**Operator interview questions:**

Restrictions of the modulo operator:   
The modulo operator has quite some restrictions or limitations.

• The % operator cannot be applied to floating-point numbers i.e float or double. If you try to use the modulo operator with floating-point constants or variables, the compiler will produce an error:

**In C++ We have a inbuilt function for the modulus of floating numbers:**

*// CPP program to find modulo of floating*

*// point numbers using library function.*

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

*using namespace std;*

*// Driver Function*

*int main()*

*{*

*double a = 9.7, b = 2.3;*

*cout << fmod(a, b);*

*return 0;*

*}*

**Shift Operator Important Notes:**

The left shift and right shift operators should not be used for negative numbers. The result of is undefined behaviour if any of the operands is a negative number. For example results of both -1 << 1 and 1 << -1 is undefined.

If the number is shifted more than the size of integer, the behaviour is undefined. For example, 1 << 33 is undefined if integers are stored using 32 bits. For bit shift of larger values 1ULL<<62  ULL is used for Unsigned Long Long which is defined using 64 bits which can store large values.

The left-shift by 1 and right-shift by 1 are equivalent to the product of first term and 2 to the power given element(1<<3 = 1\*pow(2,3)) and division of first term and second term raised to power 2 (1>>3 = 1/pow(2,3)) respectively.   
As mentioned in point

**Dynamic Cast:** Safely converts pointers and references to classes up and down and sideways along the inheritance hierarchy.

syntax: dynamic\_cast<new\_type>(expression)

If the cast is successful , dynamic\_cast returns a value of the new\_type, If the cast fails and new\_type is pointer type, it returns a null pointer of that type. If the cast fails and new\_type is a reference type, it throws and exception that matches a handler of type std:bad\_cast.

**BitWise Operators:**

• The **& (bitwise AND)** in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

• The **| (bitwise OR)** in C or C++ takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

• The **^ (bitwise XOR)** in C or C++ takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

• The **<< (left shift)** in C or C++ takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.

• The **>> (right shift)** in C or C++ takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.

• The **~ (bitwise NOT)** in C or C++ takes one number and inverts all bits of it.

**enable\_shared\_from\_this:**

The header <boost/enable\_shared\_from\_this.hpp> defines the class template enable\_shared\_from\_this. It is used as a base class that allows a shared\_ptr or a weak\_ptr to the current object to be obtained from within a member function.

enable\_shared\_from\_this<T> defines two member functions called shared\_from\_this that return a shared\_ptr<T> and shared\_ptr<T const>, depending on constness, to this. It also defines two member functions called weak\_from\_this that return a corresponding weak\_ptr.

Example

#include <boost/enable\_shared\_from\_this.hpp>

#include <boost/shared\_ptr.hpp>

#include <cassert>

class Y: public boost::enable\_shared\_from\_this<Y>

{

public:

boost::shared\_ptr<Y> f()

{

return shared\_from\_this();

}

};

int main()

{

boost::shared\_ptr<Y> p(new Y);

boost::shared\_ptr<Y> q = p->f();

assert(p == q);

assert(!(p < q || q < p)); // p and q must share ownership

}

namespace boost

{

template<class T> class enable\_shared\_from\_this

{

public:

shared\_ptr<T> shared\_from\_this();

shared\_ptr<T const> shared\_from\_this() const;

weak\_ptr<T> weak\_from\_this() noexcept;

weak\_ptr<T const> weak\_from\_this() const noexcept;

}

}

**template<class T> shared\_ptr<T> enable\_shared\_from\_this<T>::shared\_from\_this();**

**template<class T> shared\_ptr<T const> enable\_shared\_from\_this<T>::shared\_from\_this() const;**

Requires: enable\_shared\_from\_this<T> must be an accessible base class of T. \*this must be a subobject of an instance t of type T.

Returns: If a shared\_ptr instance p that *owns* t exists, a shared\_ptr<T> instance r that shares ownership with p.

Postconditions: r.get() == this.

Throws: bad\_weak\_ptr when no shared\_ptr *owns* \*this.

**template<class T> weak\_ptr<T> enable\_shared\_from\_this<T>::weak\_from\_this() noexcept;**

**template<class T> weak\_ptr<T const> enable\_shared\_from\_this<T>::weak\_from\_this() const noexcept;**

Requires: enable\_shared\_from\_this<T> must be an accessible base class of T. \*this must be a subobject of an instance t of type T.

Returns: If a shared\_ptr instance p that *owns* t exists or has existed in the past, a weak\_ptr<T> instance r that shares ownership with p. Otherwise, an empty weak\_ptr.

1.7  Null Pointers

You can initialize a pointer to 0 or NULL, i.e., it points to nothing. It is called a null pointer. Dereferencing a null pointer (\*p) causes an STATUS\_ACCESS\_VIOLATION exception.

int \* iPtr = 0; // Declare an int pointer, and initialize the pointer to point to nothing

cout << \*iPtr << endl; // ERROR! STATUS\_ACCESS\_VIOLATION exception

int \* p = NULL; // Also declare a NULL pointer points to nothing

Initialize a pointer to null during declaration is a good software engineering practice.

C++11 introduces a new keyword called nullptr to represent null pointer.

RTTI allows programs that use pointers or references to base classes to retrieve the actual derived types of the objects to which these pointers or references refer.

RTTI is provided through two operators:

• The **typeid** operator, which returns the actual type of the object referred to by a pointer (or a reference).

• The **dynamic\_cast** operator, which safely converts from a pointer (or reference) to a base type to a pointer (or reference) to a derived type.

• class Base { };

•

• class Derived : public Base { };

•

• int main()

• {

• Base b;

• Derived d;

•

• Base \*pb = dynamic\_cast<Base\*>(&d;); // #1

• Derived \*pd = dynamic\_cast<Derived\*>(&b;); // #2

•

• return 0;

• }

//line 13: will throw an error, because , base to derived class conversion allowed only if base is polymorphic(means it has virtual functions).

Error in line:13

error C2683: 'dynamic\_cast' : 'Base' is not a polymorphic type.

• Reading the multiple inputs

cin >> a >> b >>c;

2) Setting the precision while printing the double value

double d;

cout << "Geeks" << endl;

cin >> d;

cout.precision(<precisionvalue>);

cout << d;

3) Templates -- 2 types of templates, 1 one is function templates.

2nd one is class template.

4) Predefined Macros:

The following macro names are always defined (they all begin and end with two underscore characters, \_):

**macro**

**value**

\_\_LINE\_\_

Integer value representing the `rrent line in the source code file being compiled.

\_\_FILE\_\_

A string literal containing the presumed name of the source file being compiled.

\_\_DATE\_\_

A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began.

\_\_TIME\_\_

A string literal in the form "hh:mm:ss" containing the time at which the compilation process began.

\_\_cplusplus

An integer value. All C++ compilers have this constant defined to some value. Its value depends on the version of the standard supported by the compiler:

**• 199711L**: ISO C++ 1998/2003

**• 201103L**: ISO C++ 2011

Non conforming compilers define this constant as some value at most five digits long. Note that many compilers are not fully conforming and thus will have this constant defined as neither of the values above.

\_\_STDC\_HOSTED\_\_

1 if the implementation is a *hosted implementation* (with all standard headers available)

0 otherwise.

The following macros are optionally defined, generally depending on whether a feature is available:

**macro**

**value**

\_\_STDC\_\_

In C: if defined to 1, the implementation conforms to the C standard.

In C++: Implementation defined.

\_\_STDC\_VERSION\_\_

In C:

**• 199401L**: ISO C 1990, Ammendment 1

**• 199901L**: ISO C 1999

**• 201112L**: ISO C 2011

In C++: Implementation defined.

\_\_STDC\_MB\_MIGHT\_NEQ\_WC\_\_

1 if multibyte encoding might give a character a different value in character literals

\_\_STDC\_ISO\_10646\_\_

A value in the form yyyymmL, specifying the date of the Unicode standard followed by the encoding of wchar\_t characters

\_\_STDCPP\_STRICT\_POINTER\_SAFETY\_\_

1 if the implementation has *strict pointer safety* (see get\_pointer\_safety)

\_\_STDCPP\_THREADS\_\_

1 if the program can have more than one thread

Particular implementations may define additional constants.

**For example:**

1

2

3

4

5

6

7

8

9

10

11

12

13

// standard macro names  
#include <iostream>  
using namespace std;

int main()  
{  
 cout << "This is the line number " << \_\_LINE\_\_;  
 cout << " of file " << \_\_FILE\_\_ << ".\n";  
 cout << "Its compilation began " << \_\_DATE\_\_;  
 cout << " at " << \_\_TIME\_\_ << ".\n";  
 cout << "The compiler gives a \_\_cplusplus value of " << \_\_cplusplus;  
 return 0;  
}

This is the line number 7 of file /home/jay/stdmacronames.cpp.  
Its compilation began Nov 1 2005 at 10:12:29.  
The compiler gives a \_\_cplusplus value of 1

• Include a local file can be done as #include "file.h"

• Include a library can be done as #include "file"

• Compiling a code(.cpp or .h file) without main use -c option // Ex: g++ -c Vir.cpp, without -c it will give error.

**Data Types: in C++:**

**cout << "size of bool: " << sizeof(bool) << endl;**

**cout << "size of char: " << sizeof(char) << endl;**

**cout << "size of unsigned short int: " << sizeof(unsigned short) << endl;**

**cout << "size of short int: " << sizeof(short) << endl;**

**cout << "size of unsigned long int: " << sizeof(unsigned long) << endl;**

**cout << "size of long : " << sizeof(long) << endl;**

**cout << "size of int : " << sizeof(int) << endl;**

**cout << "size of unsigned int : " << sizeof(unsigned int) << endl;**

**cout << "size of unsigned long long : " << sizeof(unsigned long long) << endl;**

**cout << "size of long long : " << sizeof(long long) << endl;**

**cout << "size of float : " << sizeof(float) << endl;**

**cout << "size of double : " << sizeof(double) << endl;**

**size of bool: 1**

**size of char: 1**

**size of unsigned short int: 2**

**size of short int: 2**

**size of unsigned long int: 8**

**size of long : 8**

**size of int : 4**

**size of unsigned int : 4**

**size of unsigned long long : 8**

**size of long long : 8**

**size of float : 4**

**size of double : 8**

• “auto” key word is introduced in the C++11, type of the variable is apparent given the value initialization value its being assigned ,

Auto requires to initialize the variable for the compiler.

Ex: auto Flag = true; //here compiler check the initialization value and it finds it as best suits for bool.

**Ex:**

Auto flag = true;

Auto Number = 250000000000;

Cout << “size of” << flag << endl; //1 (as compiler derives it as bool)

Cout << “size of” << Number << endl; // 8 (as compiler derives it as long)

**Extern “c” and Name Mangling:**

C++ supports function overloading, i.e., there can be more than one functions with same name and differences in parameters. How does C++ compiler distinguishes between different functions when it generates object code – it changes names by adding information about arguments. This technique of adding additional information to function names is called Name Mangling. C++ standard doesn’t specify any particular technique for name mangling, so different compilers may append different information to function names.

int  f (void) { return 1; }

int  f (int)  { return 0; }

void g (void) { int i = f(), j = f(0); }

A C++ compiler may mangle above names to following:

int  \_\_f\_v (void) { return 1; }

int  \_\_f\_i (int)  { return 0; }

void \_\_g\_v (void) { int i = \_\_f\_v(), j = \_\_f\_i(0); }

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

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

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

int main()

{

    printf("GeeksforGeeks");

    return 0;

}

Error: undefined reference to `printf(char const\*, ...)'

ld returned 1 exit status

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

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

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

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

*extern "C"*

*{*

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

*}*

*int main()*

*{*

*printf("GeeksforGeeks");*

*return 0;*

*}*

Therefore, all C style header files (stdio.h, string.h, .. etc) have their declarations in extern “C” block.

#ifdef \_\_cplusplus

extern "C" {

#endif

    /\* Declarations of this file \*/

#ifdef \_\_cplusplus

}

#endif

Following are main points discussed above:

**1.**Since C++ supports function overloading, additional information has to be added to function names (called name mangling) to avoid conflicts in binary code.  
**2.** Function names may not be changed in C as C doesn’t support function overloading. To avoid linking problems, C++ supports extern “C” block. C++ compiler makes sure that names inside extern “C” block are not changed.

**Normal New(operator) vs Placement new: (new vs placement new)**

a) Normal new allocates memory in heap and constructs objects tehre whereas using placement new, object construction can be done at known address.

b) With normal new, it is not known that, at what address or memory location it’s pointing to, whereas the address or memory location that it’s pointing is known while **using placement new.**

c) The deallocation is done using delete operation when allocation is done by new but there is no placement delete, but if it is needed one can write it with the help of destructor.

**Syntax:**

new (address) (type) initializer:

// C++ program to illustrate the placement new operator

#include<iostream>

using namespace std;

int main()

{

    // buffer on stack

    unsigned char buf[sizeof(int)\*2] ;

    // placement new in buf

    int \*pInt = new (buf) int(3);

    int \*qInt = new (buf + sizeof (int)) int(5);

    int \*pBuf = (int\*)(buf+0) ;

    int \*qBuf = (int\*) (buf + sizeof(int));

    cout << "Buff Addr             Int Addr" << endl;

    cout << pBuf <<"             " << pInt << endl;

    cout << qBuf <<"             " << qInt << endl;

    cout << "------------------------------" << endl;

    cout << "1st Int             2nd Int" << endl;

    cout << \*pBuf << "                         "

         << \*qBuf << endl;

    return 0;

}

Output:

Buff Addr Int Addr

0x69fed8 0x69fed8

0x69fedc 0x69fedc

------------------------------

1st Int 2nd Int

3 5

• Defining constants:

• Using preprocessor directives( #define pi 3.1122)

• Using **Literal constants** (int a = 0;)

• Declared constants using the key word **const**.

• Constant expressions using the **constexpr** keyword (new to C++11)

• Enumerated constants using the **enum** keyword

• Defined constants that are not recommended and deprecated.

**Integer Literal constants:**

12 – an Integer

12U – an Unsigned integer

12L – a long integer

12LL – a long long integer

**Floating-point Literal constants:**

12.1 – a double

12.1F – a float

12.1L – a long double

**Character Literal Constants:**

\n – new line

\r – return

\t – tab

\b – backspace

\’ – single quote

\” – double quote

\\ -- back slash

**Declared constants:**

Const double pi {3.1415926};

Const int months {12};

**Note:** Don’t use defined constants (#define pi 3.1415926) in Modern C++**.**

**Const expression:**

**• The** difference b/w the const and constexpr is very little

Constexpr double GetPi() {return 22.0/7;};

Constexpr double TwicePi() {return 2\* GetPi();};

A constant Method in a class cannot modify the value of any of the variable in that class. But if we make that variable as mutable its allowed (i.e a constant method can modify the mutable variable).

**Enumerated Data Types:**

When declaring an enumerated constant, the compiler converts the enumerated values such as each one into integers.

Each enumerated value is specified is one more than the previous value. You have the choice of specifying the starting value, if this is not specified, compiler takes it as 0. If you want you can also explicitly specify value against each of the enumerated constants by initializing thems.

Ex:

Enum Colors {

Violet = 0,

Pink,

Red,

Blue,

Green

} //Here compiler will automatically assignment incrementing values from pink

Enum Colors {

Violet = 21,

Pink, //automatically 22

Red, //automatically 23

Blue,

Green

}

**Note:**  In the below snippet of code, “count” will give automatically the no. of the elements except the count

**enum** ctypes {strength,ability,intelligence, count};

std::array<**int**,count> Crtarray;

//Here count value is 3

**How Function Call Works:**

a) Functions use the “function call stack”

i) Analogous to a stack of books.

ii) LIFO – Last in First Out

iii) push and POP.

b) Stack Frame or Activation Record.

i) Functions must return control to function that is called.

ii) Each time a function is called we create an new activation record and push it on to the stack.

iii) When a function terminates we pop the activation record and return.

iv) Local variables and function parameters are allocated on the stack.

c) Stack size is finite – Stack Overflow.

**Inline:**

**Function calls have some amount of overhead**

We can suggest to the compiler to compile them ‘inline’.

i)

a) Avoid function call overhead.

b) Generate inline assembly code

c) faster

d) could cause code float.

ii) Compilers optimizations are very sophisticated.

a) Will likely inline even without your suggestion.

**Ex:**

*Inline int add\_numbers(int a, int b){*

*return a+b;*

*}*

*Int main(){*

*Int result {0};*

*Result = add\_numbers(100,200);*

*Return 0;*

*}*

**Mutable Key word:**

**The mutable storage class specifier in C++**

auto, register, static and extern are the storage class specifiers in C. typedef is also considered as a storage class specifier in C. C++ also supports all these storage class specifiers. In addition to this C++, **adds one important storage class specifier whose name is mutable**.

