{

private:

string species;

double height;

string skin\_color;

int age;

public:

Dog(string,double,string,int);

Dog(const Dog &);

//we should not have an overloaded definition of assignment operator

//since, a Dog is a living species

//Dog operator=(const Dog &);

//we need not to Because, there is no pointer variable

//hence, the default definition of destructor can do the job

//~Dog();

void display\_info();

};

//since, there is no return type for constructor

Dog::Dog(string species,double height,string skin\_color,int age)

{

if(species=="")

//cannot do null checking for string

{

throw "species of a Dog cannot be empty";

}

if(height<=0)

{

throw "The height of a Dog cannot be <=0";

}

if(skin\_color=="")

{

throw "Skin color of a Dog cannot be empty";

}

if(age<=0)

{

throw "the age of a Dog cannot be <=0";

}

this->species=species;

this->height=height;

this->skin\_color=skin\_color;

this->age=age;

}

//though conceptually, there should not be a copy constructor

Dog::Dog(const Dog &temp\_dog)

{

species=temp\_dog.species;

//reference is sent, instead of pointer. Hence, no dereferencing

height=temp\_dog.height;

skin\_color=temp\_dog.skin\_color;

age=temp\_dog.age;

}

void Dog::display\_info()

{

cout<<"The Dog's species is"<<species<<endl;

cout<<"The Dog's height is:"<<height<<" cm"<<endl;

cout<<"The Dog's skin color is: "<<skin\_color<<endl;

cout<<"The Dog's age is: "<<age<<endl;

}

int main()

{

//Dog labrador("Labrador Retriever",50,"Chocolate",5);

//labrador.display\_info();

Dog pug;

return 0;

}

main.cpp:66:9: error: no matching function for call to 'Dog::Dog()'

Dog pug;

**Note that: you can skip the destructor part as long as there is no dynamic memory allocation.**

#include<bits/stdc++.h>

using namespace std;

class Dog

{

private:

string species;

double height;

string skin\_color;

int age;

public:

//Dog(string,double,string,int);

Dog(const Dog &);

//we should not have an overloaded definition of assignment operator

//since, a Dog is a living species

//Dog operator=(const Dog &);

//we need not to Because, there is no pointer variable

//hence, the default definition of destructor can do the job

//~Dog();

void display\_info();

};

//since, there is no return type for constructor

/\*Dog::Dog(string species,double height,string skin\_color,int age)

{

if(species=="")

//cannot do null checking for string

{

throw "species of a Dog cannot be empty";

}

if(height<=0)

{

throw "The height of a Dog cannot be <=0";

}

if(skin\_color=="")

{

throw "Skin color of a Dog cannot be empty";

}

if(age<=0)

{

throw "the age of a Dog cannot be <=0";

}

this->species=species;

this->height=height;

this->skin\_color=skin\_color;

this->age=age;

}\*/

//though conceptually, there should not be a copy constructor

Dog::Dog(const Dog &temp\_dog)

{

species=temp\_dog.species;

//reference is sent, instead of pointer. Hence, no dereferencing

height=temp\_dog.height;

skin\_color=temp\_dog.skin\_color;

age=temp\_dog.age;

}

void Dog::display\_info()

{

cout<<"The Dog's species is"<<species<<endl;

cout<<"The Dog's height is:"<<height<<" cm"<<endl;

cout<<"The Dog's skin color is: "<<skin\_color<<endl;

cout<<"The Dog's age is: "<<age<<endl;

}

int main()

{

//Dog labrador("Labrador Retriever",50,"Chocolate",5);

//labrador.display\_info();

Dog pug;

return 0;

}

Now, check the following. This class has an overloaded constructor. (copy constructor) Hence, the default constructor must be overloaded. Otherwise, Dog pug this line will generate compilation error.

**34.Why constructor is better than init function?**

The use of functions such as init() to provide initialization for class objects in inelegant and error prone. Because, it is nowhere stated that the object must be initialized, a programmer can forget to do so- or do so twice. A better approach is to allow the programmer to declare a function with the entire purpose of initializing object-constructor.

**35.What is copy elision in c++?**

Answer) Copy elision (or Copy omission) is a compiler optimization technique that avoids unnecessary copying of objects. Now a days, almost every compiler uses it. Let us understand it with the help of an example.

**#include <iostream>**

**using namespace std;**

**class B**

**{**

**public:**

**B(const char\* str = "\0") //default constructor**

**{**

**cout<<"Constructor called"<<endl;**

**}**

**B(const B &b) //copy constructor**

**{**

**cout << "Copy constructor called" << endl;**

**}**

**};**

**int main()**

**{**

**B ob ="copy me";**

**return 0;**

**}**

The output of above program is:

**Constructor called**

According to theory, when the object “ob” is being constructed, one argument constructor is used to convert “copy me” to a temporary object & that temporary object is copied to the object “ob”.

However, most of the C++ compilers avoid such overheads of creating a temporary object & then copying it.

The modern compilers break down the statement:

B ob = "copy me"; //copy initialization

as

B ob("copy me"); //direct initialization

and thus eliding call to copy constructor.

However, if we still want to ensure that the compiler doesn’t elide the call to copy constructor [disable the copy elision], we can compile the program using “-fno-elide-constructors” option with g++ and see the output as following: (If the source code name is copyelision.cpp)

**g++ copyelision.cpp -fno-elide-constructors**

**36.Why is the object reference sent as constant in case of copy constructor in c++?**

Answer) When we create a copy constructor for a class, we pass an object by reference and we generally pass it as a const reference. The reasons for doing it are the followings:

* We make it const, so that the object is not accidentally modified by the copy constructor function.
* Another reason is, temporary objects created during the execution of the function can not be bound to non const reference. **(non const reference cannot be assigned with const l value or r value. Is that we are talking about?)**

**37.Why is the object reference sent as reference in case of copy constructor in c++?**

If its not passed by reference then it would pass by value. If the argument is passed by value, its copy constructor would call itself to copy the actual parameter to formal parameter. This process would go on (recursive calling) until the system runs out of memory. So, we should pass it by reference , so that copy constructor does not get invoked.

(In practical scenario, if you don’t pass an object’s reference to copy constructor, it will give you **compilation error**)

For instance, you write the following code:

**#include<bits/stdc++.h>**

**using namespace std;**

**class Dog**

**{**

**private:**

**string species;**

**double height;**

**string skin\_color;**

**int age;**

**public:**

**Dog(string,double,string,int);**

**Dog(const Dog );**

**//we should not have an overloaded definition of assignment operator**

**//since, a Dog is a living species**

**//Dog operator=(const Dog &);**

**//we need not to Because, there is no pointer variable**

**//hence, the default definition of destructor can do the job**

**//~Dog();**

**void display\_info();**

**};**

**//since, there is no return type for constructor**

**Dog::Dog(string species,double height,string skin\_color,int age)**

**{**

**if(species=="")**

**//cannot do null checking for string**

**{**

**throw "species of a Dog cannot be empty";**

**}**

**if(height<=0)**

**{**

**throw "The height of a Dog cannot be <=0";**

**}**

**if(skin\_color=="")**

**{**

**throw "Skin color of a Dog cannot be empty";**

**}**

**if(age<=0)**

**{**

**throw "the age of a Dog cannot be <=0";**

**}**

**this->species=species;**

**this->height=height;**

**this->skin\_color=skin\_color;**

**this->age=age;**

**}**

**//though conceptually, there should not be a copy constructor**

**Dog::Dog(const Dog temp\_dog)**

**{**

**species=temp\_dog.species;**

**//reference is sent, instead of pointer. Hence, no dereferencing**

**height=temp\_dog.height;**

**skin\_color=temp\_dog.skin\_color;**

**age=temp\_dog.age;**

**}**

**void Dog::display\_info()**

**{**

**cout<<"The Dog's species is"<<species<<endl;**

**cout<<"The Dog's height is:"<<height<<" cm"<<endl;**

**cout<<"The Dog's skin color is: "<<skin\_color<<endl;**

**cout<<"The Dog's age is: "<<age<<endl;**

**}**

**int main()**

**{**

**Dog labrador("Labrador Retriever",50,"Chocolate",5);**

**labrador.display\_info();**

**//Dog pug;**

**return 0;**

**}**

It will generate compilation error stating:

**error: invalid constructor; you probably meant 'Dog (const Dog&)'**

**Dog(const Dog);**

**The condition is little bit relaxing when you define copy constructor in java. Because, there is no standard provided. Because, java compiler does not include a copy constructor by default and when you define it you are not overloading the default copy constructor definition provided.**

**38.What are the difference between swallow copy and deep copy?**

Answer) A shallow copy of an object copies all of the member field values. This works well if the fields are values, but may not be what you want for fields that point to dynamically allocated memory. The pointer will be copied. but the memory it points to will not be copied -- the field in both the original object and the copy will then point to the same dynamically allocated memory,

which is not usually what you want. The default copy constructor and assignment operator make shallow copies.

A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. To make a deep copy, you must write a copy constructor and overload the assignment operator, otherwise the copy will point to the original, with disastrous consequences.

(so prevent from generating a swallow copy, we need self reference checking. Otherwise, it will be modified again)

**(swallow copy can generate undefined behaviour. If class has a pointer variable)**

**39.Why is Self Checking Required In Case Of Assignment Operator Overloading:**

For example, consider the following class Array and overloaded assignment operator function without self assignment check.

**// A sample class**

**class Array**

**{**

**private:**

**int \*ptr;**

**Int size;**

**public:**

**Array& operator = (const Array &rhs);**

**// constructors and other functions of class........**

**};**

**// Overloaded assignment operator for class Array (without self**

**// assignment check)**

**Array& Array::operator = (const Array &rhs)**

**{**

**// Deallocate old memory**

**delete [] ptr;**

**// allocate new space**

**ptr = new int [rhs.size];**

**// copy values**

**size = rhs.size;**

**for(int i = 0; i < size; i++)**

**ptr[i] = rhs.ptr[i];**

**return \*this;**

**}**

If we have an object say a1 of type Array and if we have a line like a1 = a1 somewhere, the program results in unpredictable behavior because there is no self assignment check in the above code. To avoid the above issue, self assignment check must be there while overloading assignment operator. For example, following code does self assignment check.

**// Overloaded assignment operator for class Array (with self**

**// assignment check)**

**Array& Array::operator = (const Array &rhs)**

**{**

**/\* SELF ASSIGNMENT CHECK \*/**

**if(this != &rhs)**

**{**

**// Deallocate old memory**

**delete [] ptr;**

**// allocate new space**

**ptr = new int [rhs.size];**

**// copy values**

**size = rhs.size;**

**for(int i = 0; i < size; i++)**

**ptr[i] = rhs.ptr[i];**

**}**

**return \*this;**

**}**

**40)Why assignment operator overloading requires returning a reference?**

A bit of clarification as to why it's preferable to return by reference for operator= versus return by value --- as the chain a = b = c will work fine if a value is returned.

If you return a reference, minimal work is done. The values from one object are copied to another object.

However, if you return by value for operator=, you will call a constructor AND destructor EACH time that the assignment operator is called!!

So, given:

A& operator=(const A& rhs) { /\* ... \*/ };

Then,

a = b = c; **// calls assignment operator above twice. Nice and simple.**

But,

A operator=(const A& rhs) { /\* ... \*/ };

a = b = c; **// calls assignment operator twice, calls copy constructor twice, calls destructor type to delete the temporary values! Very wasteful and nothing gained!**

In sum, there is nothing gained by returning by value, but a lot to lose.

For instance, check the following code:

**#include<bits/stdc++.h>**

**using namespace std;**

**class TestClass**

**{**

**private:**

**int a;**

**public:**

**TestClass();**

**TestClass(int);**

**TestClass(const TestClass &);**

**TestClass& operator=(const TestClass &);**

**};**

**TestClass::TestClass()**

**{**

**cout<<"The default constructor is called"<<endl;**

**a=0;**

**}**

**TestClass::TestClass(int a)**

**{**

**cout<<"The parametric constructor is called"<<endl;**

**this->a=a;**

**}**

**TestClass::TestClass(const TestClass &testClassObj)**

**{**

**cout<<"The copy constructor is called"<<endl;**

**if(this==&testClassObj)**

**{**

**//self reference checking**

**return;**

**}**

**//there is no dynamic memory allocation. Hence, no swallow copy prevention or**

**//code block to prevent the accidental modification of same memory reference**

**a=testClassObj.a;**

**//reference is there. So, testClassObj.a will do the job**

**}**

**TestClass& TestClass::operator=(const TestClass &testClassObj)**

**{**

**cout<<"The overloaded assignment operator is called"<<endl;**

**if(this==&testClassObj)**

**{**

**//self reference checking**

**return \*this;**

**}**

**a=testClassObj.a;**

**}**

**int main()**

**{**

**TestClass a(10),b,c;**

**c=b=a;**

**return 0;**

**}**

It will generate:

The parametric constructor is called (for a)

The default constructor is called (for b)

The default constructor is called (for c)

The overloaded assignment operator is called (for assignment b=a)

The overloaded assignment operator is called (for assignment c=b)

And finally segmentation fault

While the following one does not generate segmentation call, but now, lots of constructors is being called:

**#include<bits/stdc++.h>**

**using namespace std;**

**class TestClass**

**{**

**private:**

**int a;**

**public:**

**TestClass();**

**TestClass(int);**

**TestClass(const TestClass &);**

**TestClass operator=(const TestClass &);**

**~TestClass();**

**};**

**TestClass::TestClass()**

**{**

**cout<<"The default constructor is called"<<endl;**

**a=0;**

**}**

**TestClass::TestClass(int a)**

**{**

**cout<<"The parametric constructor is called"<<endl;**

**this->a=a;**

**}**

**TestClass::TestClass(const TestClass &testClassObj)**

**{**

**cout<<"The copy constructor is called"<<endl;**

**if(this==&testClassObj)**

**{**

**//self reference checking**

**return;**

**}**

**//there is no dynamic memory allocation. Hence, no swallow copy prevention or**

**//code block to prevent the acciddental modification of same memory reference**

**a=testClassObj.a;**

**//reference is there. So, testClassObj.a will do the job**

**}**

**TestClass TestClass::operator=(const TestClass &testClassObj)**

**{**

**cout<<"The overloaded assignment operator is called"<<endl;**

**if(this==&testClassObj)**

**{**

**//self reference checking**

**return \*this;**

**}**

**a=testClassObj.a;**

**}**

**TestClass::~TestClass()**

**{**

**cout<<"The destructor is called"<<endl;**

**}**

**int main()**

**{**

**TestClass a(10),b,c;**

**c=b=a;**

**return 0;**

**}**

This will generate the following output:

**The parametric constructor is called**

**The default constructor is called**

**The default constructor is called**

**The overloaded assignment operator is called**

**The overloaded assignment operator is called**

**The destructor is called**

**The destructor is called**

**The destructor is called**

**The destructor is called**

**41.What is this pointer?**

Answer) The keyword 'this' defines a very special type of pointer in case of c++. If an object of a class is created which has a non static member function, when the non static member function is called, this pointer **stores the address of the object**. **When a non static member is called, the 'this' pointer is added as an extra hidden argument.** It can be used in many contexts. Like, within assignment operator overloading function it is used for self checking. **However, a static member**

**function does not have a this pointer associated to it.**

**Note:** A static member function does not have this pointer associated with it.

**42.Mention the implicit and explicit calling procedures of default constructor, parametric constructor, copy constructor.**

Answer)

**Default constructor:**

**Implicit calling:** Car car1;

**Explicit calling:** Car car2=Car();

**Parametric Constructor:**

**Implicit calling:** Car car3("Mercedes Benz",2845000,"Black",4,"Mercedes-Benz A-Class A 200 CDI");

**Explicit calling:** Car car4=Car("Mercedes Benz",2845000,"Black",4,"Mercedes-Benz A-Class A 200 CDI");

**Copy Constructor:**

**Implicit calling:** Car car5(car4);

**Explicit calling:** Car car6=car4;

**Mention the order of constructor calling and destructor calling in case of creation of an object of derived class in c++.**

**#include<iostream>**

**using namespace std;**

**class A**

**{**

**public:**

**A(){cout<<"Base class constructor called"<<endl;}**

**~A(){cout<<"Base class destructor called"<<endl;}**

**};**

**class B:public A**

**{**

**public:**

**B(){cout<<"Derived class constructor called"<<endl;}**

**~B(){cout<<"derived class destructor called"<<endl;}**

**};**

**int main()**

**{**

**B b1;**

**return 0;**

**}**

**It will output:**

Base class constructor called

Derived class constructor called

derived class destructor called

Base class destructor called

**43.Initialization In c++**

**Default Initialization:**

Default initialization is performed in three situations:

1) when a variable with automatic, static, or thread-local storage duration is declared with no initializer;

2) when an object with dynamic storage duration is created by a new-expression with no initializer or when an object is created by a new-expression with the initializer consisting of an empty pair of parentheses (until C++03);

3) when a base class or a non-static data member is not mentioned in a constructor initializer list and that constructor is called.