**What is the need of mutable?**  
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.

The keyword mutable is mainly used to allow a particular data member of const object to be modified. When we declare a function as const, the this pointer passed to function becomes const. Adding mutable to a variable allows a const pointer to change members.  
mutable is particularly useful if most of the members should be constant but a few need to be updateable. Data members declared as mutable can be modified even though they are the part of object declared as const. You cannot use the mutable specifier with names declared as static or const, or reference.

**Virtual Function:**

It’s a member function declared in the base class and is redefined by derived class.

a) They are must public.

b) They cannot be static or friend function of other class.

c) They should be accessed using a pointer or reference of base class type(in the derived function) to achieve run time polymorphism.

Ex:

// virtual members  
#include <iostream>  
using namespace std;

class Polygon {  
 protected:  
 int width, height;  
 public:  
 void set\_values (int a, int b)  
 { width=a; height=b; }  
 virtual int area ()  
 { return 0; }  
};

class Rectangle: public Polygon {  
 public:  
 int area ()  
 { return width \* height; }  
};

class Triangle: public Polygon {  
 public:  
 int area ()  
 { return (width \* height / 2); }  
};

int main () {  
 Rectangle rect;  
 Triangle trgl;  
 Polygon poly;  
 Polygon \* ppoly1 = &rect;  
 Polygon \* ppoly2 = &trgl;  
 Polygon \* ppoly3 = &poly;  
 ppoly1->set\_values (4,5);  
 ppoly2->set\_values (4,5);  
 ppoly3->set\_values (4,5);  
 cout << ppoly1->area() << '\n';  
 cout << ppoly2->area() << '\n';  
 cout << ppoly3->area() << '\n';  
 return 0;  
}

*From <http://www.cplusplus.com/doc/tutorial/polymorphism/>*

**Abstract Classes:**

Abstract base classes are something very similar to the Polygon class in the previous example. They are classes that can only be used as base classes, and thus are allowed to have virtual member functions without definition (known as pure virtual functions). **The syntax is to replace their definition by =0 (an equal sign and a zero):**

Ex:

// abstract class CPolygon  
class Polygon {  
 protected:  
 int width, height;  
 public:  
 void set\_values (int a, int b)  
 { width=a; height=b; }  
 virtual int area () =0; //Here it is a Pure Virtual Fucntion:

};

Classes that contain at least one *pure virtual function* are known as *abstract base classes*.

Abstract base classes cannot be used to instantiate objects. Therefore, this last abstract base class version of Polygon could not be used to declare objects like:

Polygon mypolygon;

But an *abstract base class* is not totally useless. It can be used to create pointers to it, and take advantage of all its polymorphic abilities. For example, the following pointer declarations would be valid:

1

2

Polygon \* ppoly1;  
Polygon \* ppoly2;

*From <http://www.cplusplus.com/doc/tutorial/polymorphism/>*

**Note**: When deriving a struct from a Struct/class, the default access specifier for a base class/Struct is public. And when deriving a class, the default access specifier is private. That’s use the Struct always:

Struct A {

Public:

int a , b;

int area{

return a\*b

}

}

Struct B : A {};

Int main(){

B b;

b.a = 10;

getchar();

return 0;

}

**Shared Pointer or Smart Pointer:**  template < class T> class

a) Manages the storage of a pointer, provides a limited storage garbage collection facility. Possibly sharing the management with other objects.

• Objects of shared\_ptr have the ability of taking owner ship of a pointer and share that ownership, once they take ownership, the group of owners become responsible for its deletion when last one of them releases the ownership.

• The shared\_ptr objects release ownership on the object they co-own as soon as they themselves are destroyed or as soon as their value changes either by assignment operation or by an explicit call to the shared\_ptr::reset.

• Once all the shared\_ptr objects that share ownership over a pointer have released this ownership, managed object is deleted (normally by calling ::delete , but may be different deleter may be specified on construction).

• The shared\_ptr objects only share ownership by copying their value, If two shared\_ptr are constructed (or made) from the same (non-shared\_ptr) pointer, they will both be owning the pointer without sharing it, causing potential access problems when one of them releases it (deleting its managed object) and leaving the other pointing to an invalid location.

• Additionally, shared\_ptr objects can share ownership over a pointer while at the same time pointing to another object. This ability is known as *aliasing* (see constructors), and is commonly used to point to member objects while owning the object they belong to. Because of this, a shared\_ptr may relate to two pointers:

• A stored pointer, which is the pointer it is said to *point to*, and the one it dereferences with operator\*.

• An *owned pointer* (possibly shared), which is the pointer the ownership group is in charge of deleting at some point, and for which it counts as a use.

• A shared\_ptr that does not own any pointer is called an *empty* shared\_ptr. A shared\_ptr that points to no object is called a *null* shared\_ptr and shall not be dereferenced. Notice though that an *empty* shared\_ptr is not necessarily a nullshared\_ptr, and a null shared\_ptr is not necessarily an *empty* shared\_ptr.

**Constructors:**

• Special member method.

• Invoked during the object creation.

• Useful for initialization of the varialbes

• Same name as class name

• No return type is specified.

• Can be overloaded.

Ex:

Class Player {

private:

std::string name;

int age;

int xp;

public:

//Overloaded Constructors.

Player(); //No Arg constructor

Player(std::string na, int a);

Player(std::string na, int a, int x);

Player(double age);

Player(doube age, std::string name);

~Player(); //Destructor.

}

**Class that prohibits Instantiation on the Stack: (Declare destructor as Private)**

Space on the stack is often limited. If you are writing a database that may contain a gigabyte of data in the interal structures, you might want to ensure that a client of this class cannot instantiate on Stack. Instead it is forced to create instances only the heap. The key to ensuring this is declaring as destructor private.

*class Mongodb{*

*private:*

*//private destructor*

*~ Mongodb(){*

*}*

*}*

*int main(){*

*Mongodb mdb; //Compiler throws an error.*

*}*

***Solution: (It can have a static public member function that destroys an instance)***

*int main(){*

*Mongodb \*mdb. = new Mongodb();*

*Mongodb:: deleteinstance(mdb);*

*return 0;*

*}*

*class Mongodb{*

*public:*

*static void deleteinstance(Mongodb\* db){*

*delete db;*

*}*

*private:*

*//private destructor*

*~ Mongodb(){*

*}*

*}*

**Destructors:**

• Special member method.

• Same name as class name with a tilde beginning( ~)

• Invoked automatically when an object is destroyed (Compiler calls the destructor automatically).

• No return type and No parameters.

• Only one destructor per class is allowed, -- Cannot be overloaded.

• Useful to release memory and other resources ( like uninitializing variables and files closing)

Destructor will be called when a local object goes out of scope, or when we delete a pointer to an object.

**Important Note**: When a class has private destructor, only dynamic objects of that class can be created. Following is a way to create classes with private destructors and have a function as friend of the class.

//Creating of objects:

{

Player p1; //default constr.

Player p2 {"shankar", 29};

Player p3 {"madhavi", 29, 2};

} // As the objects goes out of scope, destructors will be called automatically.

Player \*p4 = new Player {2.0};

delete p4; //Destructor is called.

If a user does not provide any constructor or destructor , Compiler will provide a default constructor and default destructor.

-> when an object is created for a class, Copy constructor is also called.

--> If an Args constructor is already present in the class, compiler won't create any default constructor. If we try creating an object with default constructor, compiler will throw an error.

**Constructor Initialization List:**

• using the initialization list, members are created and initialized only once, with given value.

• Using the assignment , members are initialized with a default value, then reassigned in the constructor body.

In some cases( constants and references), you can only use initialization lists because:

a) these must be initialized with a valid value.

b) Once they are constructed, reassigning them is illegal.

class A::A()   
 : a(0), b(0), c(0)   
{   
}

class A::A()   
{   
 a = 0   
 b = 0;   
 c = 0;   
}

**Delegation of Constructors:**

• Often code for the constructors are very similar.

• Duplicated code can lead to errors

• C++ allows delegating of constructors

a) Code for one constructor can call another in the initialization list.

b) Avoid duplicating code.

**Default Constructor parameters:**

#include <iostream>

#include <string>

using namespace std;

class Player{

private:

string name;

int age;

int exp;

public:

Player(string s = "None" , int n = 0, int e = 0);

};

Player::Player(string n, int a, int e):name(n),age(a),exp(e){

cout << "three arg constr. is called" << endl;

cout << "values are " << "name: " << n << " age:" << a << " exp:" << e << endl;

};

int main(){

Player p1;

Player p2("madhavi");

Player p3("shan", 10);

Player p4("unknown", 34, 37);

return 0;

}

**Copy Constructor:**

When objects are copied, c++ must create a new object from an existing object

When a copy of an object is created:

• Passing an object by value as a parameter

• Returning an object from a function by value.

• Constructing an object based on another object of same class.

If we don’t define the copy constuctor c++ compiler will create a copy constructor.

Copies the value of each data member, to the new object.

Beware if there is a pointer data member:

• Pointer will be copied.

• Not what is pointing to .

• Shall vs Deep.

Best Practices:

• Provide a copy constrctor when your class has a raw pointer members.

2) Provide copy constructor with a const reference member.

• Use STL classes as they provide the copy constructors already.

• Avoid using the raw pointer data members if possible.

Ex:

Defining copy constructor.

Player::Player(const Player &source);

Account::Account(const Account &source);

Implementation of copy constructor:

Player::Player(const Player &source)

//Either create a new object or use initialization list

: name(source.name), age(source.age), exp(source.exp){

}

Player::Player(const Player &source)

: name{source.name} ,age{source.age}, exp{source.exp}{

}

Pass an object by value:

Ex:

Player p1{"shankar", 10,20};

Void display(Player p2){

}

display(p1);

Returning an Object from a function:

Player enemy;

Player create\_new\_player(){

Player p2{"madhavi", 30, 0};

return p2;

}

enemy = create\_new\_player();

**Creating a new object based on existing object of same class:**

Player p3 {"sachin", 45, 20};

Player p4 = p3; or Player p4 {p3};

**Shallow Copy:**

Default copy constructor:

a) Do memberwise copy.

b) Each data member is copied from source object.

c) the pointer is copied, but not what it points to (shallow copy)

**Problem**: When we release the storage in the constructor, the other object still refers to the release storage;

**Deep Copy:** (create new storage and copy values)

Deep::Deep(const Deep &source){

data = new int;

\*data = \*source.data;

cout << "copy constuctor -- deep" << endl;

}

Deep::Deep(const Deep &source)

: Deep(\*source.data){

cout << "deep copy constructor delegrate is called" << endl;

}

**Move Construtors:**

• Some times copy constructors are called, many times automatically due to copy semantics in c++

• Copy constructors doing the deep copying have a significant performance bottle neck.

• C++11 introduced move semantics, and move constructors.

• Move constructor moves an object rather than copy it.

• Optional but recommended when you have a raw poniter.

**• Copy elision, -- C++ may optimize copying away completely (RVO -- Return Value Optimization)**

**Rvalue References:**

• Used in move semantics and perfect forwarding

• Move semantics is all about r-value references.

• Used by move constructor and move assignment operator to efficiently move an object rather than copy.

d) R value references operator **(&&)**

**Ex:**

Int x {100};

Int &l = x; // l is l value reference.

L = 10; // change x to 10;

Int &&r = 200; //r value reference

R = 300; // change r 300

Int &&x\_ref = x; //Compiler error

**L-Value reference parameter:**

Int x {100}; // x is an l value

Void func(int &x) ; //A

Func(x);

Fucn(200); //Error 200 is an r- value

**Error:** cannot bind non-const lvalue reference of type int & to an rvalue reference of type int.

**R-Value references parameters:**

Int x {100};

Void func(int &&a);

Func(200);

Func (x); //x is an l-value

Error: Cannot bind r value reference of type int && to an lvalue of type int.

Combined solution:

Int x {100};

Func(int &a);

Func( int &&a);

//calling func

Func(x);

Func(200);

**Note:** References cannot refer to constant value

Note: We can create a constant reference that refers to a constant. For example, the following program compiles and runs fine.

#include<iostream>

using namespace std;

int main()

{

const int x = 10;

const int& ref = x;

cout << ref;

return 0;

}

When you first learn about them, move semantics and perfect forwarding seem pretty straightforward:

• **Move semantics** makes it possible for compilers to replace expensive copying operations with less expensive moves. In the same way that copy constructors and copy assignment operators give you control over what it means to copy objects, move constructors and move assignment operators offer control over the semantics of moving. Move semantics also enables the creation of move-only types, such as std::unique\_ptr, std::future, and std::thread.

• **Perfect forwarding** makes it possible to write function templates that take arbi‐ trary arguments and forward them to other functions such that the target func‐ tions receive exactly the same arguments as were passed to the forwarding functions. Rvalue references are the glue that ties these two rather disparate features together.

Rvalue references are the glue that ties these two rather disparate features together. They’re the underlying language mechanism that makes both move semantics and perfect forwarding possible:

*template // in namespace std*

*typename remove\_reference::type&&*

*move(T&& param) {*

*using ReturnType = typename remove\_reference::type&&; // // alias declaration see Item 9*

*return static\_cast(param); }*

std::move and std::forward are merely functions (actually function templates) that perform casts. std::move unconditionally casts its argument to an rvalue, while std::forward performs this cast only if a particular condition is fulfilled. That’s it

**Copy Constructor:**  Copy constructor is a special constructor for class/struct used to create a copy of an existing instance , copy constructor syntax is one of the following, for example for Myclass:

• Myclass(const Myclass& other)

• Myclass(Myclass & other)

• Myclass(volatile const Myclass & other)

• Myclass(volatile Myclass& other)

**Note**: Below constructors also do same thing as the copy constructors, but they are not copy constructors:

a) Myclass(Myclass\* other)

b) Myclass(const Myclass\* other)

**Note:** If we don’t create a copy constructor, Compiler creates a copy constructor implicitly. For example below:

*Myclass {*

*public:*

*int a;*

*float b;*

*std::string s;*

*}*

The compiler provided copy constructor is exactly equivalent to:

*Myclass::Myclass(Myclass& other):*

*a (other.a), b(other.b), s(other.s)*

*{}*

There are certain circumstances where like above member copy is not sufficient. By far, most common reason the default copy constructor is not sufficient is because the Object contains raw pointers and you need to take a Deep copy of the pointer, i.e you don’t copy the constructor, rather you have to copy the what pointer points to.

Why do you need deep copy, because the instance owns the pointer, because instance is responsible for calling the delete on the pointer at the same point(at destructor). If 2 objects end up calling delete on the same non-Null pointer, heap corruption results.

Rarely you will come across a class that does not contain raw

pointers yet the default copy constructor is not sufficient.

An example of this is when you have a reference-counted object.

boost::shared\_ptr<> is example.

**Const Correctness:** When passing parameters by references to a functions or constructor as arguments. Pass by non-const reference only

If the function will modify the parameter and it is the intent to change the caller's copy of the data, otherwise pass by const reference.

Why is this so important? There is a small clause in the C++ standard that says that non-const references cannot bind to temporary objects.

A temporary object is an instance of an object that does not have a variable name. For example:

Temporary object ex: std::string("Hello World") //No variable name is declared.

Non temporary object: std::string s("Hello World")

Ex:

***// Improperly declared function: parameter should be const reference:  
 void print\_me\_bad( std::string& s ) {  
 std::cout << s << std::endl;  
 }***

***// Properly declared function: function has no intent to modify s:  
 void print\_me\_good( const std::string& s ) {  
 std::cout << s << std::endl;  
 }***

***std::string hello( "Hello" );***

***print\_me\_bad( hello ); // Compiles ok; hello is not a temporary  
print\_me\_bad( std::string( "World" ) ); // Compile error; temporary object  
print\_me\_bad( "!" ); // Compile error; compiler wants to construct temporary  
 // std::string from const char\****

***print\_me\_good( hello ); // Compiles ok  
print\_me\_good( std::string( "World" ) ); // Compiles ok  
print\_me\_good( "!" ); // Compiles ok***

**Note:** Many of the STL containers and algorithms require that an object

be copyable. Typically, this means that you need to have the

copy constructor that takes a const reference, for the above

reasons.

**Assignment Operator Overloading:**

if you do not declare an assignment operator, the compiler gives you one implicitly.

The implicit assignment operator does member-wise assignment of  each data member from the source object. For example, using the class above, the compiler-provided assignment operator is exactly equivalent to:

MyClass& MyClass::operator=( const MyClass& other ) {  
 x = other.x;  
 c = other.c;  
 s = other.s;  
 return \*this;  
 }

In general, any time you need to write your own custom copy  constructor, you also need to write a custom assignment operator.

**Operator Overloading:**

• Compiler provides a default "assignment(=)" operator overloading, not for any other operator.

**•**

• The Majority of the c++ operators can be overloaded , except the below:

:: (Scope operator)

.\*

:?

.

Sizeof

**Assignment Operator Overloading: (=)**

• C++ Provides a default assignment operator used for assigning one object to another object, it means it provides a "Shallow Copy"

MyString ms1 {"Frank"};

MyString ms2 = ms1 // It’s not assignment , here Copy constructor is called. //its same as "MyString ms2{ms1};"

MyString ms3;

Ms3 = ms1; //Here Assignment operator.

Example:

**class** MyString{

**private**:

**char** \*s;

**public**:

    MyString();

    MyString(**const char**\* s1);

    MyString(**const** MyString& source);

**void** display() **const**;

**int** get\_length() **const**;

**const char**\* get\_str() **const**;

    ~MyString();

};

• If we have raw pointer data member we must do **deep Copy**.

• Overloading **Copy Assignment** Operator (deep copy):

Syntax:

Type &Type::operator=(const Type &rhs);

MyString &MyString::operator=(const MyString &rhs);

S2 = s1;

S2.operator=(s1); //operator= method is called.

S2 = "shankar"; //Temporary object is created and copy assigment operator is called.

• Example for the above class: (Deep Copy): **Copy Assigment**

MyString &MyString::operator=(const MyString& rhs){

if(this = &rhs){ //**Check for Self assignment for the cases like p1 = p1;**

return \*this;

};

delete[] s; //Deallocate Storage for this->s "since we are overriding it";

s = new char[ strlen(rhs.s) +1]; //Allocate Storage for new data memeber

strcpy(s,rhs.s);

return \*this;

}

5) **Move Assignment** Operator Overloading, (move assignment operator =)

a) You can chose to overload the move assignment operator

b) C++ uses the copy assignment operator if necessary

MyString s1;

s1 = MyString {"MoveCopyconstructor"};

Mystring s2 {"infosys limited"};

MyString s3 = std::move(s1); //Move constructor is called.

c) **If we have raw pointer, we should overload the move assignment operator for efficiency.**

Syntax :

MyString s1;

Type &Type::**operator=**(Type **&&rhs**) ;

s1 = MyString("MoveAssignment"); Move operator= is called

s1 = "Frank"; //Move operator= is called.

MyString &MyString::operator=(const MyString&& rhs){

if(this = &rhs){ //**Check for Self assignment for the cases like p1 = p1;**

return \*this;

};

delete[] s; //Deallocate Storage for this->s "since we are overriding it";

s = rhs.s // ***Steal the pointer from the Right Side Object and assign it to this->s***

rhs.s = nullptr //Null out the rhs pointer

return \*this; //Return the current object by reference to allow chain assignment.

}

**Note: If we don’t define the Move Assignment Operator overloading, Compiler will call Copy Assignment Operator during below events:**

**MyString s1 {"hello"};**

**s1 = MyString{"Bye"};**

**s1 = "How Are you";**

**Unary Operators as member methods(++,--, -, !):**

Return type::operatorOp();

Number Number::operator-();

Number Number::operator++(); //Preincrement

Number Number::operator++(int) // post increment

Boo Number::operator!() const;

Number n1 {100};

Number n2 = -n1;

N2 = ++n1;

N2 = n1++;

Binary Operators as member methods(+,-, ==, !=, <,>..etc)

Number Number::operator+(const &Number rhs) const;

Number Number::operator+(const &Number rhs) const;

Number Number::operator==(const &Number rhs) const;

Number Number::operator<(const &Number rhs) const;

Number n1 {100}, n2 {200};

Number n3 = n1+n2;

N3 = n1-n2;

If( n1 == n2);

**Operator overloading Member Methods example:**

#include "Money.h"

Money::Money(int dollars, int cents) : dollars{dollars}, cents{cents} {}

Money::Money(int total) : dollars {total/100}, cents{total%100} {}

bool Money::operator==( const Money& m2){

if (this->dollars == m2.dollars && this->cents == m2.cents){

return true;

}else{

return false;

}

}

bool Money::operator!=(const Money& m1){

if (this->dollars != m1.dollars && this->cents != m1.cents){

return true;

}else{

return false;

}

}

**Note:** The **this** pointer holds the address of current object, in simple words you can say that this **pointer** points to the current object of the class. Let’s take an example to understand this concept.

**Exception handling and Exception Safe code:**

• A function which modifies some global state is said to be "exception safe", if it leaves the global state well defined in the event of an exception that happened at any point during the function execution.

**Exception:**

Defining a customized exception:

***#include <i`a1m2***

***>***

***#include <string>***

***#include <sstream>***

***#include <exception>***

***using namespace std;***

***/\* Define the exception here \*/***

***class BadLengthException: public std::exception***

***{***

***private:***

***string l;***

***public:***

***BadLengthException(int len):l(to\_string(len)){***

***}***

***char const \* what() const throw() {***

***return l.c\_str();***

***}***

***~BadLengthException() throw() {}***

***};***

***bool checkUsername(string username) {***

***bool isValid = true;***

***int n = username.length();***

***if(n < 5) {***

***throw BadLengthException(n);***

***}***

***for(int i = 0; i < n-1; i++) {***

***if(username[i] == 'w' && username[i+1] == 'w') {***

***isValid = false;***

***}***

***}***

***return isValid;***

***}***

***int main() {***

***int T; cin >> T;***

***while(T--) {***

***string username;***

***cin >> username;***

***try {***

***bool isValid = checkUsername(username);***

***if(isValid) {***

***cout << "Valid" << '\n';***

***} else {***

***cout << "Invalid" << '\n';***

***}***

***} catch (BadLengthException e) {***

***cout << "Too short: " << e.what() << '\n';***

***}***

***}***

***return 0;***

**}**

**The C++ Casting Operators:**

a) static\_cast

b) dynamic\_cast

c) reinterpret\_cast

d) const\_cast

**Static Cast:** is a mechanism that can be used to

a) convert pointers b/w the related types

b) perform explicit type conversions for standard data types that would otherwise happen automatically or implicitly.

c) as far as pointers go, static\_cast implements a basic compile time check to ensure that pointers is being cast to related type.

d) Using static\_cast, a pointer can be upcasted to the base type, or can be down casted to the derived type

ex:

Base \*base = new Derived();

Derived \*derived = static\_cast<Derived\*>(base);

Derived derived;

Base \*base = &derived;

**Containers:**

• A container is a holder object that stores the collection of other objects. They are implemented as templates, which allows great flexibility in the types supported as elements.

• The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators.

• Containers replicate structures very commonly used in the programming

• Vectors (Dynamic Arrays)

• Queues (Queue)

• Stacks (Stack)

• Heaps (Priority Queues)

• list(Linkedlist)

• Set (trees)

• Associate arrays(Map)

**Sequence Containers:**  Array, Vector, Deque(double ended queue), forward\_list, list(List)

**Container Adaptors**: stack, queue, priority\_queue

**Associative Containers:** Set, Multiset (Multipl-key set), Map, multimap(Multiple Key Map)

**Unordered Associate Containers:** unorderd\_set, unordered\_multiset, unordered\_map, unordered\_multimap.

**Other:** Two class templates that share certain properties with containers, and are sometimes classified with them: **bitset, and valarray.**

**Queue:**

FIFO or LILO: The methods are

Pop(removes the first element)

Front( access the first element)

Back(access the last element in the queue)

Push(inserts the element at the end)

Size(size of the queue)

Empty(to check whether its empty or not)

Emplace( Construct and inserts element)

Swap (swap contents of 2 queues).

**Stack:**

Below program causes issues:

**FILO or LIFO: Methods are:**

top(),

pop(),

emplace(),

empty(),

size(),

push(),

swap().

**Priority Queue:**

Priority queues are a type of container adapters, specifically designed such that the first element of the queue is the greatest of all elements in the queue and elements are in non-decreasing order(hence we can see that each element of the queue has a priority{fixed order}).

Here are also same as above methods mostly

**Vectors: (Sequence Container):**

vectors use a dynamically allocated array to store their elements.

Note:

• Addition of the elements to the end of array in **constant time, i.e time** needed to insert at the end of array is not dependent on the size of the array. Same for removal of elements.

• The time needed to insert or remove the elements in the middle of the array is directly proportional to the no. of elements behind the element is being removed.

• The no. of elements held is dynamic and the vector class manages the memory usage.

**Member functions:**  constructor, destructor , operator=

**Iterators**: begin, end, rbegin, rend, cbegin(return constant iterator to the beginning), cend, crbegin, crend

**Capacity**: size, empty, resize, max\_size,capacity(return size of allocated storage capacity), reserve(request a change in Capacity)

Resize: will increase the size and set the all values to defaults in the vector, it will affect the size, it will have impact on the capacity also.

Reserve: will allocate the memory, but it will not have impact on size or but leaves it uninitialized, it will not set all the elements to the defaults. It will only have impact.

The resize() method (and passing argument to constructor is equivalent to that) will insert or delete appropriate number of elements to the vector to make it given size (it has optional second argument to specify their value). It will affect the size(), iteration will go over all those elements, push\_back will insert after them and you can directly access them using the operator[].

The reserve() method only allocates memory, but leaves it uninitialized. It only affects capacity(), but size() will be unchanged. There is no value for the objects, because nothing is added to the vector. If you then insert the elements, no reallocation will happen, because it was done in advance, but that's the only effect.

So it depends on what you want. If you want an array of 1000 default items, use resize(). If you want an array to which you expect to insert 1000 items and want to avoid a couple of allocations, use reserve()

The following code uses emplace\_back to append an object of type President to a std::vector. It demonstrates how emplace\_back forwards parameters to the President constructor and shows how using emplace\_back avoids the extra copy or move operation required when using push\_back.

**Declaration and Initialization:**

• Vector<int> vint(10); //instantiate a vector with 10 elements.

• Vector<int> vint(10,90); //instantiate a vector with 10 elements with (initializes) each of the value as 90.

• Vector<int> vint2(vint); //instantiating a vector and initializes with content of another vector.

• Vector<int> vint3(vint.begin(), vint.begin()+5); //Using iterators instantiating with 5 elements of another vector

**Initializer list:**

Vector<int> v1 = {1,2,3};

Vector<int> v2 {1,2,3};

**Inserting telements at a Given position using insert:**

• Vint.insert(vint.begin(),25); // Inserting an element at the beginning of the vector

• Vint.insert(vint.end(),2,40);//inserting 2 numbers of value 40 at the end of vector.

• Insert the contents of one vector into another vector at a chose position.

Vector<int> vint (2,30); // it has 2 elements whose value is 30.

Vint1.insert(vint1.begin()+1, vint.begin(), vint.end());

• You can even use the “find” method in the STL to find out an element and using that iterator , insert an element at the position.

**Note:** If the container is using frequent insertions, don’t use the vector instead , use a list.

**Note**: The Preemptive increase in the capacity of the vector when the internal buffer is reallocated is not regulated by any clause in the standard. This is dependent on the flavor of the STL being used.

Usually, initially a capacity of 8 elements will be allocated, after 9th element insertion, capacity will change to 16 automatically.

**Accessing elements:**

• Use the semantics [], with an index value in it to access an element at a particular place.

• Better to use .**at** function, as it throws the exception, if we are trying to access an element beyond the size of the vector. As it first checks the size and then try to access the element at that position.

**Removing elements from the vector:**  pop\_back()

*From <https://en.cppreference.com/w/cpp/container/vector/emplace\_back>*

**Modifiers:**

Assign: Assign a vector content

Push\_back: insert a element at the end

Pop\_back: delete an element at the end

Insert: insert elements

Erase: Erace elements

Clear: clear content

Swap: Swap contents

Emplace: Construct and insert an element

Emplace\_back: Construct and insert an element at the end.

**Element Access:**

At []

Front()

Back()

Operator[] //operator access element

Data //Access data (in c++11)

Note: The following code uses emplace\_back to append an object of type President to a std::vector. It demonstrates how emplace\_back forwards parameters to the President constr

#include <vector>

#include <string>

#include <iostream>

struct President

{

std::string name;

std::string country;

int year;

President(std::string p\_name, std::string p\_country, int p\_year)

: name(std::move(p\_name)), country(std::move(p\_country)), year(p\_year)

{

std::cout << "I am being constructed.\n";

}

President(President&& other)

: name(std::move(other.name)), country(std::move(other.country)), year(other.year)

{

std::cout << "I am being moved.\n";

}

President& operator=(const President& other) = default;

};

int main()

{

std::vector<President> elections;

std::cout << "emplace\_back:\n";

elections.emplace\_back("Nelson Mandela", "South Africa", 1994);

std::vector<President> reElections;

std::cout << "\npush\_back:\n";

reElections.push\_back(President("Franklin Delano Roosevelt", "the USA", 1936));

std::cout << "\nContents:\n";

for (President const& president: elections) {

std::cout << president.name << " was elected president of "

<< president.country << " in " << president.year << ".\n";

}

for (President const& president: reElections) {

std::cout << president.name << " was re-elected president of "

<< president.country << " in " << president.year << ".\n";

}

}

Output:

emplace\_back:

I am being constructed.

push\_back:

I am being constructed.

I am being moved.

Contents:

Nelson Mandela was elected president of South Africa in 1994.

Franklin Delano Roosevelt was re-elected president of the USA in 1936.

**Exception handling:**

What causes exception:

• Insufficient resources

• Missing resources.

• Invalid operations.

• Range violations.

• Underflows and overflows

• Illegal data and many others.

**Keys Words:**

a) throw:

throws an exception.

followed by an argument.

b) try (code that may throw exception):

if your code throws an exception the try block is exited.

The thrown exception is handled by a catch handlder.

If no catch handler exists the program terminates.

c) Catch(Exception ex) (code used to handle the exception)

i) Code can have multiple exception.

ii) may or may not cause the program to terminate.

**Note:** floating point division in c++ , gives the result as “inf”(infinity).

float a = 10.0; int b = 0, int d = 12;

int c = a/b; or int c = static\_cast<double>(b)/b;

//Program wont crash, instead it gives the output as “inf.”

**Note: When your code handles exceptions, it is said to be exception safe.**

**• Keywords: try, catch , throw(throws an exception, followed by an argument)**

**• Difference between <static\_cast> and regular typecasting.**

**User Defined Exceptions:**

We can create exception classes and throw instances of those classes.

Best Practice:

a) throw an object not a primitive.

b) throw an object by value.

c) Catch an object by reference (or const reference).

**Class Level Exceptions:**

Exceptions can also be thrown from within a class

**a) Destructor: Do NOT throw exceptions from your destructor , In C++, destructor is marked as “NO EXCEPT by default” it means that it cannot throw exception.**

The **C++** rule is that you must never **throw** an **exception** from a **destructor** that is being called during the "stack unwinding" process of another **exception**. ... So the **C++** language guarantees that it will call terminate() at this point, and terminate() kills the process.

If an **exception** is thrown and **not caught** anywhere, the program terminates abnormally

**b) Constructor:** Constructor may fail(suppose it fails to initialize variables, or allocate memory, or unable to open a file)

**C++ Standard Exception class hierarchy:**

C++ Provides a class hierarchy of exception classes

a) std::exception is the base class.

b) all base classes implement **the what() Virtual** functions.

c) We can create our own user-defined exception sub classes.

“virtual const char \* what() const noexcept;

**Ex:**

Class illegalBalanceException : public std::exception {

Public:

illegalBalanceException() **noexcept** = default;

~illegalBalanceException() = default;

Virtual const char\* what() const **noexcept** {

Return “Illegal Balance Exception”;

}

}

Note : “**noexcept” Key word is used to indicate , these method or constructor cannot throw exception**

**Notes:**

• Empty parameter list in C mean that the parameter list is not specified and function can be called with any parameters. In C, it is not a good idea to declare a function like fun(). To declare a function that can only be called without any parameter, we should use “void fun(void)”.

• As a side note, in C++, empty list means function can only be called without any parameter. In C++, both void fun() and void fun(void) are same.

• If in a C program, a function is called before its declaration then the C compiler automatically assumes the declaration of that function in the following way:

int function name();

And in that case if the return type of that function is different than INT ,compiler would show an error.

•  In c three continuous dots is known as ellipsis which is variable number of arguments of function. The values to parameters are assigned one by one. Now the question is how to access other arguments

**Function pointer:**

This is a simple program with function pointers. fun is assigned to point to demo. So the two statements "(\*fun)();" and "fun();" mean the same thing.