(For more read, <https://en.cppreference.com/w/cpp/language/default_initialization)>

**Direct Initialization:**

1) initialization with a nonempty parenthesized list of expressions or braced-init-lists (since C++11)

2) initialization of an object of non-class type with a single brace-enclosed initializer (note: for class types and other uses of braced-init-list, see list-initialization)

3) initialization of a prvalue temporary by functional cast or with a parenthesized expression list

4) initialization of a prvalue temporary by a static\_cast expression

5) initialization of an object with dynamic storage duration by a new-expression with a non-empty initializer

6) initialization of a base or a non-static member by constructor initializer list

7) initialization of closure object members from the variables caught by copy in a lambda-expression

**(https://en.cppreference.com/w/cpp/language/list\_initialization)**

**Copy Initialization:**

Copy initialization is performed in the following situations:

1) when a named variable (automatic, static, or thread-local) of a non-reference type T is declared with the initializer consisting of an equals sign followed by an expression.

2) (until C++11)when a named variable of a scalar type T is declared with the initializer consisting of an equals sign followed by a brace-enclosed expression (Note: as of C++11, this is classified as list initialization, and narrowing conversion is not allowed).

3) when passing an argument to a function by value

4) when returning from a function that returns by value

5) when throwing or catching an exception by value

6) as part of aggregate initialization, to initialize each element for which an initializer is provided

(For more, read, <https://en.cppreference.com/w/cpp/language/copy_initialization)>

**Aggregate initialization**

Initializes an aggregate from braced-init-list

Aggregate initialization is a form of list-initialization, which initializes aggregates

An aggregate is one of the following types:

array type

class type (typically, struct or union), that has

no private or protected non-static data members

no user-declared constructors (until C++11)

no user-provided constructors (explicitly defaulted or deleted constructors are allowed) (since C++11) (until C++17)

no user-provided, inherited, or explicit constructors (explicitly defaulted or deleted constructors are allowed) (since C++17) (until C++20)

no user-declared or inherited constructors (since C++20)

no virtual, private, or protected (since C++17) base classes

no virtual member functions

no default member initializers(since C++11)(until C++14)

(For more, https://en.cppreference.com/w/cpp/language/aggregate\_initialization

**List Initialization:**

**(I bring out the whole topic, just for this. Because, by the end of this you will learn the thing that a class’s object can be instantiated in more ways)**

List initialization is performed in the following situations:

direct-list-initialization (both explicit and non-explicit constructors are considered)

1) initialization of a named variable with a braced-init-list (that is, a possibly empty brace-enclosed list of expressions or nested braced-init-lists)

2) initialization of an unnamed temporary with a braced-init-list

3) initialization of an object with dynamic storage duration with a new-expression, where the initializer is a brace-init-list

4) in a non-static data member initializer that does not use the equals sign

5) in a member initializer list of a constructor if braced-init-list is used

copy-list-initialization (both explicit and non-explicit constructors are considered, but only non-explicit constructors may be called)

6) initialization of a named variable with a braced-init-list after an equals sign

7) in a function call expression, with braced-init-list used as an argument and list-initialization initializes the function parameter

8) in a return statement with braced-init-list used as the return expression and list-initialization initializes the returned object

9) in a subscript expression with a user-defined operator[], where list-initialization initializes the parameter of the overloaded operator

10) in an assignment expression, where list-initialization initializes the parameter of the overloaded operator

11) functional cast expression or other constructor invocations, where braced-init-list is used in place of a constructor argument. Copy-list-initialization initializes the constructor's parameter (note; the type U in this example is not the type that's being list-initialized; U's constructor's parameter is)

12) in a non-static data member initializer that uses the equals sign

**First Example:**

#include <iostream>

using namespace std;

class bowl

{

public:

int apples;

int oranges;

void display\_bowl\_stats()

{

cout<<"Apples: "<<apples<<endl;

cout<<"Oranges: "<<oranges<<endl;

}

};

int main()

{

bowl firstBowl = { 1, 2 };

firstBowl.display\_bowl\_stats();

return 0;

}

**This will compile time. (probably due to list initialization)  
(** in a member initializer list of a constructor if braced-init-list is used)

**Second Example:**#include <iostream>

using namespace std;

class bowl

{

public:

int apples;

int oranges;

void display\_bowl\_stats()

{

cout<<"Apples: "<<apples<<endl;

cout<<"Oranges: "<<oranges<<endl;

}

};

int main()

{

bowl bowls[1] = { 1, 2 };

bowls[0].display\_bowl\_stats();

return 0;

}

**44.Operator Overloading in c++  
  
Overloading unary operators:**

Let us consider the unary minus operator.

**#include<iostream>**

**using namespace std;**

**class space**

**{**

**private:**

**int x;**

**int y;**

**int z;**

**public:**

**void getdata(int a,int b,int c);**

**void display(void);**

**Void operator-();**

**};**

**void space::getdata(int a,int b,int c)**

**{**

**x=a;**

**y=b;**

**z=c;**

**}**

**void space:: display(void)**

**{**

**cout<<”X=”<<x<<””;**

**cout<<”Y=”<<y<<””;**

**cout<<”Z=”<<z<<””;**

**}**

**void space::operator-()**

**{**

**//overloading unary operator. Now, it cannot take any argument.**

**x=-x;**

**y=-y;**

**z=-z;**

**}**

**int main()**

**{**

**space S;**

**S.getdata(10,-20,30);**

**cout<<”S :”;**

**S.display();**

**-S;**//now that invokes the unary - operator overloading

**cout<<”-S :”;**

**S.display();**

**return 0;**

**}**

So, this is how it is called:

**void space::getdata(int a,int b,int c)**

**{**

**x=a;**

**y=b;**

**z=c;**

**}**

And, this is how it is invoked.

**-S;**

The member function for overloading unary operator generally don’t have any arguments.

**Overloading Binary Operator:**

**#include<iostream>**

**using namespace std;**

**class Complex**

**{**

**private:**

**float x;**

**float y;**

**public:**

**complex(){}**

**complex(float read,float i mag){x=real;y=i mag}**

**complex operator+(complex);**

**void display(void);**

**};**

**complex complex::operator+(complex c)**

**{**

**complex temp;**

**temp.x=x+c.x;**

**temp.y=y+c.y;**

**return temp;**

**}**

**void complex::display(void)**

**{**

**cout<<x<<”+j”<<y<<”\n”;**

**}**

**int main()**

**{**

**complex c1,c2,c3;**

**c1=complex(2.5,3.5);**

**c2=complex(1.6,2.7);**

**c3=c1+c2;**

//c1’s operator overloading function is going to be called with c2 as an argument

**cout<<”C1=”;c1.display();**

**cout<<”C2=”;c2.display();**

**cout<<”C3=”;c3.display();**

**return 0;**

**}**

Which object’s overloading function is going to be called is based on associativity.

Now, friend function can also be used for operator overloading:

**#include <iostream>**

**using namespace std;**

**class Complex**

**{**

**private:**

**float x;**

**float y;**

**public:**

**Complex(){x=0;y=0;}**

**Complex(float real,float i mag)**

**{**

**x=real;**

**y=i mag;**

**}**

**friend Complex operator+(Complex a,Complex b);**

**void display(void);**

**};**

**void Complex::display(void)**

**{**

**cout<<x<<"+j"<<y<<endl;**

**}**

**Complex operator+(Complex a,Complex b)**

**{**

**Complex temp;**

**temp.x=a.x+b.x;**

**temp.y=a.y+b.y;**

**return temp;**

**}**

**int main()**

**{**

**Complex c1,c2,c3;**

**c1=Complex(2.5,3.5);**

**c2=Complex(1.6,2.7);**

**c3=c1+c2;**

**//c1’s operator overloading function is going to be called with c2 as an argument**

**cout<<"C1=";c1.display();**

**cout<<"C2=";c2.display();**

**cout<<"C3=";c3.display();**

**return 0;**

**}**

**45.Overloading subscripting operator: (overloading subscripting operator cannot be possible using friend function)**

We are talking about array subscripting.

(you can not overload subscripting operator using friend)

Now, this is a non practical example. However, it is still very important.

**#include<iostream>**

**using namespace std;**

**class arr**

**{**

**private:**

**int a[5];**

**public:**

**arr(int \*s)**

**{**

**int i;**

**for(int i=0;i<5;i++)**

**{**

**a[i]=s[i];**

**}**

**}**

**int operator[](int k)**

**{**

**return (a[k]);**

**}**

**};**

**int main()**

**{**

**int x[5]={1,2,3,4,5};**

**arr A(x);**

**//parametric constructor invoked**

**int i;**

**for(int i=0;i<5;i++)**

**{**

**cout<<A[i]<<" ";**

**}**

**return 0;**

**}**

Here, we can access arr A’s private data by subscipt operator overloading.

**46.Mention some of the operator for which operator overloading cannot be done in c++.**

• Class member access operator (.)

• Scope resolution operator (::)

• Size of operator (sizeof)

• Conditional operator(?:)

**47.Mention Some Of The Operators Which can be overloaded but not with friend function: (but can be overloaded)  
  
=** Assignment operator.

**()** Function call operator.

**[]** Subscripting operator.

**->** Class member access operator.

**48.What is initializer list in c++? Mention some cases where initializer list must be used.**

Answer) Initializer list is a special feature added to c++ which is needed to initialize data members of a class. The list of members to be initialized is indicated with constructor as a comma separated list followed by a colon.

Now, initializer list is must in the following cases:

**1) 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;**

**}**

However, you can write it:  
  
**#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; }**

**Test(int);**

**};**

**Test::Test(int t):t(t)**

**{**

**}**

**int main()**

**{**

**Test t1(10);**

**cout<<t1.getT();**

**return 0;**

**}**

So, you can see, that during declaration of a function initializer list is actually not needed. (Here, the example is for non static const data members)

**2) 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;**

**}**

**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;**

**}**

But, the following will generate error:

**#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(A);**

**};**

**B::B(A tempa)**

**{**

**a=tempA;**

**cout << "B's Constructor called";**

**}**

**int main()**

**{**

**A a(10);**

**B obj(a);**

**return 0;**

**}**

This will generate compilation error.

**Note:**

Upto this point, how initializer list is used.

: variable name in class(variable name sent as argument to initialize the variable name in class)

Now, initializer list can invoke parent constructor, too.

**4)For initialization of base class members:**

However, in this case, in is not necessary.

the constructors of the base class expect some fields to be initialized then it need initializer list's help. Since, initializer list of a constructor executes before the constructor.

**#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;**

**}**

Here, it is not variable’s name. Rather it was parent class’s name.

**B::B(int x):A(x)**

**{ //Initializer list must be used**

**cout << "B's Constructor called";**

**}**

**Now, the last point is performance. However, I do not digest this example.**

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;**

**}**

**};**

Here compiler follows following steps to create an object of type MyClass

1. Type’s constructor is called first for “a”.

2. The assignment operator of “Type” is called inside body of MyClass() constructor to assign

variable = a;

1. And then finally destructor of “Type” is called for “a” since it goes out of scope.

Now consider the same code with MyClass() constructor with Initializer List

**// With Initializer List**

**class MyClass**

**{**

**Type variable;**

**public:**

**MyClass(Type a):variable(a)**

**{**

**// Assume that Type is an already**

**// declared class and it has appropriate**

**// constructors and operators**

**}**

**};**

With the Initializer List, following steps are followed by compiler:

1. Copy constructor of “Type” class is called to initialize : variable(a). The arguments in initializer

list are used to copy construct “variable” directly.