#include <stdio.h>

**int** main()

{

**void** demo();

**void** (\*fun)();

    fun = demo;

    (\*fun)();

    fun();

**return** 0;

}

**void** demo()

{

**printf**("GeeksQuiz ");

}

**Strings:**

• Character functions.

• c-Styled Strings.

• Working with c-styled strings.

• C++ Strings && working with c++ styled strings.

**Character functions:**

• #include <cctype> (header file).

• Function\_name(char) //Expects only a single character.

• Functions for testing character or converting character case.

Ex:

Testing character:

isalpha(d); isalpnum(d); isupper(d), islower(); isprint(d), ispunt(d)(if d is a punctuation character), isspace(d)(True if d is a whitespace)

isdigit(d);

Converting case:

a) tolower(d), toupper(d)

**c\_str():** Returns a pointer to an array that contains a null terminated sequence of characters(i.e a c string) representing the current value of the basic\_string.

**C-Styled Strings:**

Sequence of characters:

a) Contiguous in memory.

b) implemented as an array of characters.

c) terminated by a null character( null -- character with a value zero)

d) Referred to as zero or null terminated strings.

String Literal:

• Sequence of characters in double quotes "shankar"

• Constant.

• Terminated with null character.

Compiler inserts a null character at the end of the string literal.

Declaring c-styled strings: (are included in the header #include <cstring>

• Char my\_name[] {"shankar1234567891234"};

2) Below program is wrong.

char name[8];

name = "shankar"; //Error.

strcpy (name, "Shankar"); //Correct

3) Define const string:

Const char\* constString = “this is a constant string”;

//This is will work

Std::string s1(constString)//We can pass Const string while instatiating new string

//Initialize String to 1st few chars of const string:

Std::string s2(constString, 5);

Examples:

#include <cstring>

Char ch[80];

Strcpy (ch, "shankar");

Strcat (cg, "there");

Cout << strlen(str) << endl; // writes 11, it does not count the null terminated character.

Strcmp( str, "another") //return -1;

Functions to covert

#include <cstdlib> //General purpose functions

Includes functions to covert C-Style Strings to

• Interger.

• Float

• Long.

• Etc…

Note: "Cin" won't the read the complete input if the input has a space in b/w. So use getline for this.

C++ Strings:

• Std::string is a class in the Standard Template Library

i) #include <string>

ii) they are defined in the std namespace, without including the std namespace in the source file, string has to be prefixed with std

iii) Contiguous in memory.

iv) dynamic size.

v) work with input and output streams.

vi) Lots of useful member functions.

vii) Our familiar operators can be used ( +, = , ==, <, >, <=, >=, !=, []

viii) can be easily converted to C-Styled Strings.

ix) Safer.

Declaring and Initializing:

• #include <string>

Using namespace std;

Int main(){

String s1 {"shankar"};

String s2 {s1};

String s3 (3,'x') ; xxx

String s4 {"Shankar", 3}; // Sha

String s5 {s1, 0, 2}; //sh

Assignment , s2 = s1;

Concatenation , //string s3 = s1 + "Shankar" + s2;

Append , s1.append(s2)// Here S1 will be modified new string (s1+s2).

Accessing characters // s1[0] or s1.at(0);

With for loop;

Ex1:

String s {"shankar"};

For (char c; s){

Cout << c << endl;

S

H

A

N

K

A

R

Ex2:

String s {"shankar"};

For (int c; s){

Cout << c << endl;

70

114

65

Xx

Xx

Xx

0 // null character

Comparing the strings:

String s1 {"Shankar"};

S1 == s2;

S1 < s2;

S1 > s2;

S1 == "Apple"

S1 != s2;

**Substrings:**

(Extract sub string from a string)

Syntax: **substr**(x, y) ; // x is staring index to start, y is length of string staring from index x;

String s1 {"shankar goud vemula"};

S1.substr(0,2); //sh

S1. substr(5, 7); //ar

**Find(search\_string) :** Returns the index of a sub string in the std::string;

String s1 {"this is a test"};

Cout << s1.find("this") << endl; // 0

Cout << s1.find('a') << endl; // 8

Cout << s1.find("test") << endl;//10

Cout << s1.find("was") << endl; // -1, string::npos

Ex: finding recurrence of a char/word in a string:

Size\_t strops = s1.find(“day”,0);

While(strops != string::npos){

Cout << “day is found at position” << strops << endl;

Strops = s1.find(“day”, strops+1);

};

**Erase and clear :**  Remove a substring of characters with a length from a std::string

String s1 {"this is shankar"};

S1.erase(3,4); //thi shankar

S1.clear() ; //empty string

Length() // To find length of string.

Input (>>) and getline() //Reading std::string from cin

String s1 ;

Cin >> s1; //ip: This is Shankar, output: //This (it does accept the beyond the white space)

Getline(cin, s1); //this is Shankar //It stops reading until \n or new line found

Cout << s1 << endl;//this is shankar

Getline(cin, s1, 'x') ; //This is shankarx //Get line stops reading at 'x'

Cout << s1 << endl; // This is shankar

If you try to use mix of the both getline(cin,s1) or cin >> s1, it will cause issues, it will hang for "cin>> s1", instead use completely getline in the code.

**Reading a Transcoder file example:**

#include <iostream>

#include <sstream>

#include <`

#include <string>

using namespace std;

int main() {

ifstream f("/Users/svemul200/Desktop/Test/SampleCSVFile\_2kb.csv",ios::binary);

if(!f){

cout << "There is an issue in opening the file" << endl;

}

string line;

while(getline(f,line)){

cout << "Each line is: " << line << endl;

stringstream s(line);

string elm;

while(getline(s,elm,',')){ //Splitting a line based on a delimter, 3rd arg is delimiter.

cout << elm << " : ";

}

cout << endl;

}

return 0;

}

**Classes:**

a) Creating a class ex:

class Player{

int score;

std::string name;

void calculate();

void profile();

}

Player p1;

Player p2;

Player \*p3 = new Player();( Memory allocation in the heap and created dynamically);

delete p3;(Memory is freed up using the delete key word).

Assert function is used to assert that condition is satisfied, if it does not, program will terminate there itself.

Ex:

#include <stdio.h>

#include <assert.h>

int main() {

int a, b;

printf("Input two integers to divide\n");

scanf("%d%d", &a, &b);

assert(b != 0);

printf("%d/%d = %.2f\n", a, b, a/(float)b);

return 0;

}

Output:

Input two integers to divide

5 0

a.out: assrt.cpp:10: int main(): Assertion `b != 0' failed.

Aborted (core dumped)

**Access Modifiers:**

• Private: Compilation error , if we try to access private varialbes outside the class

• Public

• Protected: Can be accessed only by the inherited classes.

**Implementing Member methods:**

• Member methods have access to member attributes.

• Can be implemented inside the class declaration.(implicitly inline)

• Can be implemented outside the class declaration ( "Classname::method\_name")

• Can separate specification from the implementation ( .h file for class declaration, .cpp file for class implementation).

**Note**: If a class header file(.h file) is included in more than one file, then compiler looks for the definitions in multiple cpp files, this will give an error as duplicate definitions, to resolve this an include guard is defined. If we include this guard , compiler will process it only once, no matter how many times, this file is included in many files.

Ex: Include Guards or Preprocessor Guards

Account.h file

#ifndef \_ACCOUNT\_H\_

#define \_ACCOUNT\_H\_

< Account class declaration>

#endif

• Angle Brackets includes (ex: #include <iostream>) is system header files, wheras other "" includes local project files.

**Account.h file**

*#include <iostream>*

*using namespace std;*

*class Account {*

*private:*

*int bal;*

*public:*

*void setBalance(int b);*

*int getBalance();*

*};*

*Account.cpp file:*

*#include <iostream>*

*#include "Account.h>*

*Void Account::setBalance(int b){*

*bal = b;*

*};*

*Int Account::getBalance(){*

*return bal;*

*}*

**NOTE:** Never include .cpp file in the main file, always include .h or header file.

Main.cpp

*#include <iostream>*

*#include "Account.h>*

g++ -c Account.cpp -o Account.o

g++ Acc.cpp Account.o -std=c++11

**Note**:

17

unsigned char\* is basically a byte array and should be used to represent raw data rather than a string generally. A unicode string would be represented as wchar\_t\*

According to the C++ standard a reinterpret\_cast between unsigned char\* and char\* is safe as they are the same size and have the same construction and constraints. I try to avoid reintrepret\_cast even more so than const\_cast in general.

If static cast fails with what you are doing you may want to reconsider your design because frankly if you are using C++ you may want to take advantage of what the "plus plus" part offers and use string classes and STL (aka std::basic\_string might work better for you)

*From <https://stackoverflow.com/questions/658913/c-style-cast-from-unsigned-char-to-const-char/658915>*

Compiling boost::thread code:

sudo g++ bthrd.cpp -std=c++11 -lboost\_thread -lboost\_system

**Flush stream buffer**

Synchronizes the associated *stream buffer* with its controlled output sequence.

For *stream buffer* objects that implement intermediate buffers, this function requests all characters to be written to the controlled sequence.

Its behavior is equivalent to calling os's member function flush.

A member function with the same name and behavior exists (see ostream::flush).

*From <http://www.cplusplus.com/reference/ostream/flush-free/>*

**Function Objects:**

1) On a conceptual level , function objects are objects that work as functions.

2) On implementation level, fun. Objects are objects of a class that implement operator().

3) Although functions and function pointers can also be classified as function objects, it is the capability of an object of a class that implements an operator() to carry state(i.e , values in member attributes of the class) that makes it useful with STL algorithms.

4) Functions objects typically used by C++ programmer (working with STL) classified into following types:

a) Unary Function: A function called with one arg. Ex: f(x), when unary function returns bool, its called as Unary Predicate.

b) Binary Function: A function called with 2 args. Ex: f(x,y), when a binary function returns bool, its called as Binary Predicate.

\*\*A function object that combines 2 fun objects is called an **“Adaptive function object”.**

**Ex:**

Template <typename T>

Void fundispayElement(const T& t){

Cout << element << “ “;

}

The above function can also be represented as with different implementation, in which function is actually contained by the operator() of a class or struct.

Template <typename t>

Struct DisplayElement{

void operator()(const t& t1){

cout << t1 << ‘ ’ << endl;

}

}

Or

Template <typename t>

Class Display{

Public:

Operator(){

Cout << t1 << “ ”<< endl;

};

}

**Usage in algorithms:**

a) For\_each(vec.begin(), vec.end(), DisplayElement<int>()); //Unary function object

b) the above same thing can be called using the “fundispayElement” like

for\_each(vec.begin(),vec.end(), fundispayElement.

c) The real advantage of using a function object implemented in a struct become apparent when you are able to use the object of the struct to store information. This is something “fundispayElement” cannot do the way a struct can because a struct can have member attributes other than operator(). Sightlty modified version

template <typename elementType>

struct DisplayElement{

int count;

DisplayElement(){ count=0};

operator()(const elementType& ele){

++count;

}

**Note: C++11 introduces lamda expressions that are unnamed function objects. We can write compact code like:**

for\_each(v1.begin(), v1.end(), [](int& e){cout << e << “ ”;});

**//Unary Predicate**: A unary function that returns a bool is a predicate. . Such functions help make decisions for STL algorithms

Ex:

Template <typename T>

Struct IsMultiple{

T t1;

IsMultiple(const **T& t2):t1(t2){};**

**bool operator()(const T& e) const{**

**return (e%t1) == 0;**

**}**

**Unary Predicate Usage:**

auto iElement = find\_if(v1.begin(),v1.end(),IsMutliple<int>(div));

Unary predicates find application in a lot of STL algorithms such as **std::partition** that can partition a range using a predicate, stable\_partition that does same while keeping relative order of the elements partitioned, find funtions such as find\_if and functions that help erase elements such as std::remove\_if that erase elements in the range that satisfy the predicate.

**Binary Functions:**

Binary functions are used in algorithms such as **std::transform** where you can use to multiply the contents of 2 containers.

template <typename elementType>

class Multiply{

elementType operator()(const elementType& e1, const elementType& e2){

return e1\*e2;

}

Example of multiple 2 vectors (v1 and v2 of ints) and writing the result to v3.

*transform(v1.begin(),v1.end(),v2.begin(), v3.begin(), Multiply<int>(3));*

**Binary Predicate:**

A function which accepts 2 args and return a bool is a binary predicate. Such functions find application in STL algorithms such as std::sort

ex:

*class CmpStringNoCase{*

*public:*

*bool operator()(const string& s1, const string& s2){*

*string s3, s4;*

*//Here Resizing is very important, without that Soring will not be done.*

*s3.resize(s1.size());*

*s4.resize(s2.size());*

*transform(s1.begin(),s1.end(),s3.begin(), tolower()); //converting s1 to lower case and writing to s3;*

*transform(s2.begin(),s2.end(),s4.begin(), tolower()); //converting s2 to lower case and writing to s4;*

*return (s3 <s4);*

*}*

*}*

**High Lights:**

• Predicates are special type of function objects, they always return true.

• for remove\_if, we can use a Unary Predicate which will take a value to be processed as the intial state via the constructor.

• for map, we can use a binary predicate.

**Lamdas: (are anonymus function objects)**

• Lamdas are such an easy way to crate function objects

• Without lamdas, the STL \_if algorithms (e.g st::find\_if, std::remove\_if, std::count\_if .. Etc) tend to be employed with the only most trivial predicates, but when the lamdas are available , use of these algorithms with non-trivial algorithms blossoms.

• Outside the STL, lamdas make it possible to quickly create custom deleters for std::unique\_ptr, and std::shared\_ptr, and they make the specification of predicates for the condition variables in the threading API equally straight forward.

4) Beyond the Standard Library, lambdas facilitate the on-the-fly specification of callback functions, interface adaption functions, and context-specific

functions for one-off calls.

A lamda expression:

std::find\_if(container.begin(), container.end(), **[](int val) { return 0 < val && val < 10; }**);

• A closure is the runtime object created by a lamda. Depending upon the capture mode, closures hold copies or references to the captured data, in the call to the std::find\_if above, closure is the object that’s passed at the runtime as a third argument to the find\_if function

• A closure class is a class from which a closure is instantiated. Each lambda causes compilers to generate a unique closure class. The statements inside a lambda become executable instructions in the member functions of its closure class.

• However, closures may gen‐ erally be copied, so it’s usually possible to have multiple closures of a closure type corresponding to a single lambda. For example, in the following code.

{ int x

; // x is local variable …

auto c1 = [x](int y) { return x \* y > 55; }; // closure produced // by the lambda , // c1 is copy of the

auto c2 = c1; // c2 is copy of c1

auto c3 = c2; // c3 is copy of c2 … }

c1, c2, and c3 are all copies of the closure produced by the lambda.

**Lamda Syntax:**

[Captured variables](paameters) { function code }

Local variables from outer scope can be captured inside Lambda in 2 modes i.e.

• By Value

• By Reference

**Capturing Local Variables by value inside Lambda Function:**

// Local Variables

std::string msg = "Hello";

int counter = 10;

// Defining Lambda function and

// Capturing Local variables by Value

auto func = [msg, counter] () {

//...

};

**Note**: Now, the variables specified in capture list will be copied inside lambda by value. Inside lambda they can be accessed but can not be changed, because they are const.

To modify the we need to add mutable keyword i.e.

auto func = [msg, counter] () mutable { };

**Capturing Local Variables by Reference inside Lambda**

**immediately executed lambda expression:**

[&](){ ...your code... }(); // immediately executed lambda expression

is functionally equivalent to

{ ...your code... }

**Task Based Vs Thread Based approaches in the C++11:**

If you want to run a function asynchronously , you have 2 choices:

• Thread based approach:

doasyncjob();

std::thread t(doasyncjob)

b) Task based approach:

auto func = std::async(doAsyncWork); //the function object passed to std::async (e.g., doAsyncWork) is con‐ sidered a task. The task-based approach is typically

The task-based approach is typically superior to its thread-based counterpart, and the tiny amount of code we’ve seen already demonstrates some reasons why. Here, doAsyncWork produces a return value, which we can reasonably assume the code invoking doAsyncWork is interested in.

With the task-based approach, it’s easy, because the future returned from std::async offers the get function. The get func‐ tion is even more important if doAsyncWork emits an exception, because get pro‐ vides access to that, too. With the thread-based approach, if doAsyncWork throws, the program dies (via a call to std::terminate).

A more fundamental difference between thread-based and task-based programming is the higher level of abstraction that task-based embodies. It frees you from the details of thread management, an observation that reminds me that I need to summa‐ rize the three meanings of “thread” in concurrent C++ software:

• Hardware threads: are the threads that actually perform computation. Contempo‐ rary machine architectures offer one or more hardware threads per CPU core.

• • Software threads (also known as OS threads or system threads) are the threads that the operating system1 manages across all processes and schedules for execu‐ tion on hardware threads. It’s typically possible to create more software threads than hardware threads, because when a software thread is blocked (e.g., on I/O or waiting for a mutex or condition variable), throughput can be improved by exe‐ cuting other, unblocked, threads.

**Inheritance:**

• Public inheritance.(the is-a relationship is only applicable in public inheritance

• Private Inheritance.

• Protected inheritance.

Sub classes can access the both (public and protected) variables and member functions of super/base class.

Syntax for inheritance:

Class Fish{

};

Classs Tuna: <access specifier> Base {

}

With public inheritance , the public varialbes inside the base class can be modified in the main method also.

To make certain variables accessible by only the derived class, but not accessible outside the world. This is where the protected inheritance comes into the picture.

--> When you declare an attribute as protected , you can effectively making the attribute accessible to the classes that derive and friends. Yet simultaneously making it inaccessible to everyone else outside, including main.

This is a very important aspect of the OOP, combining data abstraction and inheritance, in ensuring that derived classes can safely inherit base class attributes that cannot be tampered with by anyone outside the hierarchical system.

**Base class Initialization:**  What if a base class contains an overloaded constructor that enforces the arguments at the time of initialization, how would such a base class is instantiated when derived class is being constructed.

The clue lies in using the initialization lists and invoking an appropriate base class constructor via the constructor of a derived class.

With this, derived class never access a base class variable directly in spite of being it at as protected member variable, so ti can be initialized through constructor of base class.

To ensure max. security, if the derived class don’t need to access base class attribute, remember to mark it as a private.

Ex:

Class Base{

public:

Base(int somenum){

}

};

Class Derived: public Base{

Public:

Derived(): Base(25){

}

//To call a method(from main method) , which is overridden by the subclass with subclass instance.

subclass.Mainclass::mainclass\_method;

//Invoking methods of base class in derived class:

Class Fish{

private:

bool freshwater;

public:

Fish(bool fresh): freshwater(fresh){

}

void swim(){

cout<< "it’s a main fish methiod" << end;

}

}

Class Carp: public Fish{

public:

Carp(): Fish(true){

}

void swim(){

cout<< "it’s a basically tuna swim" << endl;

**Fish::swim()**

**}**

**}**

**Private Inheritance:**

**Private inheritance of a base class meaning that all public members and attributes of the base class are private to anyone with an instance of the Derived class**

**POLYMORPHISM:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Polymorphism allows objects of different types to be treated similarly. Polymorphism is implemented in c++ via the inheritance.

• Object of base class will behave like an actual derived class using the Virtual functions.

• Use of Virtual keyword means, compiler ensures that any overriding variant of the requested base class method is invoked.

• What happens when a function calls operator "delete" using a pointer of type Base\* that actually points to instance of the type \*Derived.(Need for Virtual destructors). //Destructor of the deriving class (that has been instantiated on the free storage using the new key word would not be deleted if the delete is called using a pointer of type Base). This results in resources not being released, memory leaks , and so on..

• Using Virtual destructors to ensure that Destructors in the derived classes are invoked when invoking a Pointer of Type Base

• Always declare base class destructor as Virutal. This ensures that one with a Pointer \*Base cannot invoke delete in a way that the destructor of the deriving classes is not invoked.

Note: In C++, virtual functions can be private and can be overridden by the derived class. For example, the following program compiles and runs fine.

#include<iostream>

using namespace std;

class Derived;

class Base {

private:

    virtual void fun() { cout << "Base Fun"; }

friend int main();

};

class Derived: public Base {

public:

    void fun() { cout << "Derived Fun"; }

};

int main()

{

   Base \*ptr = new Derived;

   ptr->fun();

   return 0;

}

There are few things to note in the above program.  
**1)** ptr is a pointer of Base type and points to a Derived class object. When ptr->fun() is called, fun() of Derived is executed.

**2)**int main() is a friend of Base.  If we remove this friendship, the program won’t compile (See this).  We get compiler error.

To implement virtual functions, C++ uses a special form of **late binding known as the virtual table or vTable**. The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner.

Every class that uses virtual functions (or is derived from a class that uses virtual functions) is given its own virtual table.

This table is simply a static array that the compiler creates at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class.

Each entry in this vTable is simply a Function Pointer that points to the most-derived function accessible by that class ie the most Base Class.

The compiler also adds a hidden pointer to the base class, which we will call \*\_\_vPtr.

\*\_\_vPtr is set (automatically) when a class instance is created so that it points to the virtual table for that class. \*\_\_vPtr is inherited by derived classes.

*class Base*

*{*

*public:*

*virtual void function1() {};*

*virtual void function2() {};*

*};*

*class D1: public Base*

*{*

*public:*

*virtual void function1() {};*

*};*

*class D2: public Base*

*{*

*public:*

*virtual void function2() {};*

*};*

*Because there are 3 classes here, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2.*

*The compiler also adds a hidden pointer to the most Base class that uses virtual functions.*

*class Base*

*{*

*public:*

*FunctionPointer \*\_\_vptr;*

*virtual void function1() {};*

*virtual void function2() {};*

*};*

*class D1: public Base*

*{*

*public:*

*virtual void function1() {};*

*};*

*class D2: public Base*

*{*

*public:*

*virtual void function2() {};*

*};*

**• vtable:** A table of function pointers, maintained per class.

**• vptr:**A pointer to vtable, maintained per object instance (see this for an example).

When a class object is created, \*\_\_vPtr is set to point to the virtual table for that class. For example, when a object of type Base is created, \*\_\_vPtr is set to point to the virtual table for Base. When objects of type D1 or D2 are constructed, \*\_\_vPtr is set to point to the virtual table for D1 or D2 respectively.

Because there are only two virtual functions here, each virtual table will have two entries (one for function1(), and one for function2()).

Base’s virtual table is simple. An object of type Base can only access the members of Base. Base has no access to D1 or D2 functions. Consequently, the entry for function1 points to Base::function1(), and the entry for function2 points to Base::function2().

D1’s virtual table is slightly more complex. An object of type D1 can access members of both D1 and Base. However, D1 has overridden function1(), making D1::function1() more derived than Base::function1(). Consequently, the entry for function1 points to D1::function1(). D1 hasn’t overridden function2(), so the entry for function2 will point to Base::function2().

D2’s virtual table is similar to D1, except the entry for function1 points to Base::function1(), and the entry for function2 points to D2::function2().

**Polymorphism Concepts:**

a) What is polymorphism.

b) Using base class pointers.

c) Static Vs Dynamic binding

d) Virtual functions.

e) Virtual Destructors.

f) Override and Final specifiers(c++11)

e) Using Base class referece

f) Pure Virtual functions and abstract classes

g) Abstract classes as interfaces.

a) Polymorphism is not default in C++, run-time polymorphism is achieved via.

i) Inheritance

ii) Base class pointers or base class references

iii) Virtual functions.

**Note:**

If a derived class writes its own method, then all functions of base class with same name become hidden, even if signatures of base class functions are different.

In the above question, when fun() is rewritten in Derived, it hides both fun() and fun(int) of base class.

This is a typical example of object slicing, when we assign an object of derived class to an object of base type, the derived class object is sliced off and all the data members inherited from base class are copied.

Object slicing should be avoided as there may be surprising results like above. As a side note, object slicing is not possible in Java. In Java, every non-primitive variable is actually a reference.

https://www.geeksforgeeks.org/object-slicing-in-c/

b) 2 Types of polymorphism

a) Compile time /static binding /early binding polymorphism: Achieved using the Function Overloading and Operator overloading.

b) Runtime/Dynamic Polymorphism: Function overriding(virtual) ,using the base class pointer.

In above case, even though we pass a derived class object , account withdraw is classed(note: display in account in not a Virtual function).

**Virtual Destrcutors:**

a) Problems can happen when we destroy polymorphic objects.

b) If a derived class is destroyed by deleted its storage via base class pointer and the class has non-virtual destructor. Then behavior is undefined in the C++ standard.

c) Derived Objects must be destroyed in the correct order starting at correct destructor.

Solution:

a) If a class has Virtual functions, Always provide a public virtual destructor

b) If base class destructor is virtual then all the derived class destructors are also virtual.

Ex:

Class Account {

Public:

virtual void withdraw(double amount);

virtual ~Account();

..

}

 For loop in shell:

for i in {50..50}; do for j in {0..3} ; do echo pg-$i-$j ;

//TCP Sockets understaning:

https://www.tldp.org/HOWTO/html\_single/TCP-Keepalive-HOWTO/

Establishing a Secure Connection Using SSL

**Secure Socket Layer** (SSL) technology is security that is implemented at the transport layer (see Transport-Layer Security, for more information about transport layer security). SSL allows web browsers and web servers to communicate over a secure connection. In this secure connection, the data that is being sent is encrypted before being sent and then is decrypted upon receipt and before processing. Both the browser and the server encrypt all traffic before sending any data. SSL addresses the following important security considerations.

**• Authentication**: During your initial attempt to communicate with a web server over a secure connection, that server will present your web browser with a set of credentials in the form of a server certificate. The purpose of the certificate is to verify that the site is who and what it claims to be. In some cases, the server may request a certificate that the client is who and what it claims to be (which is known as client authentication).

**• Confidentiality**: When data is being passed between the client and the server on a network, third parties can view and intercept this data. SSL responses are encrypted so that the data cannot be deciphered by the third party and the data remains confidential.

**• Integrity**: When data is being passed between the client and the server on a network, third parties can view and intercept this data. SSL helps guarantee that the data will not be modified in transit by that third party.

Installing and Configuring SSL Support

An SSL HTTPS connector is already enabled in the Application Server.

If you are using a different application server or web server, an SSL HTTPS connector might or might not be enabled. If you are using a server that needs its SSL connector to be configured, consult the documentation for that server.

As a general rule, to enable SSL for a server, you must address the following issues:

• There must be a Connector element for an SSL connector in the server deployment descriptor.

• There must be valid keystore and certificate files.

• The location of the keystore file and its password must be specified in the server deployment descriptor.

You can verify whether or not SSL is enabled by following the steps in Verifying SSL Support.

**Inline Functions:**  A regular function call is translated into a CALL instruction, which results in the stack operations and microprocessor execution shift to the functions and so on. This is a quite long process.

For a simple function like below, the overhead of performing an actual function call on this might be a quite high for the amount of time spent actually executing the function. This is why C++ compilers enables the programmers to declare such functions as inline.

Double GetPi(){

return 3.14;

};

After Inline:

Inline double GetPi(){

return 3.14;

};

Note: When optimizing for size, compilers might often reject many inline requests as that might bloat code.

When optimizing for the speed, compilers typically sees and utilizes the opportunities to inline code where it would make sense and does it for you.

**Default Arguments in C++:**

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

*#include<iostream>*

*using namespace std;*

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

*// 2 arguments or 3 arguments or 4 arguments.*

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

*{*

*return (x + y + z + w);*

*}*

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

*int main()*

*{*

*cout << sum(10, 15) << endl;*

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

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

*return 0;*

*}*

**Key Points:**

• Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.

• Default arguments are overwritten when calling function provides values for them. For example, calling of function sum(10, 15, 25, 30) overwrites the value of z and w to 25 and 30 respectively.

• During calling of function, arguments from calling function to called function are copied from left to right. Therefore, sum(10, 15, 25) will assign 10, 15 and 25 to x, y, and z. Therefore, the default value is used for w only.

• Once default value is used for an argument in function definition, all subsequent arguments to it must have default value. It can also be stated as default arguments are assigned from right to left. For example, the following function definition is invalid as subsequent argument of default variable z is not default.

// Invalid because z has default value, but w after it

// doesn't have default value

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

**Advanced Pointer Notation:**

nums[i][j] is equivalent to \*(\*(nums+i)+j);

int nums[2][3] = { { 16, 18, 20 },

{ 25, 26, 27 } };

**//String Literals:**

**const char \*name = “Shankar”;**

**Void Pointers:**

This is a special type of pointer available in C++ which represents absence of type. void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).  
This means that void pointers have great flexibility as it can point to any data type. There is payoff for this flexibility. These pointers cannot be directly dereferenced. They have to be first transformed into some other pointer type that points to a concrete data type before being dereferenced.

**SemaPhores in c++:** Semaphores basically a signaling mechanism, (whereas mutex is a locking mechanism).

C++ standard does not define a semaphore type. You can write your own with an atomic counter, a mutex and condition variable if you need.

but most uses of semaphores are better replaced with mutexes and/or condition variables anyway.

Mutex: is a mutual exclusion object that sychronizes access to a resource. The mutex is a locking mechanism that ensures that only one thread can acquire the mutex at a time, and entire critical section. This thread only releses the mutex when it exits the critical section.

Semaphore: is a signaling mechanism, a thread is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only the thread that called the wait section.

**Algorithm Techniques:**

**BackTracking:** it’s a algorithm technique which is used to resolve the problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

For example, consider the SudoKo solving Problem, we try filling digits one by one. Whenever we find that current digit cannot lead to a solution, we remove it (backtrack) and try next digit. This is better than naive approach (generating all possible combinations of digits and then trying every combination one by one) as it drops a set of permutations whenever it backtracks.

**Dynamic Programming:**

Some famous Dynamic Programming algorithms are:

• Unix diff for comparing two files

• Bellman-Ford for shortest path routing in networks

• TeX the ancestor of LaTeX

• WASP - Winning and Score Predictor

• The core idea of Dynamic Programming is to avoid repeated work by remembering partial results and this concept finds it application in a lot of real life situations.

• In programming, Dynamic Programming is a powerful technique that allows one to solve different types of problems in time O(n2) or O(n3) for which a naive approach would take exponential time.

**Forward Declarations:**

https://www.geeksforgeeks.org/what-are-forward-declarations-in-c/

• Sleep(), usleep()(micro seconds sleep), are functions defined the header <uinstd.h>, and they block the execution of the thread for the amount of sleep time.

• Std::this\_thread::sleep\_for() is defined for the thread class in c++ (header <thread>).

•

//Extracting the Pod Information and export them as the environment variables:

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: dapi-envars-fieldref

**spec**:

**containers**:

- **name**: test-container

**image**: k8s.gcr.io/busybox

**command**: [ "sh", "-c"]

**args**:

- while **true**; do

echo -en '\n';

printenv MY\_NODE\_NAME MY\_POD\_NAME MY\_POD\_NAMESPACE;

printenv MY\_POD\_IP MY\_POD\_SERVICE\_ACCOUNT;

sleep 10;

done;

**env**:

- **name**: MY\_NODE\_NAME

**valueFrom**:

**fieldRef**:

**fieldPath**: spec.nodeName

- **name**: MY\_POD\_NAME

**valueFrom**:

**fieldRef**:

**fieldPath**: metadata.name

- **name**: MY\_POD\_NAMESPACE

**valueFrom**:

**fieldRef**:

**fieldPath**: metadata.namespace

**- name: MY\_POD\_IP**

**valueFrom:**

**fieldRef:**

**fieldPath: status.podIP**

- **name**: MY\_POD\_SERVICE\_ACCOUNT

**valueFrom**:

**fieldRef**:

**fieldPath**: spec.serviceAccountName

**restartPolicy**: Never

**Sizeof():** sizeof a empty class is 1 byte. t is nonzero to ensure that the two different objects will have different addresses.

The instances of the below class has the size of 4 bytes(for 32 bit compier) and 8 bytes (for 64 bit compiler), its because it has the char\* pointer whose size is either 4 bytes or 8 bytes.

MyStrSize m1{“shank”};

MyStrSize m2{“Shankar”}; //here both instances has the same size

/\*

**class** MyStrSize{

**private**:

**char** \*buffer;

**public**:

MyStrSize(**char**\* );

MyStrSize(**const** MyStrSize& m);

~MyStrSize();

**int** GetLength();

**const** **char**\* GetString();

}; \*/

**Static in the C++:**

**Note:** C++ does not allow in class initialization non-constant static data members.

Static Methods of the class only has access to the static variables of the class.

Static methods can only access static members (data and methods)

A static variable inside a class should be initialized explicitly by the user using the class name and scope resolution operator outside the class

Class player{

Private:

Static int numplayers {0}; //This is not allowed.

Std::string name;

Int health;

Int xp;

Public:

Std::string getname(){

Return name;

}

Int getxp(){

Return xp;

}

Player(std::string s, int h, int x):name(s),health(h),xp(x){

}

Player(const Player& p){

name(p.s),health(p.h),xp(p.x){

}

Static int getplayers(){

return numplayers;

}

}

**Friend Key Word:**

A class does not permit external access to its data members and methods which are declared as private. This rule is waived for those classes(A friend class inside a class) and functions (inside another class ) that are disclosed as friend classes or friend functions with the keyword “Friend”.