2. Destructor of “Type” is called for “a” since it goes out of scope.

**So, that means, if a class contains object from another class as a variable/data member, it is always better to use initializer list.**

**#include<bits/stdc++.h>**

**using namespace std;**

**class Window**

**{**

**private:**

**int length;**

**int width;**

**string frame\_color;**

**public:**

**Window()**

**{**

**cout<<"The default constructor of Window called"<<endl;**

**length=0;**

**width=0;**

**frame\_color="";**

**}**

**Window(int length,int width,string frame\_color)**

**{**

**cout<<"The parametric constructor of window called"<<endl;**

**this->length=length;**

**//for every non static member, this pointer is a hidden part**

**this->width=width;**

**this->frame\_color=frame\_color;**

**}**

**Window(const Window &temp\_window)**

**{**

**cout<<"The copy constructor of the window is called"<<endl;**

**if(this==&temp\_window)**

**{**

**//self reference check;**

**return;**

**}**

**length=temp\_window.length;**

**width=temp\_window.width;**

**frame\_color=frame\_color;**

**}**

**Window operator=(const Window &temp\_window)**

**{**

**cout<<"The assignment operator of the window is called"<<endl;**

**if(this==&temp\_window)**

**{**

**//self reference check;**

**return \*this;**

**}**

**length=temp\_window.length;**

**width=temp\_window.width;**

**frame\_color=frame\_color;**

**}**

**void display()const**

**{**

**cout<<"The length of the window is:"<<length<<endl;**

**cout<<"The width of the window is:"<<width<<endl;**

**cout<<"The frame\_color of the window is:"<<frame\_color<<endl;**

**}**

**~Window()**

**{**

**cout<<"The destructor of window is called"<<endl;**

**}**

**};**

**class Room**

**{**

**private:**

**Window first\_window;**

**string color;**

**public:**

**Room(Window temp\_window,string color): first\_window(temp\_window)**

**{**

**cout<<"The parametric constructor of the room is called"<<endl;**

**this->color=color;**

**}**

**void display()const**

**{**

**first\_window.display();**

**cout<<"The color of the room is: "<<color<<endl;**

**}**

**~Room()**

**{**

**cout<<"the destructor of the room is called"<<endl;**

**}**

**};**

**int main()**

**{**

**Window window\_design(5,3,"rose wood");**

**Room curr\_room(window\_design,"green");**

**curr\_room.display();**

**return 0;**

**}**

Check this example. Now, if you check the statements printed by program:

The parametric constructor of window called **(for window window\_design(5,3,"rose wood"))**

The copy constructor of the window is called **(Now, window\_design is initialized to first\_window variable of room via copy constructor)**

The copy constructor of the window is called

**(two copy constructor of the window is called. Why? One for :first\_window(temp\_window) and another one for actually initialization?)**

The parametric constructor of the room is called

**(obviously, after we are don with parent’s constructor, it will invoke child class’s parametric constructor is called)**

The destructor of window is called (temp\_window is destroyed)

The length of the window is:5

The width of the window is:3

The frame\_color of the window is:

The color of the room is: green

the destructor of the room is called

The destructor of window is called

The destructor of window is called

**(why? Two times destructor of window is called?)**

Now, if it is slightly modified to the following:

**#include<bits/stdc++.h>**

**using namespace std;**

**class Window**

**{**

**private:**

**int length;**

**int width;**

**string frame\_color;**

**public:**

**Window()**

**{**

**cout<<"The default constructor of Window called"<<endl;**

**length=0;**

**width=0;**

**frame\_color="";**

**}**

**Window(int length,int width,string frame\_color)**

**{**

**cout<<"The parametric constructor of window called"<<endl;**

**this->length=length;**

**//for every non static member, this pointer is a hidden part**

**this->width=width;**

**this->frame\_color=frame\_color;**

**}**

**Window(const Window &temp\_window)**

**{**

**cout<<"The copy constructor of the window is called"<<endl;**

**if(this==&temp\_window)**

**{**

**//self reference check;**

**return;**

**}**

**length=temp\_window.length;**

**width=temp\_window.width;**

**frame\_color=frame\_color;**

**}**

**Window operator=(const Window &temp\_window)**

**{**

**cout<<"The assignment operator of the window is called"<<endl;**

**if(this==&temp\_window)**

**{**

**//self reference check;**

**return \*this;**

**}**

**length=temp\_window.length;**

**width=temp\_window.width;**

**frame\_color=frame\_color;**

**}**

**void display()const**

**{**

**cout<<"The length of the window is:"<<length<<endl;**

**cout<<"The width of the window is:"<<width<<endl;**

**cout<<"The frame\_color of the window is:"<<frame\_color<<endl;**

**}**

**~Window()**

**{**

**cout<<"The destructor of window is called"<<endl;**

**}**

**};**

**class Room**

**{**

**private:**

**Window first\_window;**

**string color;**

**public:**

**Room(const Window &window\_design,string color): first\_window(window\_design)**

**{**

**cout<<"The parametric constructor of the room is called"<<endl;**

**this->color=color;**

**}**

**void display()const**

**{**

**first\_window.display();**

**cout<<"The color of the room is: "<<color<<endl;**

**}**

**~Room()**

**{**

**cout<<"the destructor of the room is called"<<endl;**

**}**

**};**

**int main()**

**{**

**Window window\_design(5,3,"rose wood");**

**Room curr\_room(window\_design,"green");**

**curr\_room.display();**

**return 0;**

**}**

You will see, there is two less calls.

The parametric constructor of window called

The copy constructor of the window is called

The parametric constructor of the room is called

The length of the window is:5

The width of the window is:3

The frame\_color of the window is:

The color of the room is: green

the destructor of the room is called

The destructor of window is called

The destructor of window is called

**(at least, though there are bugs)**

And, if the code is changed to following, I.e. we remove the initializer list:

**#include<bits/stdc++.h>**

**using namespace std;**

**class Window**

**{**

**private:**

**int length;**

**int width;**

**string frame\_color;**

**public:**

**Window()**

**{**

**cout<<"The default constructor of Window called"<<endl;**

**length=0;**

**width=0;**

**frame\_color="";**

**}**

**Window(int length,int width,string frame\_color)**

**{**

**cout<<"The parametric constructor of window called"<<endl;**

**this->length=length;**

**//for every non static member, this pointer is a hidden part**

**this->width=width;**

**this->frame\_color=frame\_color;**

**}**

**Window(const Window &temp\_window)**

**{**

**cout<<"The copy constructor of the window is called"<<endl;**

**if(this==&temp\_window)**

**{**

**//self reference check;**

**return;**

**}**

**length=temp\_window.length;**

**width=temp\_window.width;**

**frame\_color=frame\_color;**

**}**

**Window operator=(const Window &temp\_window)**

**{**

**cout<<"The assignment operator of the window is called"<<endl;**

**if(this==&temp\_window)**

**{**

**//self reference check;**

**return \*this;**

**}**

**length=temp\_window.length;**

**width=temp\_window.width;**

**frame\_color=frame\_color;**

**}**

**void display()const**

**{**

**cout<<"The length of the window is:"<<length<<endl;**

**cout<<"The width of the window is:"<<width<<endl;**

**cout<<"The frame\_color of the window is:"<<frame\_color<<endl;**

**}**

**~Window()**

**{**

**cout<<"The destructor of window is called"<<endl;**

**}**

**};**

**class Room**

**{**

**private:**

**Window first\_window;**

**string color;**

**public:**

**Room(Window temp\_window,string color)**

**{**

**cout<<"The parametric constructor of the room is called"<<endl;**

**this->first\_window=temp\_window;**

**this->color=color;**

**}**

**void display()const**

**{**

**first\_window.display();**

**cout<<"The color of the room is: "<<color<<endl;**

**}**

**~Room()**

**{**

**cout<<"the destructor of the room is called"<<endl;**

**}**

**};**

**int main()**

**{**

**Window window\_design(5,3,"rose wood");**

**Room curr\_room(window\_design,"green");**

**curr\_room.display();**

**return 0;**

**}**

The statement printing order is:  
  
**The parametric constructor of window called (**for Window window\_design(5,3,"rose wood")

**The copy constructor of the window is called** (from window\_design to temp\_window)

**The default constructor of Window called** (this->first\_window=temp\_window; will this invoke the default constructor of first\_window first, and afterwards, overloaded assignment operator)

**The parametric constructor of the room is called (finally)**

**The assignment operator of the window is called** (yes, whatever I thought 2 lines before was actually correct)

**The destructor of window is called**

**Segmentation fault**

(not working properly. Without initializer list in this case)

Now, why there’s a segmentation fault, that we can analyze later.

You can see, that whatever stated above for which we mentioned these two programs in the current context is true.

Though, we have much more to know about initializer list:

For example, (the difference between initialization and assignment makes the difference)

The way of initializing a member array of a class:

**class Something**

**{**

**private:**

**const int m\_array[5];**

**public:**

**Something(): m\_array { 1, 2, 3, 4, 5 } // use uniform initialization to initialize our member array**

**{**

**}**

**};**

(from c++11)

Initializing Member variables that are classes:

**#include <iostream>**

**class A**

**{**

**public:**

**A(int x) { std::cout << "A " << x << "\n"; }**

**};**

**class B**

**{**

**private:**

**A m\_a;**

**public:**

**B(int y)**

**: m\_a(y-1) // call A(int) constructor to initialize member m\_a**

**{**

**std::cout << "B " << y << "\n";**

**}**

**};**

**int main()**

**{**

**B b(5);**

**return 0;**

**}**

Another special case is when the base class is abstract class and has some data members.

**49. How to use Initializer list for multiple variables?**Make them comma separated.

**Inheritance**

**50.Access Specifiers:**

There are 3 access specifiers for a class/struct/Union in C++. These access specifiers define how the members of the class can be accessed. Of course, any member of a class is accessible within that class(Inside any member function of that same class). Moving ahead to type of access specifiers, they are:

**Public** - The members declared as Public are accessible from outside the Class through an object of the class.

**Protected** - The members declared as Protected are accessible from outside the class BUT only in a class derived from it.

**Private** - These members are only accessible from within the class. No outside Access is allowed.

**An Source Code Example:**

**class MyClass**

**{**

**public:**

**int a;**

**protected:**

**int b;**

**private:**

**int c;**

**};**

**int main()**

**{**

**MyClass obj;**

**obj.a = 10; //Allowed**

**obj.b = 20; //Not Allowed, gives compiler error**

**//allowed only from derived class**

**obj.c = 30; //Not Allowed, gives compiler error**

**}**

**Inheritance and Access Specifiers**

**Inheritance and Access Specifiers**

Inheritance in C++ can be one of the following types:

Private Inheritance

Public Inheritance

Protected inheritance

Here are the member access rules with respect to each of these:

**First and most important rule Private members of a class are never accessible from anywhere except the members of the same class.**

**Public Inheritance:**

All Public members of the Base Class become Public Members of the derived class &

All Protected members of the Base Class become Protected Members of the Derived Class.

**Private Inheritance:**

All Public members of the Base Class become Private Members of the Derived class &

All Protected members of the Base Class become Private Members of the Derived Class.

**Protected Inheritance:**All Public members of the Base Class become Protected Members of the derived class &

All Protected members of the Base Class become Protected Members of the Derived Class.

**(private member of a base class cannot be inherited)**

**51.Is Friend Function Inherited?**

**#include<bits/stdc++.h>**

**using namespace std;**

**class A**

**{**

**int x;**

**friend class B;**

**};**

**class B**

**{**

**public:**

**// Now children of B can access foo**

**void foo(A& a, int n) { a.x = n; }**

**};**

**class D : public B**

**{**

**public:**

**void foo(A& a, int n)**

**{**

**B::foo(a, n + 5);**

**}**

**};**

**int main()**

**{**

**return 0;**

**}**

Consider the following code. Now, think if there exists a child class of A, will it be accessible from B (which is a friend class of A)?

Answer is **no.**

**Similarly, if there’s a friend function exists in class A, and class A has a derived class class B, the friend function cannot access class B.**

**52.Single Inheritance:**

**53.Multilevel Inheritance:**

**54.Multiple Inheritance:**

A class in c++, can inherit the attributes of two or more class.

**Class D: visibility B-1,visibility B-2..**

**{**

**//body of D**

**}**

Now, suppose, class D inherits both class A and B

And both class have a function display()

In class B, which function version of display will be inherited?

**This ambiguity is to be resolved:**

**Class D: public A, public B**

**{**

**void display()**

**{**

**A::display();**

**}**

**}**

Multiple(Hybrid) inheritance has been a sensitive issue for many years, with opponents pointing to its increased complexity and ambiguity in situations such as the **"diamond problem"**, where it may be ambiguous as to which parent class a particular feature is inherited from if more than one parent class implements said feature. This can be addressed in various ways, including using virtual inheritance. However, diamond problem is little different: instead of being related to hybrid inheritance, it is related to multiple inheritance

**#include <iostream>**

**class LivingThing {**

**protected:**

**void breathe() {**

**std::cout << "I'm breathing as a living thing." << std::endl;**

**}**

**};**

**class Animal : protected LivingThing {**

**protected:**

**void breathe() {**

**std::cout << "I'm breathing as an animal." << std::endl;**

**}**

**};**

**class Reptile : protected LivingThing {**

**protected:**

**void crawl() {**

**std::cout << "I'm crawling as a reptile." << std::endl;**

**}**

**};**

**class Snake : protected Animal, protected Reptile {**

**public:**

**void breathe() {**

**std::cout << "I'm breathing as a snake." << std::endl;**

**}**

**void crawl() {**

**std::cout << "I'm crawling as a snake." << std::endl;**

**}**

**};**

**int main() {**

**Snake snake;**

**snake.breathe();**

**snake.crawl();**

**return 0;**

**}**

Now, this will compile ok. Since, only Animal class inherits breathe. (snake class did not)

What if Reptile class overrides the breathe() method?

The Snake class would not know which breathe() method to call. This is the “Diamond Problem”.

**#include <iostream>**

**class LivingThing {**

**protected:**

**void breathe() {**

**std::cout << "I'm breathing as a living thing." << std::endl;**

**}**

**};**

**class Animal : protected LivingThing {**

**protected:**

**void breathe() {**

**std::cout << "I'm breathing as an animal." << std::endl;**

**}**

**};**

**class Reptile : protected LivingThing {**

**public:**

**void breathe() {**

**std::cout << "I'm breathing as a reptile." << std::endl;**

**}**

**void crawl() {**

**std::cout << "I'm crawling as a reptile." << std::endl;**

**}**

**};**

**class Snake : public Animal, public Reptile {**

**};**

**int main() {**

**Snake snake;**

**snake.breathe();**

**snake.crawl();**

**return 0;**

**}**

If you try compiling the program, it won’t. You’ll be staring at an error message like the one below.

member ‘breathe’ found in multiple base classes of different types

**55.Hybrid inheritance and Virtual Base Class:**

**Class A**

**{**

**….…………**

**….…………**

**};**

**Class B: virtual public A**

**{**

**….…………..**

**….………….**

**};**

**Class C: public virtual A**

**{**

**….……………..**

**….……………..**

**};**

**Class D: public B,public C**

**{**

**….……….**

**….………**

**};**

When a class is made as a virtual base class, c++ takes necessary care to see that one copy of that class is inherited, regardless of how many inheritance paths exist b/w the virtual base class and a derived class.

**Inheritance And Polymorphism**

**56.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.

A virtual function makes its class a polymorphic base class. Derived classes can override virtual functions. Virtual functions called through base class pointers/references will be resolved at run-time. That is, the dynamic type of the object is used instead of its static type:

Derived d;

Base& rb = d;

// if Base::f() is virtual and Derived overrides it, Derived::f() will be called

rb.f()

**A pure virtual function is a virtual function whose declaration ends in =0:**

**class Base**

**{**

**// ...**

**virtual void f() = 0;**

**// …**

**}**

A pure virtual function implicitly makes the class it is defined for abstract (unlike in Java where

you have a keyword to explicitly declare the class abstract). Abstract classes cannot be instantiated.

Derived classes need to override/implement all inherited pure virtual functions. If they do not, they

too will become abstract.