*#include <iostream>*

*#include <string>*

***using******namespace*** *std;*

***class*** *Human{*

***private****:*

***int*** *age;*

*string name;*

***friend******void*** *displayDetails(****const*** *Human& h); // this indicates to the compiler that this function in global scope be permitted special access to the private members of class Human*

***public****:*

*Human(****int*** *a, string n):age(a),name(n){};*

*};*

***void*** *displayDetails(****const*** *Human& h){*

*cout << "Age is " << h.age << endl;*

*cout << "name is " << h.name << endl;*

*}*

***int*** *main(){*

*Human h1 {15, "baby"};*

*displayDetails(h1);*

***return*** *0;*

*}*

Like functions, external classes can also be designed as a trusted friend:

***using******namespace*** *std;*

***class*** *Human{*

***private****:*

***int*** *age;*

*string name;*

***friend******class Utility;***

***public****:*

*Human(****int*** *a, string n):age(a),name(n){};*

*};*

Class Utility{

Static void DisplayDetails(const Human& h){

*cout << "Age is " << h.age << endl;*

*cout << "name is " << h.name << endl;*

};

There are a great number of online platforms that can help you get started with programming. YouTube is one of the independent and free sources of learning. There are a great of number creators on YouTube posting programming and technical tutorials. Here are the top-rated coding and programming channels on YouTube.  
  
1. freeCodeCamp.org  
The non-profit community on YoTube offers coding and tech learning. The channel provides free videos, articles, coding interview questions, and interacting learning lessons. You can get certification after completion of coding challenges. The Python course for beginners, SQL tutorial, and JavaScript full course are the most popular courses on this channel. freeCodeCamp.org has 1.92 million subscribers.  
  
2. Edureka  
The e-learning platform offers instructor-led courses, webinars, and lectures. It also has a Youtube Channel that hosts most of these courses. The most popular technologies covered by the channel include big data, DevOps, Data Science, Hadoop, Apache Spark, Python, Selenium, Blockchain, Tableau, Artificial Intelligence (AI), AWS, and digital marketing. The YouTube channel has 1.68 million subscribers.  
3. ProgrammingKnowledge  
This channel offers fundamental knowledge one arias programming topics. It talks about online programming tutorials, coding strategies, installation of open-source software etc. The topics covered in the vides include Elastic Stack, Python, Android, Flutter, Socket Programming, MongoDB etc.  
  
4. Telusco  
This channel offers free tutorials from beginner to advanced level. The technical topics covered in their videos include Python, Blockchain, Android, JavaScript, Rest API, Kotlin, Scala, Spring Framework, Networking etc. Telusco also offers motivation videos and online sessions with industry experts. The channel has 993,000 subscribers.  
  
5. Intellipaat  
This YouTube channel offers free courses in big data, data science, and artificial intelligence. The video content also helps professionals in making a career decision. Some videos talk about how to assist corporate clients in upskilling their workforce. The channel has 496,000 subscribers.

**Condition Variables:** condition variable class is a synchronization primitive that can be used to block a thread or multiple threads at the same time, until another thread both modifies a shared variable(the condition) and notifies the condition variable.

• The thread that intends to modify the variable has to:

• Aquire a std::mutex (typically via std::lock\_guard)

• Perform modification while the lock is held

• Execute notify\_one and notify\_all on the std::condition variable.

• Even if the shared variable is atomic, it must be modified under the mutex in order to correctly publish the modification to the waiting thread.

• Any thread that intends to wait on std::condition\_variable has to

Methods:

Wait: wait causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied.

• 1) Atomically unlocks lock, blocks the current executing thread, and adds it to the list of threads waiting on \*this. The thread will be unblocked when notify\_all() or notify\_one() is executed. It may also be unblocked spuriously. When unblocked, regardless of the reason, lock is reacquired and wait exits. If this function exits via exception, lock is also reacquired. (until C++14)

• 2) Equivalent to

• while (!pred()) {

• wait(lock);

• }

• This overload may be used to ignore spurious awakenings while waiting for a specific condition to become true.

• Note that lock must be acquired before entering this method, and it is reacquired after wait(lock) exits, which means that lock can be used to guard access to pred().

•

Parameters

**lock**

-

an object of type std::unique\_lock<std::mutex>, which must be locked by the current thread

**pred**

-

predicate which returns ​false if the waiting should be continued.

The signature of the predicate function should be equivalent to the following:

Bool::pred

The class unique\_lock is a general-purpose mutex ownership wrapper allowing deferred locking, time-constrained attempts at locking, recursive locking, transfer of lock ownership, and use with condition variables.

The class unique\_lock is movable, but not copyable -- it meets the requirements of *MoveConstructible* and *MoveAssignable* but not of *CopyConstructible* or *CopyAssignable*.

The class unique\_lock meets the *BasicLockable* requirements. If **Mutex** meets the *Lockable* requirements, unique\_lock also meets the *Lockable* requirements (ex.: can be used in std::lock); if **Mutex** meets the *TimedLockable* requirements, unique\_lock also meets the *TimedLockable* requirements.

**Note:** Here inside the wait, if the predicate condition met ,(or it returns true), then only lock will happen and steps(after the wait) will start to execute.

b) **Wait\_for:**  Blocks the current thread until either notify\_one or notify\_all is executed or relative timeout(rel\_timeout) is executed.  It may also be unblocked spuriously. When unblocked, regardless of the reason, lock is reacquired and wait\_for() exits. If this function exits via exception, lock is also reacquired.

/\*\*\*

*template< class Rep, class Period >*

*std::cv\_status wait\_for( std::unique\_lock<std::mutex>& lock,*

*const std::chrono::duration<Rep, Period>& rel\_time);*

*\*\*\*\*\*/*

**std::ref, std::cref:**  Definition is in the header <functional>: Function templates ref and cref are helper functions that generate an object of type std::reference\_wrapper, using template argument deduction to determine the template argument of the result.

template< class T >  
std::reference\_wrapper<T> ref(T& t) noexcept;

**Parameters**

**t**

-

lvalue reference to object that needs to be wrapped or an instance of std::reference\_wrapper

**Return value**

1) std::reference\_wrapper<T>(t)

2) std::ref(t.get())

4) std::reference\_wrapper<const T>(t)

5) std::cref(t.get())

**std::function:**  header is <functional> and Class template std::function is a general-purpose polymorphic function wrapper. Instances of std::function can store, copy, and invoke any *Callable* *target* -- functions, lambda expressions, bind expressions, or other function objects, as well as pointers to member functions and pointers to data members.

**Std::StringStream:**  It’s a Stream class to operate on Strings.

Objects this class use a string buffer that contains a sequence of characters. The sequence of characters can be accessed directly as a String object or using a member str.

Characters can be inserted and/or extracted from stream using any operation allowed on the both input and output streams.

Apart from the internal *string buffer*, objects of these classes keep a set of internal fields inherited from ios\_base, iosand istream:

**Input and Output Operations in C++:**

Base Derived

**Streams: (Files, Streams, I/O)**

• Stream is a sequence of bytes. I/p steam provides data to the stream, O/P stream receives the dat from the stream.

• C++ uses streams as an interface b/w the program and input and output devices.

3) Independent of actual device.

Common header files:

• iostream -->provides definitions for formatted input and output from i/o streams.

• fstream --> provides definitions for formatted inptu and output from the file i/o steams.

• Iomanip --> provides definitions for the manipulators used to format stream i/o.

After including the above headers we have access to many c++ stream classes :

Class details

Ios : provides support for both formatted and unformatted i/o operations.

Base class for most of the other classes

Ifstream: provides for highlevel input operations on the file based streams.

Ofstream: provides for highlevel output operations on the file based streams.

Fstream: provides for high-level i/o operations on the file based streams. Derived from the ifstream and ofstream

Stringstream: provides for high-level i/o operations on memory based strings . Derived from istringstream and ostringstream.

Cin -- standard Input stream, by default its connected to the standard input device(keyboard). Instance of istream class.

Cout -- standard output stream, by default its connected to the standard output device(console). Instance of ostream class.

Both cin and cout are buffered steams . i.e unless user presses enter , i/p wont be considered, for ouput ,user either has to provide newline or flushes the output.

Clog and cerr: Both are connected to the default standard error/log device (console). Cerr is instance of ostream(unbuffered)

Clog is also instance of ostream class(unbuffered).

Explore: Redirecting i/o.

**Stream Manipulators( iomanip):**

• Streams have many useful member functions to control formatting.

• Common stream manipulators:

a) Boolean : boolalpha, noboolalpha.

b) Integer: dec,hex,oct, showbase, noshowbase, showpos, noshowpos, uppercase, nouppercase.

c) Floating point: fixed, scientific, setprecision, showpoint, noshowpoint, showpos, noshowpos.

d) Field Width, justification and fill: setw, left, right, internal, setfill.

e) Others: endl, flush, skipws, noskipws, ws(whitespace).

Formatted integer types:

Default:

a) decimal (base 10)

b) noshowbase: prefix used to show hexadecimal or octal.

c) nouppercase: when displaying a prefix and hex values it will be lower case.

d) noshowpos: no "+" is displayed for +ve integers.

Floating point formatters:

Default:

a) setprecision: no. of digits displayed are controlled to 6.

b) fixed: not fixed to a specific no. of digits after the decimal.

c) noshowpoint: -- trailing zeros are not displayed.

• Nouppercase: when displaying in a scientific notation.

• Noshowpos: no "+" is displayed for +ve integers.

S**tream Manipulators -- align and fill:**

Default:

setw -- not set by default.

left -- when no field width, right -- when using field width

fill -- not set by default -- blank space is used.

Some of these manipulators affect only the next data element put on the stream.

Setw, std::right(default is right justified), std::left

-> if setw is done without any justificatoin, its right justified.

--> for left justification: , do like this setw(10) << std::left << "printnumber" << endl

--> setfill('\*');

**Note:** if we use setfill first and then use the setw, then default alignment goes to the left

**Reading data from a text file:**

Input files: fstream or ifstream classes are used for input files:

Header , #include <fstream>

• Define the steam object.

• Connect/open to the file on your file system(opens for reading)

• Read data from the file via the stream.

• Close the stream

Modes of opening file:

Std::ios::in (input mode) -- cannot write here

Std::ios::binary (binary mode) -- for non text files

std::ios::trun(truncate mode), -- clearing the files

Ex: fstream fi {"../shankar.txt", std::ios::in};

fstream fi {"../shankar.txt", std::ios::in | std::ios::binary};

Opening file in reading mode:

Std::ifstream in\_file;

Std::string filename;

Cin >> filename;

In\_file.open(filename) or in\_file.open(filename, std::ios::binary);

To check whether file opened successfully or not:

If(In\_file.is\_open()){ Or We can check like if(in\_file)

//read from the file

}else{

//error in reading the file

}

Reading lines from the file

String line;

Infile >> line;

//reading 2lines at at time

(while(Infile >> line1 >> line2))

Each line at a time

While(getline(infile,line))

In\_file.close();

//Closing the file stream is very important especially for the files which are opened for writing to flush out unwritten data

**Writing the data to files:**

-> fstream and ofstream are commonly used for output files.

Steps

• #include <fstream>

• Declare a fstream or an ofstream object.

• Connect to the file on your file system.

• Write data to the file via stream.

• Close the stream.

• By default C++ creates a file, if it doesnot exist.

• Write using a binary or text mode

• Open the file in below ways:

• Std::fstream outst {"output.txt" , std::ios::out} Or { "output.txt} //here ios::out means opening the file in the output mode.

//here both declarations are same.

• Std::ofstream outst {"output.txt" , std::ios::out | ios::binary};

std::ofstream outst {"output.txt", std::ios::trunc} //Truncate or(discard contents) when opening the file.

std::ofstream outst {"output.txt", std::ios::app} // append on each write

std::ofstream outst {"output.txt", std::ios::ate} // seek to end of stream when opening

\*\*\*\*\*Writing to the files\*\*\*\*\*\* Use operator "<<"

**StringStream:**

Using streams, we can read and write to the streams.

There are 3 classes: stringstream(read & write from stringstreams)

Istringstream(read from stringstream)

Ostringstream(write to the stringstream)

• Header is, #include <sstream>

• Decleare an stringstream, istringstream, ostringstream object.

• Connect it to a std::string

• Read/Write data from/to the string stream using the formatted I/O.

Reading from a stringstream

/\*\*\*\*\*\*\*\*\*

#include <sstream>

Int num {};

Double total {};

Std::string name {};

Std::string info {"Moe 100 1234.5"};

Std::istringstream iss{info};

Iss >> name >> num >> total;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <sstream>

Int num {100};

Std::string name {"shankar"};

Double d {1234.5};

Std::ostringstream oss {};

Oss << name << " " << num << " " << total;

Std::cout << oss.str() << std::endl;//converting to a string

**Example 2: function chaining calls using this pointer**

Another example of using this pointer is to return the reference of current object so that you can chain function calls, this way you can call all the functions for the current object in one go. Another important point to note in this program is that I have incremented the value of object’s num in the second function and you can see in the output that it actually incremented the value that we have set in the first function call. This shows that the chaining is sequential and the changes made to the object’s data members retains for further chaining calls.

#include <iostream>  
using namespace std;  
class Demo {  
private:  
 int num;  
 char ch;  
public:  
 Demo &setNum(int num){  
 this->num =num;  
 return \*this;  
 }  
 Demo &setCh(char ch){  
 this->num++;  
 this->ch =ch;  
 return \*this;  
 }  
 void displayMyValues(){  
 cout<<num<<endl;  
 cout<<ch;  
 }  
};  
int main(){  
 Demo obj;  
 //Chaining calls  
 obj.setNum(100).setCh('A');  
 obj.displayMyValues();  
 return 0;  
}

**Unary Operators as Global Functions :**

Returntype operatorOp(Type &obj)

Number operator-(const Number &n);

Number operator++(Number &obj) // pre-increment

Number operator++(Number &obj, int) // Post-increment

Bool operator!(const Number& obj);

Number n1{100};

Number n2 = -n1;

N2 = ++n1;

N2 = n1++;

These methods will be accessed using the friend functions.

//In Case of Binary operator overloading, there will be 2 args passed to the function

(binary Operators as global functions)

Number operator+(const &Number lhs, const &Number rhs);

Number operator-(const &Number lhs, const &Number rhs);

Number operator==(const &Number lhs, const &Number rhs);

Number operator<(const &Number lhs, const &Number rhs);

Number n1 {100}, n2 {200};

Number n3 = n1+n2;

N3 = n1 -n2;

If( n1 == n2) …

**Programming Conversion Operators:**

Suppose if we have a class date as below, if we do below:

Date holiday(5,4,2016);

cout << holiday; //Compiler throws an error: binary << : no operator found which takes a righ hand operarand of type ‘Date’. This error essentially indicates that cout does not know how to interpret an instance of Date class, Date does not support the relevant operator.

*class Date{*

*private:*

*int dd;*

*int mm;*

*int year;*

*public:*

*Date(int d, int m, int y):dd(d),mm(m),yy(y){*

*}*

*};*

Conversion function is declared like a non-static member function or member function template with no parameters, no explicit return type, and with the name of the form:

**operator** *conversion-type-id*

(1)

**explicit** **operator** *conversion-type-id*

(2)

(since C++11)

**explicit (** *expression* **)** **operator** *conversion-type-id*

(3)

(since C++20)

1) Declares a user-defined conversion function that participates in all implicit and explicit conversions.

2) Declares a user-defined conversion function that participates in direct-initialization and explicit conversions only.

3) Declares a user-defined conversion function that is conditionally explicit.

**How Ever Cout work well with a const char\***

***operator const char\*(){***

***//Implementation***

***}***

**Pointers:**

• Pointer is a variable whose value is an address. That address can be address of a variable or function.

• Inside functions , pointers can be used to access data that are defined outside the function, those variables may not be in the scope , you cannot access them by their name.

• Pointers can be used to operate on arrays very efficiently.

• We can allocate memory dynamically on a heap or free store, that memory does not even have a variable name. The only way to get them is via pointer.

• With OO. Pointers are how polymorphism works.

• Can access specific addresses in memory, useful in embedded and system applications.

**Pointer Declaration and Initialization:**

• Int \*data; (integer pointer)

• Double\* d; (double pointer)

• String \*str1 (pointer to a string object)

• Char \*char (character pointer)

/

An uninitialized pointer may point to any where(or any address). Initializing pointer:

• Int \*int\_ptr {}; //It will not point to anywhere

• double \*d {nullptr};

• char \*ch {nullptr};

Best Practices:

• Always initialize pointers.

• Uninitialized pointers contain garbage data and can point to anywhere.

• Initializing to zero or nullptr (c++11) represents address zero.

Ex:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int \*p;

cout << "the value of p " << p << endl; // 06xxxxfyz (garbage) as it’s a uninitialized pointers.

cout << "the address of p" << &p << endl // 0x6abcde

cout << "the size of p is" << sizeof p << endl; 4

p = nullptr

cout << "the value of p is: " << p << endl; // 0

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Size of a pointer varialbe or sizeof of what it pointers are both are different. So all addresses have the same size.

cout << "the size of the int pointer " << sizeof p << endl; //8

cout << "the size of the double pointer " << sizeof d << endl;

cout << "the size of the char pointer " << sizeof ch << endl;

cout << "the size of the addr int pointer " << sizeof &p << endl; //8

cout << "the size of the addr double pointer " << sizeof &d << endl;

cout << "the size of the addr char pointer " << sizeof &ch << endl;

cout << "the size of the int pointer " << sizeof \*p << endl; //4

cout << "the size of the double pointer " << sizeof \*d << endl; //1

cout << "the size of the char pointer " << sizeof \*ch << endl; //1

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int ab {10};

double d {20.0};

int cd {100};

int \*ptr;

ptr = &ab; //compiler works.

ptr = &d; // Compilation errors, you are trying to point to address which is of double.

ptr = &cd //It works.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Dereferencing the pointer: (\*)

int ab {10};

int \*ab\_ptr {&ab};

cout << ab << endl; //10

cout << \*ab\_ptr << endl; //10

\*ab\_ptr = 20;

cout << ab << endl; //20

cout << \*ab\_ptr << endl; //20

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

string str1 = "shankar";

string \*str\_ptr {&str1};

cout << \*str\_ptr << endl ; //shankar

str1 = "madhavi";

cout << \*str\_ptr << endl ; //madhavi

**Dynamic Memory Allocation:**

*int \*int\_ptr {nullptr};*

*int\_ptr = new int; //allocate integer on the heap.*

*cout << int\_ptr << endl;   //ox2747f2xxx*

*cout << \*int\_ptr << endl; //some garbage value*

*\*int\_ptr = 100;*

*Cout << \*int\_ptr << endl; //100*

delete int\_ptr; //Frees allocated storage.

//Array of Pointers

int \*array\_ptr {nullptr};

int size {};

cout << “How big do you want the array size?” << endl;

cin >> size;

array\_ptr = new int[size];

…

delete[] array\_ptr; //Free allocated storage.

**Pointer Arithmetic:**

Pointers can be used in:

a) Assignment expressions

b) Arithmetic expressions

c) Comparison expressions

C++ allows pointer arithmetic

Note: Pointer arithmetic only makes sense with raw arrays.

a) (++) increment a pointer to point to the next array element.ex: int\_ptr++.

b) (--) decrement a pointer to point to the previous array element: ex: int\_ptr==

c) + (addition), increment pointer by n\*sizeof(type) ex: int\_ptr += n; or int\_ptr = int\_ptr+n;

d) – (substraction), decrement a ponter by n\*sizeof(type)

int\_ptr -= n; or int\_ptr = int\_ptr-n;

**Subtracting two pointers:**

**Note: Determine no. of elements b/w the pointers**

**Both pointers must point to the same data type**

int n = int\_ptr2 – int\_ptr1;

**Comparing 2 pointers: ( ==) and (!=):**

Determine if 2 pointers point to the same location. (does not compare the data where they point).

string s1 {“shankar”};

string s2 {“shankar”};

string \*p1 {&s1};

string \*p2 {&s2};

string \*p3 {&s1};

cout << (p1 == p2) << endl; //false

cout << (p1 == p3) << endl; //true;

cout << (\*p1 == \*p2) << endl; //true

cout << (\*p1 == \*p3) << endl; //true

**Constant Pointers:**

There are several ways to qualify pointers using **const.**

a) Pointers to constants

b) Constant Pointers

c) Constant Pointers to constants.

**Pointers to Constants:**

The data pointed to by the pointers is constant and cannot be changed.

The pointer itself can change and point somewhere else.

int high\_score {100};

int low\_score {65};

const int \*score\_ptr {&high\_score};

\*score\_ptr = 86;//Error

score\_ptr = &low\_score; //Ok

**Constant Pointers:**

a) The data pointed to by the pointers can be changed.

b) The pointer itself cannot change and point somewhere else.

int high\_score {100};

int low\_score {65};

int \*const score\_ptr {&high\_score};

\*score\_ptr = 86;//Ok

score\_ptr = &low\_score; //Error

**Constant Pointers to Constants:**

a) The data pointed to by the pointer is constant and cannot be changed.

b) The Pointer itself cannot change and point somewhere else.

int high\_score {100};

int low\_score {65};

const int \*const score\_ptr {&high\_score};

\*score\_ptr = 86;//Error

score\_ptr = &low\_score; //Error

**Passing pointers to a function:**

a) Pass by reference with pointer parameters

b) We can use pointers and dereference operator to achieve pass-by-reference.

c) the function parameter is pointer

d) The actual parameter can be a pointer or address of a another variable.

ex:

void double\_data(int \*int\_ptr);

void double\_data(int \*int\_ptr){

\*int\_ptr \*=2; // Or \*int\_ptr = \*int\_ptr \*2;

}

**Pass by reference ex:**

int main(){

int value {10};

cout << value << endl;

double\_data(&value);

cout <<value << endl;

}

**Returning pointer from a function:**

syntax : typename \*createarray(){

return typename;

}

//Error Code:

void display( int\* const a,int n){

for(int i=0;i<n;i++){

cout << "array values: " << \*(a++) << endl; // Cannot assign to variable 'a' with const-qualified type 'int \*const'

//cout << "array values: " << \*(a+i) << endl;

//or cout << "array values: " << a[i] << endl;

}

}

//which will work:

void display( int\* a,int n){

for(int i=0;i<n;i++){

cout << "array values: " << \*(a++) << endl; //

//cout << "array values: " << \*(a+i) << endl;

//or cout << "array values: " << a[i] << endl;

}

}

**Don’t do these at any time: (As we returning the address of a stack variable, it will cause issues, the scope variable will not exist after the control comes out the scope.**

*int\* value(){*

*int a;*

*return &a;*

*}*

*or*

*int\* value(){*

*int \*a;*

*int size=10;*

*a = &size;*

*return a;*

*}*

**Pointer Pitfalls:**

a) Uninitialized pointers. (int \*intrptr; //pointing to anywhere)

\*int\_ptr = 100; //Hopefully a crash.

b) Dangling pointers.

c) Not checking if “new” failed to allocate memory.

c) Memory Leaks or Leaking Memory.

**Dangling Pointers:**

a) Pointer that is pointing to released memory.

ex: 2 pointers point to same data( create a pointer using a new, assign to a 2nd pointer of same type,1 pointer release data using the delete, 2nd will be pointing to invalid location or data, results are unpredictable).

b) Pointer that points to a memory that is invalid.

a) this will happen when we return a pointer to a function local variable.

**Not checking if new failed:**

a) If new fails an exception is thrown.

b) We can use exception handling to catch exceptions.

c) Dereferencing a null pointer will cause your program to crash.

**Memory leak:**

a) Forgetting to release allocated memory with delete.

b) If you lose your pointer to the storage allocated on the heap you don’t have a way to get that storage again.

c) The memory is orphaned or leaked.

d) One of the most common pointer problems.

Bubble Sort ( In each swap the biggest element will be pushed to the end) (It moves to the highest element to the right most corner on each pass).

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

**Example:**

**First Pass:**

( **5** **1** 4 2 8 ) –> ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.

( 1 **5** **4** 2 8 ) –>  ( 1 **4** **5** 2 8 ), Swap since 5 > 4

( 1 4 **5** **2** 8 ) –>  ( 1 4 **2** **5** 8 ), Swap since 5 > 2

( 1 4 2 **5** **8** ) –> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**

( **1** **4** 2 5 8 ) –> ( **1** **4** 2 5 8 )

( 1 **4** **2** 5 8 ) –> ( 1 **2** **4** 5 8 ), Swap since 4 > 2

( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )

( 1 2 4 **5** **8** ) –>  ( 1 2 4 **5** **8** )

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass:**

( **1** **2** 4 5 8 ) –> ( **1** **2** 4 5 8 )

( 1 **2** **4** 5 8 ) –> ( 1 **2** **4** 5 8 )

( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )

( 1 2 4 **5** **8** ) –> ( 1 2 4 **5** **8** )

*From <https://www.geeksforgeeks.org/bubble-sort/>*

**Good Links for tutorial:**

https://thispointer.com/c11-move-contsructor-rvalue-references/

https://stackoverflow.com/questions/36568377/stdvector-memory-handling?noredirect=1&lq=1

https://slideplayer.com/slide/4271342/

**Install Make in the Ubuntu OS:**

Run the command:

**sudo apt-get install build-essential**

Chances are you will need things like gcc to actually do the building so you might as well install those as well. The build-essential package will install other tools used along with make.

**Sorting Techniques:**

**• Selection Sort**: The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array

• A Sorted subarray

• Remaining array which is unsorted.

• In each iteration the minimum element is moved to the left or beginning of the array.

**Stable Selection Sort**: A sorting algorithm is said to be stable if two objects with equal or same keys appear in the same order in sorted output as they appear in the input array to be sorted.

Selection sort works by finding the minimum element and then inserting it in its correct position by swapping with the element which is in the position of this minimum element. This is what makes it unstable.

Swapping might impact in pushing a key(let’s say A) to a position greater than the key(let’s say B) which are equal keys. which makes them out of desired order.

To make selection sort as stable:

Example: **4A 5 3 2 4B 1**

First minimum element is 1, now instead

of swapping. Insert 1 in its correct place

and pushing every element one step forward

i.e forward pushing.

**1 4A 5 3 2 4B**

Next minimum is 2 :

**1 2 4A 5 3 4B**

Next minimum is 3 :

**1 2 3 4A 5 4B**

Repeat the steps until array is sorted.

**1 2 3 4A 4B 5**

/\*\*\*\*\*\*

*for(int i= 0; i<n-1 ; i++){*

*int min = i;*

*for(int j= i+1;j<n;j++){*

*if(a[min] > a[j]){*

*min = j;*

*}*

*}*

*int key = a[min];*

*while (min > i)*

*{*

*a[min] = a[min - 1];*

*min--;*

*}*

*a[i] = key; }. \*\*\*\*\*\*/*

**Time Complexity:** O(n2) as there are two nested loops.

**Auxiliary Space:** O(1)  
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

**Exercise :**  
Sort an array of strings using Selection Sort

Stability : The default implementation is not stable. However it can be made stable. Please see stable selection sort for details.

**Bubble Sort**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

**Rule is**: In each iteration, the highest value is pushed to the end of the array:

Example:  
**First Pass:**  
( **5** **1** 4 2 8 ) –> ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) –>  ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) –>  ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) –> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**  
( **1** **4** 2 5 8 ) –> ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) –> ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –>  ( 1 2 4 **5** **8** )  
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass:**  
( **1** **2** 4 5 8 ) –> ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) –> ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –> ( 1 2 4 **5** **8** )

*for(int i= 0;i<n-1;i++){*

*for(int j=0;j<n-i-1;j++){*

*if(a[j] > a[j+1])*

*interchange(a[j], a[j+1]);*

*}*

*cout << "After each pass: " << i << endl;*

*/\* for (int i= 0;i<n;i++){*

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

*} \*/*

*}*

**Worst and Average Case Time Complexity:**O(n\*n). Worst case occurs when array is reverse sorted.

**Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.

**Auxiliary Space:** O(1)

**Boundary Cases:** Bubble sort takes minimum time (Order of n) when elements are already sorted.

**Sorting In Place:**Yes

**Stable:** Yes

Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm.  
In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n). For example, it is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to x axis) and with incrementing y their order changes (two elements are swapped) only at intersections of two lines (Source: Wikipedia)

**Insertion Sort:**

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Here are also in every swap , we will ensure that min key value is pushed to the left/beginning, and importantly remaining keys will be shifted.

• STL:

• A library of powerful, reusable, adaptable, generic classes and functions.

• Implemented using C++ templates.

• Implemented common data structures and algorithms.

• Huge class library.

• STL has 3 main Elements

**• Containers** (Collection of objects or primitive types) like array, vector, stack, queue, set, map ,deque etc.

**• Algorithms** (Functions for processing sequence of elements from containers) like find, max, count, accumulate, sort etc.., It has 60 algorithms.

**• Iterators** (Generate sequence of element from containers) like, forward, reverese, by value, by reference, constant etc.

It has other elements also such as Functors and allocators also. (Need to explore them)

• Types of Containers:

a) Sequence Containers: Array, list, deque, vector, forward\_list. (Maintains the order of insertion elements).

b) Associate Containers: set, multi\_set, map, multi map (Inserts elements in a predefined order, or no order at all)

c) Container Adapters: Stack, Queue, priority queue. (Here iterators will not work).

• Iterators:

a) Input Iterators: from the container to the program.

b) Output Iterators: from the program to the container

c) Forward Iterators:

d) Bidirectional Iterators.

e) Random Access Iterators -- directly access a Container item.

5) Algorithms:

a) About 60 algorithms in the STL.

b) Non-Modifying

c) Modifying

• Generic Programming with Macros

• Generic Programming

• Macros

• Function Templates

• Class Templates

7) **Macros: (# define)**

a) C++ Preprocessor directives

b) No Type information

c) Simple Substitution

ex : #define MAX\_SIZE 100

#define PI 3.1456

Suppose we need to determine maximum of 2 ints, 2 longs, 2 chars

int max(int a, int b){

return (a>b) ? a: b;

double max(double a, double b){

return (a>b) ? a: b;

char max(char a, char b){

return (a>b) ? a: b;

Instead of above, we can write a generic Macro with arguments instead:

#define MAX(a,b) ((a>b) ? a: b)

Std:cout << MAX(1,2) << end;

Std::cout << MAX(2.0, 4.0) << endl;

Std::cout << MAX(z, a) << endl;

What is C++ Templates:

• Blue Print.

• Function and Class Templates

• Allow Plugging-in any data type

• Compiler generates appropriate function/class from the blueprint.

• Generic Programming/ meta programming

**• Note**: Compiler creates a new instance of a template function for every data type. So compiler creates two functions in the above example, one for int and other for double. Every instance has its own copy of static variable. The int instance of function is called twice, so count is incremented for the second call.

*#include <iostream>*

*using namespace std;*

*template <typename T>*

*void fun(const T&x)*

*{*

*static int count = 0;*

*cout << "x = " << x << " count = " << count << endl;*

*++count;*

*return;*

*}*

*int main()*

*{*

*fun<int> (1);*

*cout << endl;*

*fun<int>(1);*

*cout << endl;*

*fun<double>(1.1);*

*cout << endl;*

*return 0;*

*}*

*output:*

*x = 1 count = 0*

*x = 1 count = 1*

*x = 1.1 count = 0*

**Note**: We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of template. The important thing to note about non-type parameters is, they must be const. Compiler must know the value of non-type parameters at compile time. Because compiler needs to create functions/classes for a specified non-type value at compile time. Following is another example of non-type parameters.

#include <iostream>

using namespace std;

template <class T, int max>

int arrMin(T arr[], int n)

{

   int m = max;

   for (int i = 0; i < n; i++)

      if (arr[i] < m)

         m = arr[i];

   return m;

}

int main()

{

   int arr1[]  = {10, 20, 15, 12};

   int n1 = sizeof(arr1)/sizeof(arr1[0]);

   char arr2[] = {1, 2, 3};

   int n2 = sizeof(arr2)/sizeof(arr2[0]);

   cout << arrMin<int, 10000>(arr1, n1) << endl;

   cout << arrMin<char, 256>(arr2, n2);

   return 0;

}

Ouput: 10,1

**Generic Programming with Function Templates:**

**//Single Type Template Parameter**

• We can replace type with we want to generalize with a name, say T

T max( T a, T b){

return (a>b) ? a: b ;

Above one will not compile

• We need to tell the compiler that this is a Template Function

• We also need to tell that T is a Template parameter

template <typename T>

T max( T a, T b){

return (a>b) ? a: b;

d) We may also **class** instead of "typename"

template <class T>

T max( T a, T b){

return (a > b) ? a: b;

}

Notice that type MUST support the ">" operator either natively or as an overloaded operator( Operator >)

With Steps c and d compiler won't generate any code, if we use the templates in the code with actual function , then compiler generates

We can call the max function in 2 ways:

Int a = 5, b=7

Max(a,b); //Compiler deduce the types based on the type the arguments.