**There are some interesting facts about abstract class:**1) A class is abstract if it has at least one pure virtual function.

1. We can have pointers and references of abstract class type.
2. **If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.**

**57.Can A Pure virtual Function have it’s own implementation?**

A pure virtual function must be implemented in a derived type that will be directly instantiated, however the base type can still define an implementation. A derived class can explicitly call the base class implementation (if access permissions allow it) by using a fully-scoped name (by calling A::f() in your example - if A::f() were public or protected). Something like:

**class B : public A {**

**virtual void f() {**

**// class B doesn't have anything special to do for f()**

**// so we'll call A's**

**// note that A's declaration of f() would have to be public**

**// or protected to avoid a compile time problem**

**A::f();**

**}**

**};**

So, see, it cannot be inherited directly. Since, f() was a pure virtual function. However, since, A still provides an implementation, that implementation can be reused A::f().

For instance, the following code will be compiled successfully.

**#include<bits/stdc++.h>**

**using namespace std;**

**class Base**

**{**

**public:**

**virtual void f()=0;**

**};**

**void Base::f()**

**{**

**cout<<"Base class's f() is called"<<endl;**

**}**

**class Derived: public Base**

**{**

**public:**

**//class derived class name::visibility base class name**

**void f()**

**{**

**Base::f();**

**}**

**};**

**int main()**

**{**

**//Base cannot be inherited**

**Derived d;**

**d.f();**

**return 0;**

**}**

This will print,

**Base class's f() is called**

Now, the following:

**#include<bits/stdc++.h>**

**using namespace std;**

**class Base**

**{**

**public:**

**virtual void f()=0;**

**};**

**/\*void Base::f()**

**{**

**cout<<"Base class's f() is called"<<endl;**

**}\*/**

**class Derived: public Base**

**{**

**public:**

**//class derived class name::visibility base class name**

**void f()**

**{**

**Base::f();**

**}**

**};**

**int main()**

**{**

**//Base cannot be inherited**

**Derived d;**

**d.f();**

**return 0;**

**}**

Now, this will generate compilation error. Since, Derived class does not override the base class, it will become an abstract class. And, abstract class cannot be instantiated.

**58.What Will Happen If you try to instantiate an abstract class?**

Compilation error.

**59.Can A abstract class Have A Constructor? If yes, should It have one?  
  
  
Answer)** it can have a constructor. Suppose, the abstract class have some private members. Now, we have to initialized them too. Now, if we want to initialize them during derived class’s (of that abstract class’s )initialization. We need a constructor.

I have an example, though the example is not a good one:

#include <iostream>

#include<cstdio>

using namespace std;

class p

{

protected:

int width, height;

public:

p(int width,int height)

{

this->width=width;

this->height=height;

}

virtual int area (void) = 0;

};

class r: public p

{

public:

r(int w,int h):p(w,h)

{

}

int area (void)

{

return (width \* height);

}

};

class t: public p

{

public:

t(int w,int h):p(w,h)

{

}

int area (void)

{

return (width \* height / 2);

}

};

int main ()

{

int ch;

for(int i=0;i<2;i++)

{

printf("Enter your choice. 1 for rectangle, 2 for triangle\n");

scanf("%d",&ch);

p \*polygon;

switch(ch)

{

case 1:

{

printf("A new rectangle will be created\n");

//for simplicity. Otherwise, you can scan the arguments, too

polygon=new r(12,15);

printf("The rectangle's area is: %d\n",polygon->area());

break;

}

case 2:

{

printf("A new triangle will be created\n");

polygon=new t(12,15);

printf("The triangle's area is: %d\n",polygon->area());

break;

}

default:

{

printf("Invalid choice\n");

break;

}

}

}

}

**60.Pointers To Derived Class:**

We can use pointers of base class not only to base objects but also to the object of derived class. Pointers to object of base class are type compatible with pointers to object of derived class.

**(however, vice versa is not true)**B \*ctpr;

B b;

D d;

Ctpr=&b;

We can make ctpr to point to the object d as follows:

ctpr=&d;

**However, there is a problem in using ctpr to access the public members of the derived class D, using ctpr, we can access only those members which are inherited from B and not the members that originally belong to D.**

**61.Importance Of Making Functions Virtual:**

#include<bits/stdc++.h>

using namespace std;

class Base

{

public:

void display()

{

cout<<"Base class's display is called"<<endl;

}

};

class Derived: public Base

{

public:

void display()

{

cout<<"Derived class's display is called"<<endl;

}

};

int main()

{

Base \*baseObj=new Derived();

baseObj->display();

return 0;

}

Here, Though a a pointer to Base class is initialized with a Derived class object whose memory is allocated from heap, the Base class’s display function will be called. In compile time, the function definition of display() function will be searched in base class and in runtime, the static linking done in compile time exists.

**Until, you use the keyword virtual in Base class’s display function.**

#include<bits/stdc++.h>

using namespace std;

class Base

{

public:

virtual void display()

{

cout<<"Base class's display is called"<<endl;

}

};

class Derived: public Base

{

public:

void display()

{

cout<<"Derived class's display is called"<<endl;

}

};

int main()

{

Base \*baseObj=new Derived();

baseObj->display();

return 0;

}

**This will print:** Derived class's display is called

**62.Virtual Destructor:**

**class Base**

**{**

**// some virtual methods**

**};**

**class Derived : public Base**

**{**

**~Derived()**

**{**

**// Do some important cleanup**

**}**

**}**

**Here, you'll notice that I didn't declare Base's destructor to be virtual. Now, let's have a look at the**

**following snippet:**

**Base \*b = new Derived();**

**// use b**

**delete b; // Here's the problem!**

Since Base's destructor is not virtual and b is a Base\* pointing to a Derived object, delete b has undefined behaviour. In most implementations, the call to the destructor will be resolved like any non-virtual code, meaning that the destructor of the base class will be called but not the one of the derived class, resulting in resources leak.

So, you have to make the base class's destructor virtual to call the derived class constructor's mandatorily.

To sum up, always make base classes' destructors virtual when they're meant to be manipulated Polymorphically**.**

**63.Check the difference b/w following codes and their outputs:**

**#include <bits/stdc++.h>**

**using namespace std;**

**class A**

**{**

**public:**

**A()**

**{**

**cout<<"The A class's contructor is called"<<endl;**

**}**

**virtual ~A()**

**{**

**cout<<"The A class's destructor is called"<<endl;**

**}**

**};**

**class B: public A**

**{**

**public:**

**B()**

**{**

**cout<<"The B class's constructor is called"<<endl;**

**}**

**~B()**

**{**

**cout<<"The B class's destructor is called"<<endl;**

**}**

**};**

**int main()**

**{**

**A \*a\_ptr=new B();**

**delete a\_ptr;**

**return 0;**

**}**

The A class's contructor is called

The B class's constructor is called

The B class's destructor is called

The A class's destructor is called

**But, the following will generate a different output:**

**#include <bits/stdc++.h>**

**using namespace std;**

**class A**

**{**

**public:**

**A()**

**{**

**cout<<"The A class's contructor is called"<<endl;**

**}**

**~A()**

**{**

**cout<<"The A class's destructor is called"<<endl;**

**}**

**};**

**class B: public A**

**{**

**public:**

**B()**

**{**

**cout<<"The B class's constructor is called"<<endl;**

**}**

**~B()**

**{**

**cout<<"The B class's destructor is called"<<endl;**

**}**

**};**

**int main()**

**{**

**A \*a\_ptr=new B();**

**delete a\_ptr;**

**return 0;**

**}**

The A class's contructor is called

The B class's constructor is called

The A class's destructor is called

**As you can see, B’s destructor is not called.**

**64.Can Destructor be Pure Virtual?**

Yes. But, that pure virtual destructor must have a definition body.

**#include <bits/stdc++.h>**

**using namespace std;**

**class A**

**{**

**public:**

**A()**

**{**

**cout<<"The A class's contructor is called"<<endl;**

**}**

**virtual ~A()=0;**

**//this is pure virtual function**

**};**

**A::~A()**

**{**

**cout<<"The A class's destructor is called"<<endl;**

**}**

**class B: public A**

**{**

**public:**

**B()**

**{**

**cout<<"The B class's constructor is called"<<endl;**

**}**

**~B()**

**{**

**cout<<"The B class's destructor is called"<<endl;**

**}**

**};**

**int main()**

**{**

**A \*a\_ptr=new B();**

**delete a\_ptr;**

**return 0;**

**}**

**This will give output:**

The A class's contructor is called

The B class's constructor is called

The B class's destructor is called

The A class's destructor is called

**Whether the following will generate compilation error.  
  
#include <bits/stdc++.h>**

**using namespace std;**

**class A**

**{**

**public:**

**A()**

**{**

**cout<<"The A class's contructor is called"<<endl;**

**}**

**virtual ~A()=0;**

**//this is pure virtual function**

**};**

**class B: public A**

**{**

**public:**

**B()**

**{**

**cout<<"The B class's constructor is called"<<endl;**

**}**

**~B()**

**{**

**cout<<"The B class's destructor is called"<<endl;**

**}**

**};**

**int main()**

**{**

**A \*a\_ptr=new B();**

**delete a\_ptr;**

**return 0;**

**}**

**Function Templates**

**65.Function Templates:  
  
#include <bits/stdc++.h>**

**using namespace std;**

**template <typename T>**

**void swap\_val(T &x,T &y)**

**{**

**T temp=x;**

**x=y;**

**y=temp;**

**}**

**int main()**

**{**

**int a=5,b=10;**

**cout<<"Before swapping A: "<<a<<" B: "<<b<<endl;**

**swap\_val<int>(a,b);**

**cout<<"After swapping A: "<<a<<" B :"<<b<<endl;**

**return 0;**

**}**

If you pass variables of predefined datatype rather than value, you can skip <int> part in swap\_val<int>(a,b)

**#include <bits/stdc++.h>**

**using namespace std;**

**template <typename T>**

**void swap\_val(T &x,T &y)**

**{**

**T temp=x;**

**x=y;**

**y=temp;**

**}**

**int main()**

**{**

**float a=5,b=10;**

**cout<<"Before swapping A: "<<a<<" B: "<<b<<endl;**

**swap\_val(a,b);**

**cout<<"After swapping A: "<<a<<" B :"<<b<<endl;**

**return 0;**

**}**

However, if you pass values rather than variables and the values are not of int type, generally it is good to mention the datatype.

**66.Function Template With Multiple Parameters:  
  
template<class T1,class T2>**

**returntype functionname(args of type t1,type t2,…)**

**{**

**….……….**

**….………(body of function)**

**….……….**

**}**

**Overloading of Template functions:**you can.

**Non Type Template Arguments:**

(this approach is very helpful for creation of stack which is an array underneath)

**Template<class T,int size>**

**Class Array**

**{**

**T a[size];**

**};  
array<int,10> a1;**

**array<float,5> a2;**

**array<char,20> a3;**

Here, array size is passed as an argument to the template class.

Note, how is it passed?

**67.Overloading Function Templates**

template <class T>

inline T square(T x)

{

T result;

result = x \* x;

return result;

};

template <>

**//generalization**

string square<string>(string ss)

{

return (ss+ss);

};

This is how you can overload function templates. Now, this is also called specialization. We will come back to this later.

**68.Class Templates:**

**template<class T>**

**class classname**

**{**

**//class member specificiation**

**//with anonymous type t**

**//whereever appropiate**

**};**

An example:

**#include<bits/stdc++.h>**

**using namespace std;**

**template<class T>**

**class Myvector**

**{**

**private:**

**T \*v;**

**int size;**

**public:**

**Myvector(int m)**

**{**

**v=new T[size=m];**

**for(int i=0;i<size;i++)**

**{**

**v[i]=0;**

**}**

**}**

**Myvector(T \*a)**

**{**

**for(int i=0;i<size;i++)**

**{**

**v[i]=a[i];**

**}**

**}**

**T operator\*(Myvector &y)**

**{**

**T sum=0;**

**for(int i=0;i<size;i++)**

**{**

**sum+=this->v[i]\*y.v[i];**

**}**

**return sum;**

**}**

**};**

**int main()**

**{**

**Myvector<int> v1(10);**

**return 0;**

**}**

Or,

**#include<bits/stdc++.h>**

**using namespace std;**

**template < class T > class Myvector**

**{**

**private:**

**T \* v;**

**int size;**

**public:**

**Myvector (int);**

**Myvector (T \*);**

**T operator\* (Myvector &);**

**};**

**template < class T > Myvector < T >::Myvector (int m)**

**{**

**v = new T[size = m];**

**for (int i = 0; i < size; i++)**

**{**

**v[i] = 0;**

**}**

**}**

**template < class T > Myvector < T >::Myvector (T \* a)**

**{**

**for (int i = 0; i < size; i++)**

**{**

**v[i] = a[i];**

**}**

**}**

**template<class T>**

**T Myvector < T >::operator\* (Myvector & y)**

**{**

**T sum = 0;**

**for (int i = 0; i < size; i++)**

**{**

**sum += this->v[i] \* y.v[i];**

**}**

**return sum;**

**}**

**int main ()**

**{**

**Myvector < int >v1 (10);**

**return 0;**

**}**

**69.Can you overload template class?**No.  
  
**template<class T>  
class String**

**{**

**};**

**class String**

**{**

**/\***

**\*/**

**};**

**//error: double definition**

**70.Default Template Parameters:**

#include <iostream>

using namespace std;

template <typename T = float, int count = 3>

T multIt(T x)

{

for(int ii = 0; ii < count; ii++)

{

x = x \* x;

}

return x;

};

int main()

{

float xx = 2.1;

cout << xx << ": " << multIt<>(xx) << endl;

}

**Now, this is one of the way of having default template parameters.**

**However, to compile it, it requires special options.**

**71. Specialization:  
  
Full specialization:**

template<class T> class Array { /\*...\*/ };

template<class T> void sort(Array<T>& v); // primary template

template<> void sort(Array<int>&); // specialization for T = int

// no need to write

// template<> void sort<int>(Array<int>&);

**Full program:**

template< typename T>

struct A {

struct B {}; // member class

template<class U> struct C { }; // member class template

};

template<> // specialization

struct A<int> {

void f(int); // member function of a specialization

};

// template<> not used for a member of a specialization

void A<int>::f(int) { /\* ... \*/ }

template<> // specialization of a member class

struct A<char>::B {

void f();

};

// template<> not used for a member of a specialized member class either

void A<char>::B::f() { /\* ... \*/ }

template<> // specialization of a member class template

template<class U> struct A<char>::C {

void f();

};

// template<> is used when defining a member of an explicitly

// specialized member class template specialized as a class template

template<>

template<class U> void A<char>::C<U>::f() { /\* ... \*/ }

**Static members of a template class and explicit specialization of it:**

**An explicit specialization of a static data member of a template is a definition if the declaration includes an initializer; otherwise, it is a declaration. These definitions must use braces for default initialization:**

template<> X Q<int>::x; // declaration of a static member

template<> X Q<int>::x (); // error: function declaration

template<> X Q<int>::x {}; // definition of a default-initialized static member

**What does it mean? Because, I am not getting it.**#include<iostream>

#include<cstdio>

using namespace std;

template <typename T>

struct S

{

public:

static double something\_relevant;

static void display\_val();

};

template <typename T>

double S<T>::something\_relevant = 1.5;

template <>

double S<char>::something\_relevant=2.5;

template<typename T>

void S<T>::display\_val()

{

printf("The value is: %g\n",something\_relevant);

}

int main()

{

S<int>::display\_val();

S<char>::display\_val();

return 0;

}

Because, this works fine for me.

This prints

**The value is: 1.5**

**The value is: 2.5**

**Partial Specialization:**

Now, there are some partial specialization examples:

**template<class T1, class T2, int I>**

**class A {}; // primary template**

**template<class T, int I>**

**class A<T, T\*, I> {}; // #1: partial specialization where T2 is a pointer to T1**

**template<class T, class T2, int I>**

**class A<T\*, T2, I> {}; // #2: partial specialization where T1 is a pointer**

**template<class T>**

**class A<int, T\*, 5> {}; // #3: partial specialization where T1 is int, I is 5,**

**// and T2 is a pointer**

**template<class X, class T, int I>**

**class A<X, T\*, I> {}; // #4: partial specialization where T2 is a pointer**

**Now, Partial Specialization has a set of rules:**

1) The argument list cannot be identical to the non-specialized argument list (it must specialize something)

**template<class T1, class T2, int I> class B {}; // primary template**

**template<class X, class Y, int N> class B<X,Y,N> {}; // error**

**1) The argument list cannot be identical to the non-specialized argument list (it must specialize something)**

template<class T1, class T2, int I> class B {}; // primary template

template<class X, class Y, int N> class B<X,Y,N> {}; // error

**Moreover, the specialization has to be more specialized than the primary template**

template<int N, typename T1, typename... Ts> struct B;

template<typename... Ts> struct B<0, Ts...> { }; // Error: not more specialized

**2)Default arguments cannot appear in the argument list.  
  
Now, this is a simple example.**

#include<iostream>

#include<cstdio>

using namespace std;

template<typename T1,typename T2,int c>

struct something\_meaningful

{

private:

T1 a;

T2 b;

public:

something\_meaningful(T1,T2);

void display();

};

template<typename T1,typename T2,int c>

something\_meaningful<T1,T2,c>::something\_meaningful(T1 tempa,T2 tempb)

{

a=tempa;

b=tempb;

}

template<typename T1,typename T2,int c>

void something\_meaningful<T1,T2,c>::display()

{

cout<<"a is: "<<a<<endl;

cout<<"b is: "<<b<<endl;

cout<<"c is: "<<c<<endl;

}

int main()

{

something\_meaningful<int, float, 10> obj(12,13.00);

obj.display();

return 0;

}

You can obviously do it. Because, here specialization concept is not even used.

3)If any argument is a pack expansion, it must be the last argument in the list

**Pack expansion:**

A variadic class template can be instantiated with any number of template arguments:

template<class ... Types> struct Tuple {};

Tuple<> t0; // Types contains no arguments

Tuple<int> t1; // Types contains one argument: int

Tuple<int, float> t2; // Types contains two arguments: int and float

Tuple<0> error; // error: 0 is not a type

A variadic function template can be called with any number of function arguments (the template arguments are deduced through template argument deduction):

template<class ... Types> void f(Types ... args);

f(); // OK: args contains no arguments

f(1); // OK: args contains one argument: int

f(2, 1.0); // OK: args contains two arguments: int and double

4) Non-type argument expression cannot use the name of the template parameter except when it is exactly the name of the template parameter (until C++14) Non-type argument expressions can use template parameters as long as the parameter appears at least once outside a non-deduced context (since C++14)

5) Non-type template argument cannot specialize a template parameter whose type depends on a parameter of the specialization.

**Exception Handling:  
  
72. Basic Exception Handling:**

#include <iostream>

#include <string>

using namespace std;

class Exception

{

public:

Exception(const string& msg) : msg\_(msg) {}

~Exception() {}

string getMessage() const {return(msg\_);}

private:

string msg\_;

};

void f()

{

throw(Exception("Mr. Sulu"));

}

int main()

{

try

{

f();

}

catch(Exception& e)

{

cout << "You threw an exception: " << e.getMessage() << endl;

}

}

**Another way of writing it:**// using standard exceptions

#include <iostream>

#include <exception>

using namespace std;

class myexception: public exception

{

virtual const char\* what() const throw()

{

return "My exception happened";

}

};

int main ()

{

myexception myex;

try

{

throw myex;

}

catch (exception& e)

{

cout << e.what() << endl;

}

return 0;

}

**What():** This class has a virtual member function called what() that returns a null-terminated character sequence (of type char \*) and that can be overwritten in derived classes to contain some sort of description of the exception.

**73.Specifying Data Type Of An Exception:  
  
 type function(arg-list) throw (type-list)**

**{**

**}**

**Type list will be comma separated.**

Consider the following example:

void test(int x)throw (int,double)

{

If(x==0) throw ‘x’;

else

if(x==1) throw x;

else

if(x==-1) throw 1.0;

cout<<”End of function block\n”;

}

**74.How many types of standard exception are there in c++?**

There are nine standard exceptions in c++. They are bad\_alloc, bad\_cast, bad\_exception, bad\_function\_call, bad\_typeid, bad\_weak\_ptr, ios\_base::failure, logic\_error and runtime\_error.

**75.Can Destructor Throw Exception?**

No.

**76.Can Constructors Throw Exception?**Yes.

**77. How is an uncaught exception handled in c++?**the default version of std::terminate() will call implicitly and after it will handle certain things, abort function will call. (not from inside terminate)

1. **Can we define our own terminate function in c++?**Yes.

void terminator()

{

cout << "terminate" << endl;

}

void (\*old\_terminate)() = set\_terminate(terminator);

1. **How to catch all exception?**The ellipses operator in catch statement.

catch(…)

**{**

**}**

**Casting Operators In c++**

**80. const\_cast**

const\_cast is used to cast away the constness of variables. Following are some interesting facts about const\_cast.

1) const\_cast can be used to change non-const class members inside a const member function. Consider the following code snippet. Inside const member function fun(), ‘this’ is treated by the compiler as ‘const student\* const this’, i.e. ‘this’ is a constant pointer to a constant object, thus compiler doesn’t allow to change the data members through ‘this’ pointer. const\_cast changes the type of ‘this’ pointer to ‘student\* const this’.

**#include <iostream>**

**using namespace std;**

**class student**

**{**

**private:**

**int roll;**

**public:**

**// constructor**

**student(int r):roll(r) {}**

**// A const function that changes roll with the help of const\_cast**

**void fun() const**

**{**

**( const\_cast <student\*> (this) )->roll = 5;**

**}**

**int getRoll() { return roll; }**

**};**

**int main(void)**

**{**

**student s(3);**

**cout << "Old roll number: " << s.getRoll() << endl;**

**s.fun();**

**cout << "New roll number: " << s.getRoll() << endl;**

**return 0;**

**}**

Now, value can only be changed for the variable, for which the constantness is taken.

For instance, the following code is ok

**#include<bits/stdc++.h>**

**using namespace std;**

**class student**

**{**

**private:**

**int roll;**

**string name;**

**public:**

**// constructor**

**student(int r,string n):roll(r),name(n){}**

**// A const function that changes roll with the help of const\_cast**

**void fun() const**

**{**

**( const\_cast <student\*> (this) )->roll = 5;**

**//name="Suman";**

**}**

**int getRoll() { return roll; }**

**void display()**

**{**

**cout<<"Student info:"<<endl;**

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

**cout<<"Roll: "<<roll<<endl;**

**}**

**};**

**int main(void)**

**{**

**student s(3,"Sourav");**

**s.display();**

**s.fun();**

**s.display();**

**}**

But, the following is not.

**#include<bits/stdc++.h>**

**using namespace std;**

**class student**

**{**

**private:**

**int roll;**

**string name;**

**public:**

**// constructor**

**student(int r,string n):roll(r),name(n){}**

**// A const function that changes roll with the help of const\_cast**

**void fun() const**

**{**

**( const\_cast <student\*> (this) )->roll = 5;**

**name="Suman";**

**}**

**int getRoll() { return roll; }**

**void display()**

**{**

**cout<<"Student info:"<<endl;**

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

**cout<<"Roll: "<<roll<<endl;**

**}**

**};**

**int main(void)**

**{**

**student s(3,"Sourav");**

**s.display();**

**s.fun();**

**s.display();**

**}**

2) const\_cast can be used to pass const data to a function that doesn’t receive const. For example, in the following program fun() receives a normal pointer, but a pointer to a const can be passed with the help of const\_cast.

For instance, the following function fun does not receive const.

So, from main, when we send a pointer to the function fun(), we first declared a const pointer in the main. Now, it cannot be passed directly to the fun. Now, we declare another pointer and create an alias with the first pointer by using const\_cast.

Now, that temporarily takes the const\_cast away.

Now, we passed that to the function. However, it’s constantness remain. Because, try the following:

It is undefined behavior to modify a value which is initially declared as const. Consider the following program. The output of the program is undefined. The variable ‘val’ is a const variable and the call ‘fun(ptr1)’ tries to modify ‘val’ using const\_cast.

**#include <iostream>**

**using namespace std;**

**int fun(int\* ptr)**

**{**

**\*ptr = \*ptr + 10;**

**return (\*ptr);**

**}**

**int main(void)**

**{**

**const int val = 10;**

**const int \*ptr = &val;**

**int \*ptr1 = const\_cast <int \*>(ptr);**

**fun(ptr1);**

**cout << val;**

**return 0;**

**}**

It will generate the result:

**The value is 20**

**10**

**(but, the upper instruction says it will be compiler depended)**

**#include <iostream>**

**using namespace std;**

**int fun(int\* ptr)**

**{**

**return (\*ptr + 10);**

**}**

**int main(void)**

**{**

**const int val = 10;**

**const int \*ptr = &val;**

**int \*ptr1 = const\_cast <int \*>(ptr);**

**cout << fun(ptr1);**

**return 0;**

**}**

Now, you can return \*ptr+10 but what you cannot do is \*ptr=\*ptr+10 (but, I am doing it)

(since, const cast make it const int \* ptr so, value contained by the memory location pointed by it cannot be changed)

3)const\_cast is considered safer than simple type casting. It’safer in the sense that the casting won’t happen if the type of cast is not same as original object. For example, the following program fails in compilation because ‘int \*’ is being typecasted to ‘char \*.  
  
**what does that mean?**

**#include<bits/stdc++.h>**

**using namespace std;**

**int main()**

**{**

**int a=40;**

**int \*ptr\_a=&a;**

**char \*ptr\_b=const\_cast<char\*>(ptr\_a);**

**return 0;**

**}**

Now, it will generate compilation error.

**main.cpp:7:40: error: invalid const\_cast from type 'int\*' to type 'char\*'**

Now, check the following:

**#include<bits/stdc++.h>**

**using namespace std;**

**int main()**

**{**

**int a=40;**

**int \*ptr\_a=&a;**

**char \*ptr\_b=(char\*)ptr\_a;**

**return 0;**

**}**

This is completely allowed. No compilation error. Whatever happens in runtime is not compiler’s concern.

4) const\_cast can also be used to cast away volatile attribute. For example, in the following program, the typeid of b1 is PVKi (pointer to a volatile and constant integer) and typeid of c1 is Pi (Pointer to integer)

**#include <iostream>**

**#include <typeinfo>**

**using namespace std;**

**int main(void)**

**{**

**int a1 = 40;**

**const volatile int\* b1 = &a1;**

**cout << "typeid of b1 " << typeid(b1).name() << '\n';**

**int\* c1 = const\_cast <int \*> (b1);**

**cout << "typeid of c1 " << typeid(c1).name() << '\n';**

**return 0;**

**}**

Now, this will generate the following result:

**typeid of b1 PVKi** (typeid shows the type: P: pointer V: volatile, K: constant, I: integer)

**typeid of c1 Pi**

Now, here, in this example, you should notice one new thing:

**const volatile int\* b1 = &a1;**

Now, const and volatile both are type qualifier.

**Now, c’s volatile type qualifier concept is something like:** C's volatile keyword is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time--**without any action being taken by the code the compiler finds nearby. (so even the currently executing code does not change it, it could be changed)**

By declaring a variable volatile, you're effectively asking the compiler to be as inefficient as possible when it comes to reading or writing that variable. **Specifically, the compiler should generate object code to perform each and every read from a volatile variable as well as each and every write to a volatile variable--even if you write it twice in a row or read it and ignore the result. Not a single read or write can be skipped. In other words, no compiler optimizations are allowed with respect to volatile variables.**

The use of volatile variables may also create additional sequence points within the functions that access them. I**n a nutshell, the order of accesses of volatile variables A and B in the object code must be the same as the order of those accesses in the source code. The compiler is not allowed to reorder volatile variable accesses for any reason.** (Consider what might go wrong if the referenced memory locations were hardware registers.)

**Now, in which cases, it will be combined?**

This is possible and mostly used in embedded system.The example is Interrupt Status Register as it is a status register , in the program we should not modify this Variable so it should be a constant.But this variable can be changed by the processor or hardware based on the interrupt condition. So when in the program ,we want to read the value of this variable , it should read the actual value with out any optimization.For this reason ,the variable can be declared as volatile too.

(so, current program cannot modify it. Hence, const. But, interrupt routine can change it. (imagine the scenario in a multicore CPU. So, when in the program, we want to read the value of this variable, it should read the actual value of this variable. Hence, volatile too)

Const qualifier makes sure the variable declared isn’t changed by the code, say assignment or something similar. This doesn’t restrict the variable from being modified external to the code. Take a peripheral’s status register as an example. If we don’t want the user to modify the contents of the register (Read-only mode), it is made const. Now if there is a chance for this register to be updated outside the code, the variable shouldn’t be cached. Make it volatile.

**Now, const\_cast in a nutshell:**

1) const\_cast can be used to change non-const class members inside a const member function. Consider the following code snippet. Inside const member function fun(), ‘this’ is treated by the compiler as ‘const student\* const this’, i.e. ‘this’ is a constant pointer to a constant object, thus compiler doesn’t allow to change the data members through ‘this’ pointer. const\_cast changes the type of ‘this’ pointer to ‘student\* const this’. I.e. it takes the constantness away from a variable.

1. const\_cast can be used to pass const data to a function that doesn’t receive const. For example, in the following program fun() receives a normal pointer, but a pointer to a const can be passed with the help of const\_cast.
2. const\_cast is considered safer than simple type casting. It’safer in the sense that the casting won’t happen if the type of cast is not same as original object. For example, the following program fails in compilation because ‘int \*’ is being typecasted to ‘char \*
3. It casts away the volatileness of a value.

**81.Static cast:**

The type parameter must be a data type to which object can be converted via a known method, whether it be a builtin or a cast. The type can be a reference or an enumerator. All types of conversions that are well-defined and allowed by the compiler are performed using static\_cast.

The static\_cast operator can be used for operations such as:

Converting a pointer of a base class to a pointer of a nonvirtual derived class, Converting numeric data types such as enums to ints or ints to floats. Although static\_cast conversions are checked at compile time to prevent obvious incompatibilities, **no run-time type check is performed that would prevent a cast between incompatible data types, such as pointers. Also, the result of a static\_cast from a pointer of a virtual base class to a pointer of a derived class is undefined.**

Now, static cast can be used for upcasting. But, upcasting is also supported by default. Static cast also support downcasting:

#include<bits/stdc++.h>

using namespace std;

class Base

{

public:

void display()

{

cout<<"Base class's display function called"<<endl;

}

};

class Derived:public Base

{

public:

void display()

{

cout<<"Derived class's display function called"<<endl;

}

};

int main()

{

Base \*ptrToBaseObj=new Base;

Derived \*ptrToDerivedObj=static\_cast<Derived\*>(ptrToBaseObj);

ptrToDerivedObj->display();

return 0;

}

This will compile fine and print Derived class’s display function called.

This static casting will help invoking the base class specific methods.

static\_cast, aside from manipulating pointers to classes, can also be used to perform conversions explicitly defined in classes, as well as to perform standard conversions between fundamental types:

double d=3.14159265;

int i = static\_cast<int>(d);

**82.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, is granted 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, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

**class A {};**

**class B {};**

**A \* a = new A;**

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

**83.Dynamic\_cast:**

dynamic\_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

Therefore, dynamic\_cast is always successful when we cast a class to one of its base classes:

**class CBase { };**

**class CDerived: public CBase { };**

**CBase b; CBase\* pb;**

**CDerived d; CDerived\* pd;**

**pb = dynamic\_cast<CBase\*>(&d); // ok: derived-to-base**

**pd = dynamic\_cast<CDerived\*>(&b); // wrong: base-to-derived**

The second conversion in this piece of code would produce a compilation error **since base-to-derived conversions are not allowed with dynamic\_cast unless the base class is polymorphic.**

When a class is polymorphic, dynamic\_cast performs a special checking during runtime to ensure that the expression yields a valid complete object of the requested class:

**// dynamic\_cast**

**#include <iostream>**

**#include <exception>**

**using namespace std;**

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

**class CDerived: public CBase { int a; };**

**int main () {**

**try {**

**CBase \* pba = new CDerived;**

**CBase \* pbb = new CBase;**

//this is allowed. Since, CBase is not an abstract class. As, it does not hold any pure virtual method

**CDerived \* pd;**

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

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

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

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

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

**return 0;**

**}**

Now, there many things I don’t understand. I clearly understand const\_cast, but I don’t understand the difference between static cast, dynamic cast and reinterpret\_cast.

One is reinterpret\_cast allows any class’s object into any class’s object. Whereas, static cast allows some fundamental type casting as well as casting base class’s object to derived class pointer. However, in static cast, we have a problem casting base class’s object to derived class pointer if the base class is virtual because, this will give undefined behaviour. It also does not have run time checking to ensure that type cast does not happen between incompatible type, whereas, the dynamic cast checks it in runtime.

If new\_type is a pointer or reference to some class D and the type of expression is a pointer or reference to its non-virtual base B, static\_cast performs a downcast. Such static\_cast makes no runtime checks to ensure that the object's runtime type is actually D, and may only be used safely if this precondition is guaranteed by other means, such as when implementing static polymorphism. Safe downcast may be done with dynamic\_cast.

Now, some example:

**#include <bits/stdc++.h>**

**using namespace std;**

**class A**

**{**

**int var;**

**public:**

**A()**

**{**

**cout<<"The default constructor of A is called"<<endl;**

**var=0;**

**}**

**A(int var)**

**{**

**cout<<"The parametric constructor of A is called"<<endl;**

**this->var=var;**

**}**

**void display()**

**{**

**cout<<"A class's display is called"<<endl;**

**cout<<"The var is: "<<var<<endl;**

**}**

**~A()**

**{**

**cout<<"The destructor of A is called"<<endl;**

**}**

**};**

**class B**

**{**

**int var;**

**public:**

**B()**

**{**

**cout<<"The default constructor of B is called"<<endl;**

**}**

**B(int var)**

**{**

**cout<<"The parametric constructor of B is called"<<endl;**

**this->var=var;**

**}**

**void display()**

**{**

**cout<<"B class's display is called"<<endl;**

**cout<<"The var is: "<<var<<endl;**

**}**

**~B()**

**{**

**cout<<"The B class's destructor is called"<<endl;**

**}**

**};**

**int main()**

**{**

**A \*aObj=new A(10);**

**B \*bObj=static\_cast<B\*>(aObj);**

**bObj->display();**

**delete bObj;**

**return 0;**

**}**

**This will not be compiled. Since, A and B does not belong to same inheritance tree.**

main.cpp: In function 'int main()':

main.cpp:53:33: error: invalid static\_cast from type 'A\*' to type 'B\*'

B \*bObj=static\_cast<B\*>(aObj);

**Whether A is a parent class of B is checked during compile time.**

(This can be achieved using reinterpret\_cast)

**#include <bits/stdc++.h>**

**using namespace std;**

**class A**

**{**

**int var;**

**public:**

**A()**

**{**

**cout<<"The default constructor of A is called"<<endl;**

**var=0;**

**}**

**A(int var)**

**{**

**cout<<"The parametric constructor of A is called"<<endl;**

**this->var=var;**

**}**

**void display()**

**{**

**cout<<"A class's display is called"<<endl;**

**cout<<"The var is: "<<var<<endl;**

**}**

**~A()**

**{**

**cout<<"The destructor of A is called"<<endl;**

**}**

**};**

**class B**

**{**

**int var;**

**public:**

**B()**

**{**

**cout<<"The default constructor of B is called"<<endl;**

**}**

**B(int var)**

**{**

**cout<<"The parametric constructor of B is called"<<endl;**

**this->var=var;**

**}**

**void display()**

**{**

**cout<<"B class's display is called"<<endl;**

**cout<<"The var is: "<<var<<endl;**

**}**

**~B()**

**{**

**cout<<"The B class's destructor is called"<<endl;**

**}**

**};**

**int main()**

**{**

**A \*aObj=new A(10);**

**B \*bObj=reinterpret\_cast<B\*>(aObj);**

**bObj->display();**

**delete bObj;**

**return 0;**

**}**

This will be compiled.

**The parametric constructor of A is called**

**B class's display is called**

**The var is: 10**

**The B class's destructor is called**

Because, reinterpret\_cost does not check compatibility. This will just check if B class’s object is big enough to hold A’s object. No, compile time checking is done.

Now, this is simple typecasting:

**#include <bits/stdc++.h>**

**using namespace std;**

**class A**

**{**

**int var;**

**public:**

**A()**

**{**

**cout<<"The default constructor of A is called"<<endl;**

**var=0;**

**}**

**A(int var)**

**{**

**cout<<"The parametric constructor of A is called"<<endl;**

**this->var=var;**

**}**

**void display()**

**{**

**cout<<"A class's display is called"<<endl;**

**cout<<"The var is: "<<var<<endl;**

**}**

**~A()**

**{**

**cout<<"The destructor of A is called"<<endl;**

**}**

**};**

**class B**

**{**

**int var;**

**public:**

**B()**

**{**

**cout<<"The default constructor of B is called"<<endl;**

**}**

**B(int var)**

**{**

**cout<<"The parametric constructor of B is called"<<endl;**

**this->var=var;**

**}**

**void display()**

**{**

**cout<<"B class's display is called"<<endl;**

**cout<<"The var is: "<<var<<endl;**

**}**

**~B()**

**{**

**cout<<"The B class's destructor is called"<<endl;**

**}**

**};**

**int main()**

**{**

**A \*aObj=new A(10);**

**B \*bObj=(B\*)aObj;**

**bObj->display();**

**delete bObj;**

**return 0;**

**}**

This will also compile. And generate the same result:

**The parametric constructor of A is called**

**B class's display is called**

**The var is: 10**

**The B class's destructor is called**

**#include <bits/stdc++.h>**

**using namespace std;**

**class A**

**{**

**int var;**

**public:**

**A()**

**{**

**cout<<"The default constructor of A is called"<<endl;**

**var=0;**

**}**

**A(int var)**

**{**

**cout<<"The parametric constructor of A is called"<<endl;**

**this->var=var;**

**}**

**virtual void display()=0;**

**~A()**

**{**

**cout<<"The destructor of A is called"<<endl;**

**}**

**};**

**class B: public A**

**{**

**int var;**

**public:**

**B()**

**{**

**cout<<"The default constructor of B is called"<<endl;**

**}**

**B(int var)**

**{**

**cout<<"The parametric constructor of B is called"<<endl;**

**this->var=var;**

**}**

**void display()**

**{**

**cout<<"B class's display is called"<<endl;**

**cout<<"The var is: "<<var<<endl;**

**}**

**~B()**

**{**

**cout<<"The B class's destructor is called"<<endl;**

**}**

**};**

**int main()**

**{**

**A \*aObj=new A(10);**

**B \*bObj=static\_cast<B\*>(aObj);**

**bObj->display();**

**delete bObj;**

**return 0;**

**}**

Now, this will generate compilation error no matter which casting are you trying. Because, A is a pure virtual class.

**#include <bits/stdc++.h>**

**using namespace std;**

**class A**

**{**

**int var;**

**public:**

**A()**

**{**

**cout<<"The default constructor of A is called"<<endl;**

**var=0;**

**}**

**A(int var)**

**{**

**cout<<"The parametric constructor of A is called"<<endl;**

**this->var=var;**

**}**

**virtual void display(){};**

**~A()**

**{**

**cout<<"The destructor of A is called"<<endl;**

**}**

**};**

**class B: public A**

**{**

**int var;**

**public:**

**B()**

**{**

**cout<<"The default constructor of B is called"<<endl;**

**}**

**B(int var)**

**{**

**cout<<"The parametric constructor of B is called"<<endl;**

**this->var=var;**

**}**

**void display()**

**{**

**cout<<"B class's display is called"<<endl;**

**cout<<"The var is: "<<var<<endl;**

**}**

**~B()**

**{**

**cout<<"The B class's destructor is called"<<endl;**

**}**

**};**

**int main()**

**{**

**A \*aObj=new A(10);**

**B \*bObj=static\_cast<B\*>(aObj);**

**bObj->display();**

**delete bObj;**

**return 0;**

**}**

This produce unpredictable behaviour.

Now, check the output:

**The parametric constructor of A is called**

**The B class's destructor is called**

**The destructor of A is called**

As you can understand, A class’s display is called. Though B, the derived class overrides the method.

#include <bits/stdc++.h>

using namespace std;

class A

{

int var;

public:

A()

{

cout<<"The default constructor of A is called"<<endl;

var=0;

}

A(int var)

{

cout<<"The parametric constructor of A is called"<<endl;

this->var=var;

}

virtual void display(){};

~A()

{

cout<<"The destructor of A is called"<<endl;

}

};

class B: public A

{

int var;

public:

B()

{

cout<<"The default constructor of B is called"<<endl;

}

B(int var)

{

cout<<"The parametric constructor of B is called"<<endl;

this->var=var;

}

void display()

{

cout<<"B class's display is called"<<endl;

cout<<"The var is: "<<var<<endl;

}

~B()

{

cout<<"The B class's destructor is called"<<endl;

}

};

int main()

{

A \*aObj=new A(10);

B \*bObj=dynamic\_cast<B\*>(aObj);

if(bObj==NULL)

{

cout<<"This cast cannot be possible"<<endl;

}

else

{

bObj->display();

delete bObj;

}

return 0;

}

This will print, This cast cannot be possible.

**File Manipulation:  
  
C Style:**

**84.Opening and closing a file:  
  
f\_read = fopen("/home/me/data.txt", "r");**

**if (!f\_read)**

**{ /\* open operation failed. \*/**

**perror("Failed opening file '/home/me/data.txt' for reading:");**

**exit(1);**

**}**

If file is not successfully opened, then fopen returns a NULL pointer.

Now, several file opening modes are there.

**85.File Opening Mode:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Mode** | **Meaning** | **fopen Returns if FILE-** | |
| **Exists** | **Not Exists** |
| r | Reading | – | NULL |
| w | Writing | Over write on Existing | Create New File |
| a | Append | – | Create New File |
| r+ | Reading + Writing | New data is written at the beginning overwriting existing data | Create New File |
| w+ | Reading + Writing | Over write on Existing | Create New File |
| a+ | Reading + Appending | New data is appended at the end of file | Create New File |

**More modified version of this list including modes for binary operators:**

|  |  |
| --- | --- |
| **File Modes** | |
| r | Open a text file for reading. |
| w | Create a text file for writing. If the file exists, it is overwritten. |
| a | Open a text file in append mode. Text is added to the end of the file. |
| rb | Open a binary file for reading. |
| wb | Create a binary file for writing. If the file exists, it is overwritten. |
| ab | Open a binary file in append mode. Data is added to the end of the file. |
| r+ | Open a text file for reading and writing. |
| w+ | Create a text file for reading and writing. If the file exists, it is overwritten. |
| a+ | Open a text file for reading and writing at the end. |
| r+b or rb+ | Open binary file for reading and writing. |
| w+b or wb+ | Create a binary file for reading and writing. If the file exists, it is overwritten. |
| a+b or ab+ | Open a text file for reading and writing at the end. |

**86.Closing the file:**

**if (!fclose(f\_readwrite))**

**{**

**perror("Failed closing file '/usr/local/lib/db/users':");**

**exit(1);**

**}**

The fclose function returns zero if successful or EOF if an error was encountered.

**87.What is EOF?**

In computing, end-of-file (commonly abbreviated EOF[1]) is a condition in a computer operating system where no more data can be read from a data source. The data source is usually called a file or stream.

**If there is EOF, why is it not sufficient to understand that end of file is reached?**

Because, many file associated functions even return EOF upon failing. That is why feof checking is also important.

**if (feof(fp))**

**{**

**printf("\n End of file reached.");**

**}**

//because, if EOF is reached, a flag is set. Feof just checks that flag status.

**80.Getc Function:**

In the C Programming Language, the getc function reads a character from the stream pointed to by stream.

**int getc(FILE \*stream);**

The getc function returns the character read. If an error occurs, the getc function will set the stream's error indicator and return EOF. If the getc function encounters the end of stream, it will set the stream's end-of-file indicator and return EOF.

**88.Putc function:**

**int putc ( int character, FILE \* stream );**

**character**

Character to be written. The character is passed as its int promotion.

**stream**

Pointer to a FILE object that identifies the stream where the character is to be written.

If there are no errors, the same character that has been written is returned. If an error occurs, EOF is returned and the error indicator is set.

**89.Fgets function:**

fgets(line, maxline, \*fp)

It gets a character array null terminated whose length is maxline.

It follow some parameter such as Maximum length, buffer, input device reference.

It is safe to use because it checks the array bound.

It keep on reading until new line character encountered or maximum limit of character array.

Example : Let’s say the maximum number of characters are 15 and input length is greater than 15 but still fgets() will read only 15 character and print it.

**(unlike, gets which is not safe because, it does not check upper bound, fgets checks upper bound)**

**90.Fputs:**

**int fputs(const char \*str, FILE \*stream)**

str − This is an array containing the null-terminated sequence of characters to be written.

stream − This is the pointer to a FILE object that identifies the stream where the string is to be written.

**C++ File Handling:**

**91.Basics:**

**#include<iostream>**

**#include<fstream>**

**using namespace std;**

**int main()**

**{**

**ofstream outf("ITEM");**

**//does not ofstream's constructor throw any exception**

**cout<<"Enter item name:";**

**string name;**

**cin>>name;**

**outf<<name<<"\n";**

**cout<<"Enter item cost:";**

**float cost;**

**cin>>cost;**

**outf<<cost<<"\n";**

**outf.close();**

**/\*If the operation fails (including if no file was open before the call),**

**the failbit state flag is set for the stream (which may throw ios\_base::**

**failure if that state flag was registered using member exceptions).**

**\*/**

**/\*However, here, we are not throwing exceptions**

**\*/**

**ifstream inf("ITEM");**

**inf>>name;**

**inf>>cost;**

**cout<<"\n";**

**cout<<"Item name: "<<name<<endl;**

**cout<<"Item cost: "<<cost<<endl;**

**inf.close();**

**return 0;**

**}**

Now, this do things as expected.

Now, we can read things like cin, if stream is null terminated.

just replace, cin with the input file stream associated with current file.

Similarly, it’s very easy to write in the file. Just replace the cout with output file stream associated with file pointer.

Howeveer, there are other functions.

Like, getline.

One example of reading lines from a input file stream associated with some file is the following:

**while(fin)**

**{**

**fin.getline(line,N);**

**cout<<line;**

**}**

//if fin is the ifstream object which is associated with the file for reading

**85.Detecting End Of File:**

One is while(fin)

**If(fin.eof()!=0)**

**{**

**exit(1);**

**}**

//because, fin.eof() is not 0, that file EOF (end of File marker) is reached for current file.

**92.File Mode Parameters:**ios\_base::app Seek to end-of-file before each write.

ios\_base::ate Seek to end-of-file immediately after opening the file, if it

exists.

ios\_base::binary Open file in binary mode (alternative is text mode).

ios\_base::in Open file for input (implied for istream).

ios\_base::out Open file for output (implied for ostream).

ios\_base::trunc Truncate file, if it exists (default for ostream).

There are some special modes as well.

Ios::nocreate: open fails if file does not exist.

ios::noreplace: open fails if file already exists.

**93.The Differences B/w Certain modes:**

Both ios::app and ios::ate take us to the end of file when it is opened. The difference b/w the two parameters is that the ios::app allows us to add data to the end of the file. While ios::ate mode permits us to add data or modify data anywhere in the file. In both the cases, a file is created by the specified file name, if the file does not exist.

Opening a file is ios::out mode also opens it it ios::trunc mode by default.

The mode ios::app can be used only with the files capable of output.

Creating a stream using ifstream implies input, and creating a stream using ofstream implies output, So, in these cases, it is not necessary to provide the mode parameters.

The two modes can be used combinedly.

fout.open(“data”,ios::app|ios::nocreate);

**Functions For Manipulation Of File Pointers:**

Seekg(): Moves get pointer(input) to a specified location.

Seekp(): moves put pointer (output) to a specified location.

Tellg(): Gives the current position of the get pointer.

Tellp(): gives the current position to the put pointer.

**Note:** in the seekg(), the g stands for get function and in seekp, the p stands for put function.

**94.Sequential Input And Output Operations:**

**Put() and get() functions.**

This is to read and write characters.

(Fin.getline(str,N); (N is the maximum number of characters string can contain)

Is for line reading)

**#include<iostream>**

**#include<fstream>**

**#include<string>**

**using namespace std;**

**int main()**

**{**

**string str;**

**cout<<"Enter a string:";**

**cin>>str;**

**int len=str.size();**

**fstream file;**

**cout<<"Opening the \'TEXT\' file and storing the string in it"<<endl;**

**file.open("TEXT",ios::in|ios::out);**

**//now, note that file is a filestream neither an input filestream nor an output**

**//filestream**

**for(int i=0;i<len;i++)**

**{**

**file.put(str[i]);**

**}**

**file.seekg(0);**

**//move to the starting position for get i.e. we only change the read head**

**char ch;**

**cout<<"Reading the file contents: "<<endl;**

**while(file)**

**{**

**file.get(ch);**

**cout<<"Hi:"<<ch<<endl;**

**}**

**cout<<endl;**

**return 0;**

**}**

**95.Binary Operations for sequencing file read and write:**

Infile.read((char \*) &v,sizeof(v));

outfile.write((char \*)&v,sizeof(v));

(this two are important)

These functions.

**Implementation Of Classes And Functions:**

**96.How to implement your own vector class?**

**Keypoints:**

Template parameter.

Array subscripting operator overloading.  
(Now, array subscripting operator is one of those which cannot be overloaded by friend function)

**97.How to implement Streaming Class?**

**Keypoints:**

Operator overloading of << and >> operator.

**98.Implement printf function**

Now, below here, is c printf implementation.

1 #include <stdio.h>

2 #include <stdlib.h>

3 #include <stdarg.h>

4 #include <string.h>

5

6 /\* Note: this is a minimal printf implementation \*/

7 /\* This is for building understanding only \*/

8 int print (char \* str, ...)

9 {

10 va\_list vl;

11 int i = 0, j=0;

12 char buff[100]={0}, tmp[20];

13 va\_start( vl, str );

14

15 while (str && str[i])

16 {

17 if(str[i] == '%'){

18 i++;

19 switch (str[i]) {

20 case 'c': {

21 buff[j] = (char)va\_arg( vl, int );

22 j++;

23 break;

24 }

25 case 'd': {

26 itoa(va\_arg( vl, int ), tmp, 10);

27 strcpy(&buff[j], tmp);

28 j += strlen(tmp);

29 break;

30 }

31 case 'x': {

32 itoa(va\_arg( vl, int ), tmp, 16);

33 strcpy(&buff[j], tmp);

34 j += strlen(tmp);

35 break;

36 }

37 }

38 } else {

39 buff[j] =str[i];

40 j++;

41 }

42 i++;

43 }

44 fwrite(buff, j, 1, stdout);

45 va\_end(vl);

46 return j;

47 }

Now, fwrite(const void \*ptr, size\_t size, size\_t n, FILE \*fp);

Now, ptr will be the buffer.

Size\_t size is the number of bytes we need to write.

size\_t n is the number of elements we need to write.

fp: will be stdout.

int atoi(const char \*str)

This function can help converting a null terminated character array to integer.

#include <stdio.h>

#include <stdarg.h>

void

foo(char \*fmt, ...)

{

va\_list ap;

int d;

char c, \*s;

va\_start(ap, fmt);

while (\*fmt)

switch (\*fmt++) {

case 's': /\* string \*/

s = va\_arg(ap, char \*);

printf("string %s\n", s);

break;

case 'd': /\* int \*/

d = va\_arg(ap, int);

printf("int %d\n", d);

break;

case 'c': /\* char \*/

/\* need a cast here since va\_arg only

takes fully promoted types \*/

c = (char) va\_arg(ap, int);

printf("char %c\n", c);

break;

}

va\_end(ap);

}

This function can be changed into printf too.

**99.Implementing scanf:**

int vfscanf ( FILE \* stream, const char \* format, va\_list arg );

int scanf (const char \*fmt, ...)

{

int count;

va\_list ap;

va\_start (ap, fmt);

**//va\_start is either initialized with count or the format specifier string**

count = vfscanf (stdin, fmt, ap);

va\_end (ap);

return (count);

}

**Important STL function:  
  
100.binary\_search(start\_ptr, end\_ptr, num) :**

This function returns boolean true if the element is present in the container, else returns false.

// C++ code to demonstrate the working of binary\_search()

#include<bits/stdc++.h>

using namespace std;

int main()

{

// initializing vector of integers

vector<int> arr = {10, 15, 20, 25, 30, 35};

// using binary\_search to check if 15 exists

if (binary\_search(arr.begin(), arr.end(), 15))

cout << "15 exists in vector";

else

cout << "15 does not exist";

cout << endl;

// using binary\_search to check if 23 exists

if (binary\_search(arr.begin(), arr.end(), 23))

cout << "23 exists in vector";

else

cout << "23 does not exist";

cout << endl;

}

**101.lower\_bound(start\_ptr, end\_ptr, num) :**

Returns pointer to “position of num” if container contains 1 occurrence of num. Returns pointer to “first position of num” if container contains multiple occurrence of num. Returns pointer to “position of next higher number than num” if container does not contain occurrence of num. Subtracting the pointer to 1st position i.e “vect.begin()” returns the actual index.

**102.upper\_bound(start\_ptr, end\_ptr, num) :**   
Returns pointer to “position of next higher number than num” if container contains 1 occurrence of num. Returns pointer to “first position of next higher number than last occurrence of num” if container contains multiple occurrence of num. Returns pointer to “position of next higher number than num” if container does not contain occurrence of num. Subtracting the pointer to 1st position i.e “vect.begin()” returns the actual index.

**103. Sort function:**The algorithm header contains it.

Now, sort function for arr:

sort(arr,arr+n);

Sort function for vector:

sort(vector.begin(),vector.end());

**Optional function pointer in sort function:**

**Now, in activity selection problem we need to sort it based on the finish time.**

Now, how to sort it based on finish time.

The sort function in c++ algorithm header accepts one optional functional pointer. (an example of callback function right??)

Now, we can write a function like the following:

bool activityCompare(Activity s1,Activity s2)

{

return s1.finish<s2.finish;

}

And, call the sort function

sort(arr,arr+n,activityCompare);

**How to sort a array in decreasing order?**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int arr[] = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};

int n = sizeof(arr)/sizeof(arr[0]);

sort(arr, arr+n, greater<int>());

cout << "Array after sorting : \n";

for (int i = 0; i < n; ++i)

cout << arr[i] << " ";

return 0;

}

This is an array which is sorted in decreasing order.

Now, for non-primitive types, we have to **write own Compare function.**

**STL Containers:**

**104.STL STACK:**

Main functions associated with stack stl:  
  
empty() – Returns whether the stack is empty  
size() – Returns the size of the stack  
top() – Returns a reference to the top most element of the stack  
push(g) – Adds the element ‘g’ at the top of the stack  
pop() – Deletes the top most element of the stack

**105.Dequeue:**

**std::deque**

major functions are:

push\_back:

push\_front:

pop\_back:

pop\_front:

operator[]: Access element

at: Access element

front: Access first element

back: Access last element

(front, back are normal queues)

**106.Queue:**

**empty**

Test whether container is empty (public member function )

**size**

Return size (public member function )

**front**

Access next element (public member function )

**back**

Access last element (public member function )

**push**

Insert element (public member function )

**pop**

Remove next element (public member function )

**107.std::unordered\_map**

In an unordered\_map, the key value is generally used to uniquely identify the element, while the mapped value is an object with the content associated to this key. Types of key and mapped value may differ.

Internally, the elements in the unordered\_map are not sorted in any particular order with respect to either their key or mapped values, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their key values (with a constant average time complexity on average).

unordered\_map containers are faster than map containers to access individual elements by their key, although they are generally less efficient for range iteration through a subset of their elements.

The Standard effectively mandates std::unordered\_set and std::unordered\_map implementations that use open hashing, which means an array of buckets, each of which holds the head of a logical (and typically actual) list. That requirement is subtle: **it's a consequence of the default max load factor being 1.0 and the guarantee that the table will not be rehashed unless grown beyond that load factor: that would be impractical without chaining, as the collisions with closed hashing become overwhelming as the load factor approaches 1:**

**insert**

Insert elements (public member function )

**erase**

Erase elements (public member function )

**clear**

Clear content (public member function )

**Swap**

Swap content (public member function)

**[]**Element access.

**at**

Access element.

**Buckets**

**bucket\_count**

Return number of buckets

**max\_bucket\_count**

Return maximum number of buckets

**bucket\_size**

Return bucket size (current number of elements in bucket)

**bucket**

Locate element's bucket

**Hash policy**

**load\_factor**

Return load factor

**max\_load\_factor**

Get or set maximum load factor

**rehash**

Set number of buckets

**reserve**

Request a capacity change

**108.std::map:  
  
Common operation:**

**insert**

Insert elements (public member function )

**erase**

Erase elements (public member function )

**swap**

Swap content (public member function )

**clear**

Clear content (public member function )

**operator[]**

Access element (public member function )

**at**

Access element (public member function )

**find()**

**New Operation:**

**count**

Count elements with a specific key (public member function )

**lower\_bound**

Return iterator to lower bound (public member function )

**upper\_bound**

Return iterator to upper bound (public member function )

**equal\_range**

Get range of equal elements (public member function )

(this functions are not supported by unordered\_map.

**Std::map internal implementation is red black tree.**

**Sample function Use:  
  
// erasing from map**

**#include <iostream>**

**#include <map>**

**int main ()**

**{**

**std::map<char,int> mymap;**

**std::map<char,int>::iterator it;**

**// insert some values:**

**mymap['a']=10;**

**mymap['b']=20;**

**mymap['c']=30;**

**mymap['d']=40;**

**mymap['e']=50;**

**mymap['f']=60;**

**it=mymap.find('b');**

**mymap.erase (it); // erasing by iterator**

**mymap.erase ('c'); // erasing by key**

**it=mymap.find ('e');**

**mymap.erase ( it, mymap.end() ); // erasing by range**

**// show content:**

**for (it=mymap.begin(); it!=mymap.end(); ++it)**

**std::cout << it->first << " => " << it->second << '\n';**

**return 0;**

**}**

Now, as you can see, it can be erased by key. We can erase it by range. (**mymap.erase ( it, mymap.end() );)**

We can erase it by iterator.

**Insert:**

**#include <iostream>**

**#include <map>**

**int main ()**

**{**

**std::map<char,int> mymap;**

**// first insert function version (single parameter):**

**mymap.insert ( std::pair<char,int>('a',100) );**

**mymap.insert ( std::pair<char,int>('z',200) );**

**std::pair<std::map<char,int>::iterator,bool> ret;**

**ret = mymap.insert ( std::pair<char,int>('z',500) );**

**if (ret.second==false) {**

**std::cout << "element 'z' already existed";**

**std::cout << " with a value of " << ret.first->second << '\n';**

**}**

**// second insert function version (with hint position):**

**std::map<char,int>::iterator it = mymap.begin();**

**mymap.insert (it, std::pair<char,int>('b',300)); // max efficiency inserting**

**mymap.insert (it, std::pair<char,int>('c',400)); // no max efficiency inserting**

**// third insert function version (range insertion):**

**std::map<char,int> anothermap;**

**anothermap.insert(mymap.begin(),mymap.find('c'));**

**// showing contents:**

**std::cout << "mymap contains:\n";**

**for (it=mymap.begin(); it!=mymap.end(); ++it)**

**std::cout << it->first << " => " << it->second << '\n';**

**std::cout << "anothermap contains:\n";**

**for (it=anothermap.begin(); it!=anothermap.end(); ++it)**

**std::cout << it->first << " => " << it->second << '\n';**

**return 0;**

**}**

**Make\_pair:**

**#include <utility>**

**#include <iostream>**

**int main () {**

**std::pair <int,char> foo;**

**std::pair <int,int> bar;**

**foo = std::make\_pair (1,'A');**

**bar = std::make\_pair (100,3);**

**std::cout << "foo: " << foo.first << ", " << foo.second << '\n';**

**std::cout << "bar: " << bar.first << ", " << bar.second << '\n';**

**return 0;**

**}**

**109.std:: set  
  
insert**

Insert element (public member function )

**erase**

Erase elements (public member function )

**swap**

Swap content (public member function )

**clear**

Clear content (public member function )

**find**

Count elements with a specific value (public member function )

**lower\_bound**

Return iterator to lower bound (public member function )

**upper\_bound**

Return iterator to upper bound (public member function )

**equal\_range**

Get range of equal elements (public member function )

**count**

Count elements with a specific key (public member function )

**Now, Note:**since, it’s set, it cannot support [], at (element access operator)  
Now, since, it is red black tree, it supports order statistics operation like: lower\_bound, upper\_bound, equal\_range.  
  
Now, in both set and map, I am seeing a function named **count.**

**It’s return value is : 1 if the container contains an element equivalent to val, or zero otherwise.**

**Supporting count function does not mean that these data structures, support duplicate elements.**

**110.std::multimap**It supports duplicate elements. Now, it is also implemented using red black tree. It just supports duplicate elements.

**insert**

Insert element (public member function )

**erase**

Erase elements (public member function )

**swap**

Swap content (public member function )

**clear**

Clear content (public member function **)**

**find**

Get iterator to element (public member function )

**count**

Count elements with a specific key (public member function )

**lower\_bound**

Return iterator to lower bound (public member function )

**upper\_bound**

Return iterator to upper bound (public member function )

**equal\_range**

Get range of equal elements (public member function )

**111.Use Of Iterator:**Iterators are used to point at the memory addresses of STL containers. They are primarily used in sequence of numbers, characters etc. They reduce the complexity and execution time of program.

**Operations of iterators :-**

**1. begin() :-** This function is used to return the beginning position of the container.

**2. end() :-** This function is used to return the end position of the container.

**// C++ code to demonstrate the working of**

**// iterator, begin() and end()**

**#include<iostream>**

**#include<iterator> // for iterators**

**#include<vector> // for vectors**

**using namespace std;**

**int main()**

**{**

**vector<int> ar = { 1, 2, 3, 4, 5 };**

**// Declaring iterator to a vector**

**vector<int>::iterator ptr;**

**// Displaying vector elements using begin() and end()**

**cout << "The vector elements are : ";**

**for (ptr = ar.begin(); ptr < ar.end(); ptr++)**

**cout << \*ptr << " ";**

**return 0;**

**}**

**advance() :-** This function is used to increment the iterator position till the specified number mentioned in its arguments.

**// C++ code to demonstrate the working of**

**// advance()**

**#include<iostream>**

**#include<iterator> // for iterators**

**#include<vector> // for vectors**

**using namespace std;**

**int main()**

**{**

**vector<int> ar = { 1, 2, 3, 4, 5 };**

**// Declaring iterator to a vector**

**vector<int>::iterator ptr = ar.begin();**

**// Using advance() to increment iterator position**

**// points to 4**

**advance(ptr, 3);**

**// Displaying iterator position**

**cout << "The position of iterator after advancing is : ";**

**cout << \*ptr << " ";**

**return 0;**

**}**

**4. next() :-** This function returns the new iterator that the iterator would point after advancing the positions mentioned in its arguments.

**5. prev() :-** This function returns the new iterator that the iterator would point after decrementing the positions mentioned in its arguments.

**// C++ code to demonstrate the working of**

**// next() and prev()**

**#include<iostream>**

**#include<iterator> // for iterators**

**#include<vector> // for vectors**

**using namespace std;**

**int main()**

**{**

**vector<int> ar = { 1, 2, 3, 4, 5 };**

**// Declaring iterators to a vector**

**vector<int>::iterator ptr = ar.begin();**

**vector<int>::iterator ftr = ar.end();**

**// Using next() to return new iterator**

**// points to 4**

**auto it = next(ptr, 3);**

**// Using prev() to return new iterator**

**// points to 3**

**auto it1 = prev(ftr, 3);**

**// Displaying iterator position**

**cout << "The position of new iterator using next() is : ";**

**cout << \*it << " ";**

**cout << endl;**

**// Displaying iterator position**

**cout << "The position of new iterator using prev() is : ";**

**cout << \*it1 << " ";**

**cout << endl;**

**return 0;**

**}**

**Reverse Iterator:**

Consider the following example:

**// vector::rbegin/rend**

**#include <iostream>**

**#include <vector>**

**int main ()**

**{**

**std::vector<int> myvector (5); // 5 default-constructed ints**

**int i=0;**

**std::vector<int>::reverse\_iterator rit = myvector.rbegin();**

**for (; rit!= myvector.rend(); ++rit)**

**\*rit = ++i;**

**std::cout << "myvector contains:";**

**for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)**

**std::cout << ' ' << \*it;**

**std::cout << '\n';**

**return 0;**

**}**

**Sample Code**

**112.Implementing Your Own String Class:**//Implementing my own string class

**#include<bits/stdc++.h>**

**using namespace std;**

**class myString**

**{**

**private:**

**char \*str;**

**int curr\_str\_size;**

**int curr\_position;**

**public:**

**myString();**

**myString(char \*str);**

**myString(const myString &);**

**myString& operator=(const myString &);**

**myString& operator=(const char \*);**

**friend ostream& operator<<(ostream &,const myString &);**

**//it cannot be part of this**

**//hence, friend function**

**//subscript operator overloading:**

**//gives direct access to the ith character of a string**

**char operator[](int);**

**int length();**

**void push\_back(char);**

**char pop\_back();**

**};**

**myString::myString()**

**{**

**//if no size is specified**

**//one default size can be given**

**//let's suppose that default size is 32**

**cout<<"Default constructor is called"<<endl;**

**try**

**{**

**str=new char[32];**

**curr\_str\_size=32;**

**curr\_position=0;**

**}**

**catch(bad\_alloc e)**

**{**

**throw ("Further string can not be created. As further heap memory cannot be allocated");**

**}**

**}**

**myString::myString(char \*str)**

**{**

**cout<<"parametric constructor is called"<<endl;**

**if(str==NULL)**

**{**

**throw ("The str which is passed as argument to this function is NULL");**

**}**

**//however, it does not handle if the str is not null terminated**

**this->curr\_str\_size=strlen(str)+1;**

**this->curr\_position=strlen(str);**

**this->str=new char[this->curr\_str\_size];**

**strcpy(this->str,str);**

**}**

**myString::myString(const myString &myStr)**

**{**

**cout<<"Copy constructor is called"<<endl;**

**//self reference checking**

**if(this==&myStr)**

**{**

**return;**

**}**

**//why const reference sending is important**

**//const because, we need not the argument to be modified**

**//reference because, otherwise copy constructor will be invoked for the argument itself**

**//delete str;**

**//here, you cannot delete it, because, it is an uninitialized pointer**

**//uninitialized pointer dereferencing or deleting can cause segmentation fault**

**//hence, this will does not cause segmentation fault**

**this->curr\_str\_size=strlen(myStr.str)+1;**

**this->curr\_position=strlen(myStr.str);**

**str=new char[this->curr\_str\_size];**

**strcpy(str,myStr.str);**

**//in copy constructor, we can access other object's private member. But, the object**

**//should be of same class**

**}**

**//assignment operator overloading**

**myString& myString::operator=(const myString &myStr)**

**{**

**//self reference checking**

**cout<<"Assignment operator is overloaded"<<endl;**

**if(this==&myStr)**

**{**

**return \*this;**

**}**

**delete str;**

**//it was never initialized to null**

**//hence, this will does not cause segmentation fault**

**/\*Important for this kind of stupid operation we do every day**

**myString str;**

**str="Sayak Haldar";**

**\*/**

**this->curr\_str\_size=strlen(myStr.str)+1;**

**str=new char[this->curr\_position];**

**strcpy(str,myStr.str);**

**return \*this;**

**}**

**myString& myString::operator=(const char\* str)**

**{**

**cout<<"Assignment operator is overloaded"<<endl;**

**//no need to self reference checking**

**delete this->str;**

**this->curr\_position=strlen(str);**

**this->curr\_str\_size=this->curr\_position;**

**strcpy(this->str,str);**

**return \*this;**

**}**

**char myString::operator[](int index)**

**{**

**if(index<0||index>=curr\_position)**

**{**

**throw ("Index out of bound exception");**

**}**

**return str[index];**

**}**

**int myString::length()**

**{**

**return this->curr\_position;**

**}**

**void myString::push\_back(char c)**

**{**

**if(this->curr\_position+1>=this->curr\_str\_size)**

**{**

**//we need to realloc**

**char \*tempStr=new char[this->curr\_str\_size+1];**

**strcpy(tempStr,str);**

**delete str;**

**//now, tempStr is not a pointer alias**

**//hence, it will remain fine even if we delete str**

**str=new char[this->curr\_str\_size+1+32];**

**//allocating some extra space to make push\_back operation little less time consuming**

**this->curr\_str\_size=this->curr\_str\_size+32;**

**strcpy(str,tempStr);**

**str[this->curr\_position]=c;**

**str[this->curr\_position+1]='\0';**

**//very important, otherwise it wont be null terminated**

**this->curr\_position=this->curr\_position+1;**

**delete tempStr;**

**//otherwise, there will be a memory leak**

**}**

**//now, it good for checking**

**//however, we still need to remember that curr\_position is in 1 indexing**

**else**

**{**

**str[this->curr\_position]=c;**

**str[this->curr\_position+1]='\0';**

**this->curr\_position=this->curr\_position+1;**

**}**

**}**

**char myString::pop\_back()**

**{**

**char c=str[curr\_position-1];**

**//remember it is a null terminated string and curr\_position is in 1 indexing**

**//hence,str[curr\_position] will actually contain '\0';**

**str[curr\_position-1]='\0';**

**return c;**

**}**

**ostream& operator<<(ostream &os,const myString &myStr)**

**{**

**return os<<myStr.str;**

**}**

**int main()**

**{**

**myString str1;**

**//this will invoke default constructor**

**str1="Sayak Haldar";**

**//this will invoke assignment operator**

**cout<<str1<<endl;**

**myString str2("Surekha Haldar");**

**//this will invoke parametric constructor**

**cout<<str2<<endl;**

**myString str3(str2);**

**//this will invoke copy constructor**

**cout<<str3<<endl;**

**myString str4;**

**str4=str1;**

**//this will invoke assignment operator**

**cout<<str4<<endl;**

**}**

However, in this approach, we do not use a destructor, which is a must is this case. Like, this program will leak memory.

**Future goals:**enabling str[i]=’c’ where str is an object. But, this is actually violating the class rules and all.

**113.Implementing Your Own Stack:**

**#include<bits/stdc++.h>**

**using namespace std;**

**template<class T,int size=32>**

**class myStack**

**{**

**private:**

**T stack[size];**

**int topIdx;**

**//it will be initialised to -1**

**public:**

**myStack();**

**void push(T);**

**T pop();**

**bool empty();**

**T top();**

**};**

**template<class T,int size>**

**myStack<T,size>::myStack()**

**{**

**}**

**template<class T,int size>**

**void myStack<T,size>::push(T n)**

**{**

**if(topIdx==size-1)**

**{**

**throw "Stack is full";**

**}**

**stack[++topIdx]=n;**

**}**

**template<class T, int size>**

**T myStack<T,size>::pop()**

**{**

**if(topIdx==-1)**

**{**

**throw "Stack is empty";**

**}**

**T elem=stack[topIdx--];**

**return elem;**

**}**

**template <class T,int size>**

**T myStack<T,size>::top()**

**{**

**if(topIdx==-1)**

**{**

**throw "Stack is empty";**

**}**

**return stack[topIdx];**

**}**

**template<class T,int size>**

**bool myStack<T,size>::empty()**

**{**

**return (topIdx==-1);**

**}**

**int main()**

**{**

**myStack<int,40> myStack1;**

**myStack1.push(10);**

**myStack1.push(20);**

**myStack1.push(30);**

**myStack<int> myStack2;**

**myStack1.push(40);**

**return 0;**

**}**

Why this cannot be like following:

**#include<bits/stdc++.h>**

**using namespace std;**

**template<class T,int size=32>**

**class myStack**

**{**

**private:**

**T stack[size];**

**int topIdx;**

**//it will be initialised to -1**

**public:**

**myStack();**

**void push(T);**

**T pop();**

**bool empty();**

**T top();**

**};**

**template<class T,int size=32>**

**myStack<T,size>::myStack()**

**{**

**}**

**template<class T,int size=32>**

**void myStack<T,size>::push(T n)**

**{**

**if(topIdx==size-1)**

**{**

**throw "Stack is full";**

**}**

**stack[++topIdx]=n;**

**}**

**template<class T, int size=32>**

**T myStack<T,size>::pop()**

**{**

**if(topIdx==-1)**

**{**

**throw "Stack is empty";**

**}**

**T elem=stack[topIdx--];**

**return elem;**

**}**

**template <class T,int size=32>**

**T myStack<T,size>::top()**

**{**

**if(topIdx==-1)**

**{**

**throw "Stack is empty";**

**}**

**return stack[topIdx];**

**}**

**template<class T,int size=32>**

**bool myStack<T,size>::empty()**

**{**

**return (topIdx==-1);**

**}**

**int main()**

**{**

**myStack<int,40> myStack1;**

**myStack1.push(10);**

**myStack1.push(20);**

**myStack1.push(30);**

**myStack<int> myStack2;**

**myStack1.push(40);**

**return 0;**

**}**

Default template arguments, just like default function arguments, may only be defined once (per translation unit); not even repeating the exact same definition is allowed.