Max<int>(a,b); //we can explicitly mention the type

**Multiple Template Parameters: (mutlple types as function parameters):**

Template <class K, Class V>

Func(class K, class V){

cout << K << " " << V << endl;

It can be called as func<int, double>(10, 20.4) or func(10,20.4)

**Generic Programming with Class Templates:**

• Similar to function templates, but at the class level,

• Allows plugging in any data type.

• Compiler generates appropriate class from the blue print.

**Containers:**

a) Data structures that store object of almost any type (Template based classes)

b) Each container has member functions(some are specific to the containers and other all available to the all containers)

c) Each container has an associated header file (#include <container\_type>)

**Common methods:**

a) Default constructors (initializes empty container)

b) Overloaded constructors (initializes container with many options)

c) Copy constructors (Initializes a Container as a copy of another container)

d) Move constructors (Move existing container to the new container)

e) Destructor

f) Copy Assignment (operator=) , copy one container to another

g) Move Assignment (operator=), Move container to another.

h) size (return the no. of elements in the container)

i) empty( return Boolean – to check empty or not)

j) insert(Inerst an element into the container).

k) operator< and operator<= (compare, return Boolean)

l) operator> and operator>= (compare, return Boolean)

m) operator== (compare, return Boolean)

n) swap (swap the elements of 2 containers)

o) erase (remove element(s) from a container)

p) clear (remove all elements from a container)

q) begin and end (return iterator to begin and end)

r) rbegin and rend (return reverse iterators to first element or end)

s) cbegin and cend (return constant iterators to the first element or end)

t) crbegin and crend (return constant reverse iterators to first element or end)

What types of elements can we store in containers?

a) A copy of the element will be stored in the container (all primities Ok).

b) Element should be

i) Copyable and Assignable (copy constructor / copy assignment)

ii) Moveable for efficiency (move constructor / move assignment)

c) Ordered Associative containers must be able to compare elements

operator< , operator ==

**Iterators:**

a) Allows abstracting an arbitrary container as a sequence of elements.

b) They are objects that work like pointers by design

c) Most container classes can be traversed with iterators(except the stack and queue)

Iterator declaration:

a) Iterators must be declared on the container type they will iterate over

container\_type::iterator\_type iterator\_name;

std::vector<int>::iterator it1;

std::list <std::string>::iterator it2;

std::map<int,int>::iterator it3;

std::set<char>::iterator it4;

vec.begin(). points to the first element of the container

vec.end() points to the location after the last element, its not the last element

std::vector<int>::iterator it1 = vec.begin();

or

auto it = vec.begin();

while(it != vec.end()){

cout << \*it << endl;

++it;

}

//This is how Range based for loop works.

For(auto it = vec.begin(); it! = vec.end();++it){

Std::cout << \*it << endl;

}

Operations with iterators:

a) ++it, it++, (it = it1)(assignment), \*it (Dereference), it-> (Arrow operator, input an output type operator),

it == it1(comparison ), it != it1 , --it, it--, (it + k) or (it += i) (increment by specific value).

**Reverse Iterators:**

a) Works in reverse

b) last element is the first and first is the last

c) ++ moves backward , -- moves forward

*std::vector<int> vec {1,2,3};*

*std::vector<int>::reverse\_iterator it = vec.begin();*

*while(it != vec.end()){*

*cout << \*it << endl;*

*++it;*

*}*

*//3 2 1*

**Algorithms: (#include <algorithm>**

a) Algorithms works on sequence of container elements provided to them by an iterator.

b) STL has many common and useful algorithms.

c) too many describe in this section (http://en.cppreference.com/w/cpp/algorithm)

d) Many algorithms require an extra information inorder to do their work.

i) Functors (Function objects)

ii) Function pointers

iii) Lamda expressions

e) Different containers support different types of containers

f) All STL algorithms expect iterators as arguments. (To determine the sequence obtained from the container).

**Iterator Invalidation:**

a) its possible iterators become invalid during the processing

An Iterator becomes invalidate when the container it points to changes its shape internally i.e. move elements from one location to another and the initial iterator still points to old invalid location.

Iterator invalidation in vector happens when,

• An element is inserted to vector at any location

• An element is deleted from vector.

b) Suppose we are iterating over a vector of 10 elements

i) And we clear the vector while iterating ? what happens.

Undefined behavior – Our iterators are pointing to invalid locations.

Both operator[] & at() provides random access to elements in vector in O(1) Complexity. But in case of out of range access operator[] causes undefined behaviour, whereas at() returns proper out\_of\_range exception. So, at() is more safe to use as compared to operator[].

**Note:**

**Iterator invalidation will not happen for the list container:**

**Example Algorithms:**

**A) Find (used to findout the 1st occurrence of an element).**

**(find on primitive data type)**

/\* Vector<int> v1 {1,2,3,4,5};

Auto loc = std::find(vec.begin(), vec.end(), 3);

While (loc != v1.end()){

Cout << \*loc << endl;

} \*/

**(If we need to use find on the User defined classes):**

• Find needs to be able to compare object

• Operator== is used and must be provided by our class.

Std::vector<Player> team

Player p {“hero” ,100, 12};

Auto loc = std::find(team.begin(), team.end(), p);

If(loc != team.end(){

Cout << \*loc << endl;

} operator << is called.

B) **for\_each :**

For\_each algorithm applies a function to each element in the iterator sequence.

Function must be provided to the algorithm as:

a) Functor (function object) : it exists even before the C++11

b) Function Pointer

c) Lamda Expression(c++11)

ex: for square of the element:

//Fucntor or function object

struct square\_e{

void operator()(int x){

return x\*x;

}

}

//Function pointer

Void square\_e(int x){

Return x\*x;

}

vector<int> v1 {1,2,3,4,5,6};

for\_each(v1.begin, v1.end(), square\_e); //Function object

for\_each(v1.begin,v1.end(), [](int x) {return x\*x};); //lamda

r

for\_each(v1.begin(), v1.end(), square); //Function pointer

C) std::count count the no. of occurrences of an element in a container

D) std::count\_if-- > count only if a condition or matches.

E) std::all\_of -> return true or false, checks if all the elements satisfy the condition in lamda if(std::all\_of(v1.begin(),v1.end(), [](int x){return x>0}))

F) std::transform , (ex: conversion from lower to uppercase), std::transform(str1.begin(),str1.end(), str1.begin(), ::toupper); here :: (Scope resolution means global operator).

G) std::sort (sort the elements in ascending order) ex: std::sort(vec.begin(), vec.end());

H) std::min\_element or std::max\_element, returns minimun element (ex: std::min\_element(arr1.begin(),arr1.end());//It returns an iterator to that element.

Std::array::iterator min = std::min\_element(arr.begin(),arr.end());

I) std::accumulate(sum of all elements of a array container)(header for this #include <numeric>)

std::array<**int**,5> a1 {{10,2,30,30,1}};

**int** sum = std::accumulate(a1.begin(),a1.end(),0);

**int** sum1 = std::accumulate(a1.begin(),a1.end(),15);

cout << "sum is" << sum << endl; //73

cout << "sum1 is" << sum1 << endl; //88

**Std:array:**  (C++11) “#include <array>”, its an object array compared to c array, we cannot convert this to pointer like c array. This array will not declare a pointer to the first element of raw array.

It’s a wrapper class around array.

We can use iterators and algorithms.

a) Fixed Size (size must be known at compile time).

b) Direct element access

c) provides access to underlying raw array

d) Use instead of raw arrays when possible.

e) All iterators available and do not invalidate.

**Initialization and assignment:**

Std::arry<int, 5> arr1 { {1,2,3,4,5}}; //c++11

Std::arry<int, 5> arr1 {1,2,3,4,5}; //c++14

Std::array<string, 3> stooges {

Std::string(“Larry”),

“Moe”,

Std::string(“curly”)

};

//We can assign values with assignment operator also like below:

Arr1 = {2,4,6,8,10};

//We can covert a raw array to STL array also.

**Difference b/w “at” and subscript operator “[]”:**

“At” method will do boundary checking and throw an excetion.

Whereas subscript operator will not throw an exception

**Common Methods:**

Std::arry<int, 5> arr1 { {1,2,3,4,5}}; //c++11

Std::arry<int, 5> arr2 { {2,4,6,8,10}}; //c++11

cout << “size” << arr1.size(); //5

arr.at(0); //1

arr[1]; //2

arr.front(); //1, it retruns a reference to 1st element

arr.back(); //5, it returns a reference to last element

arr.empty();//return true or false, if empty – true,

arr.max\_size(); // 5

arr.fill(10)// fill all elements all to 10

arr.swap(arr1); //swaps the 2 arrays

//Very important:

int \*data = arr.data(); //get raw array address

returns a pointer to the address of 1st element

\*data , return 1;

\*(data+1), return 2;

**deque vs vector:**

• Vector provides insertion and deletion at middle and end only. Whereas, deque provides operations for insertion at front, middle and end. That is, apart from push\_back() and pop\_back() APIs jus like vector, deque also has push\_front() and pop\_front() API to add and delete elements from front.

• Vector provides good performance while insertion and deletion at end only and bad performance for insertion and deletion at middle.

• Deque provides same kind of performance as vector for insertion & deletion at end and middle. Apart from that deque provides good performance for insertion and deletion at front also.

• As Vector stores elements contiguously, where as deque internally contains a list of memory chunks which store elements contiguously. Due this basic architectural difference between vector and deque following things happen,

• Performance of addition and deletion at end for vector is better than deque.

• No Iterator invalidation happens in deque for insertion and deletion at front and end because like vectors, deque doesn’t have to shift elements from one memory to another in case current allocated memory is not sufficient to store the newly added element.

• Iterator invalidation happens in deque just like vector, if insertion or deletion takes place in the middle.

• Just like vector, deque also supports random access operations i.e. operator [] and at() function. Although performance of random access in deque will be little slower than vector.

When to choose deque over vector:

One should choose deque over vector if he wants to either add or delete from both the ends like implementing a Queue.

**When to choose vector over deque:**

One should choose vector if insertion or deletions are required mostly in end like implementing a Stack.

Sorting User Defined Objects with std::sort:

Suppose we have a class Person with name and id as member variables. Now we want to sort a vector of class Person objects on the basis of id.

To that wee need to to overload < operator in class Person because std::sort algorithm uses this < operator for comparision while sorting.  
Lets see the code of class Person,

***class Person {***

***public:***

***std::string m\_name;***

***int m\_id;***

***Person(std::string name, int id) :***

***m\_name(name), m\_id(id) {***

***}***

***bool operator <(const Person & obj) {***

***if (m\_id < obj.m\_id)***

***return true;***

***else***

***return false;***

***}***

***};***

std::vector<Person> vecOfPersons = { Person("aaa", 7), Person("kkk", 3),

Person("ddd", 5), Person("abc", 2) };

std::sort(vecOfPersons.begin(), vecOfPersons.end());

std::cout << "Sorted Persons List based on ID\n";

std::for\_each(vecOfPersons.begin(), vecOfPersons.end(), [](Person & obj) {

std::cout<<obj.m\_id<< " :: "<<obj.m\_name<<std::endl;

});

**SET**

• Set and multiset are containers that facilitate the quick look up of keys in a container that stores them.

• The difference b/w the set and multiset are , the latter allows the duplicate values, wheras set allows only unique values.

• To facilitate quick searching, STL implementations of Set and multiset internally lookslike a binary tree **i.e elements inserted in the set and multiset are sorted on the insertion for quick lookups, it also means that , unlike in a vector where elements at a position can be replaced by other, an element in a given position in set cannot be replaced by a different value.**

Ex: #include <set>

std::set, or std::multiset

std::set<int> setint;

std::multiset<int> msetint;

std::set<Tuna> setTun;

std::set::const\_iterator siter;

std::multiset<int>::const\_iterator msiter;

4) Given that, set and multiset containers ,that sort the elements on the insertion, they use default predicate is std::less if you don’t supply sort criteria. It means set contains the elements in the sorted order.

• You create a binary sort predicate by defining a class with operator() that takes 2 values of the type contained in the set as input and returns true depending upon your criteria. One such sort predicate that sorts in the descending order oois the following:

template <typename T>

struct SortDescending{

bool operator()(const T& lhs, const T& rhs) const [

return (lhs>rhs);

}

• Multiset::count(<element>) is used to find the no. of duplicates passed to the count function.

**Map:**

• The map and multimap are key-value pair containers that allow for a lookup on the basis of key.

• The difference b/w the map and multimap are, latter allows the duplicate keys, whereas the former can store only unique keys.

• To facilitate quick searching, STL implementations of the map and multimap internally look like binary tree, this means that elements in the map and multimap are sorted on insertion.

**Instantiation:**

Map<keyType, valueType, Prdicate=std::less<keyType>> mapObject;

Multimap <keyType, valueType, Prdicate=std::less<keyType> mmapObject;

Here 3rd parameter, predicate is optional

When you supply only, key and value, ignoring third template parameter, map and multimap default to class std:less<> to define the sort criteria.

**Instantiation:**

Multimap<int, string> mmap;

Multimap<int, string> mmap1(mmap);//it’s a copy of another.

Mmap<int,string> mmap2(mmap.cbegin(), mmap.cend());

Map<int,string, ReverseSort<int>> mmap4(mmap.cbegin(), mmap.cend());

Tempate <typename keyType>

Struct ReverseSort{

Bool operator()(const keyType& key1, const keyType& key2){

{ return (key1 > key2);

}

**Inserting of elements: (insert)**

Ex:

Map<int,string> mapiS;

mapiS.insert(pair<int,string>(1,"shankar")) //this is from the std::pair class)

or std::pair<int,string> p1 {3, “Krishna”};

mapiS.insert(p1);

mqpiS.insert(make\_pair(2, "vrinda"));

mapiS[3] = "krishna";

mapiS.at(3) = “Gopala”; //It may throw an exception, if the key does not exist.

mapiS.insert(mapiS::value\_type(4, "mukunda"));

**To find a Key and Value:**

auto itr = mapIs.find("1);

if(itr != mapiS.end()){

cout << "the key" << itr->first << "the value is" << itr->second() << endl

//Find no. of pairs that have the same supplied key.

auto mitr = mapiS.find(1000);

if(mitr != mapiS.end()){

size\_t n = mapiS.count(key);

**Erase Elements:**

//To erase a particular key and or it will delete all pairs which has that key

mapiS.erase(key);

auto it = mapiS.find(“key”);

if (it != mapiS.end()){

mapiS.erase(it);

}

//Erase a range of elements from a map or multimap using iterators that supply the bound.

Mmap.erase(iLowerbuild, iUppoerbound);

//Erase a range from mmulitmap

Mmap.erase(mmap.lower\_bound(1000), mmap.upper\_bound(1000));

Mmap.**count(it->first or key); used to find how many duplicates of a key are present in the map;**

**Map.count(“key”);** //It returns 0, if the key is not found, it returns 1 if they key is found.

**Unordered map:** (Internally they implement the Hashtable i.e based on the hash of the key, the hashtable indices are calculated), we can the indice or index of a key using method “bucket” like

Int bukid = map.bucket(“key”);

Or auto it = map.find(key);

Int buketid = map.bucket(it->key);

#include <unordered\_map>

a) elements are unordered.

b) No duplicate elements are allowed.

c) No reverse iterators are allowed.

**Unordered Multimap:**

#include <unordered\_map>

a) elements are unordered.

b) Duplicates are allowed.

c) No reverse iterators are allowed.

**Deque: (Double ended queue): header (#include <deque>)**

a) Dynamic Size

• Handled automatically

• Can expand and contract as needed.

• Elements are NOT stored in the Contiguous memory.

b) Direct element access (constant time)

c) Rapid insertion and deletion at the front and back (constant time) unlike vector.

d) Insertion or Removal of elements (linear time).

e) All iterators available and may validate.

**Initialization and Assignment:**

a) std::deque<int> d{1,2,3,4,5}; (d.size() = 5, d.max\_size

b) std::deque<int> d1 (10,100); // ten 100s

c) std:deque<std::string> stooges {

std::string(“Larry”),

“Moe”,

Std::string(“Curly”)

}

e) d = {2,4,6,8,10};

f) Deque is implemented as **collection of Memory Blocks** ( or think of linked list of vectors), each block has elements in continuous memory. But blocks them selves are not in continguos memory.

g) std::deque<int> d{1,2,3,4,5};

d.size() = 5;

d.max\_size // a very large number

d.front(); //1

d.back(); //5

d.at(1);//2

d[2]; //3

Person p1 {“Larry”, 18};

Std::deque<Person> d;

d.push\_front(p1); // Add p1 to the back

d.pop\_back(); //Remove p1 from the back

d.push\_front(Person{“Larry”, 18}); //

d.pop\_front(); //Remove element from the front.

d.emplace\_back(“Larry”, 18);// add to the back efficient

d.emplace\_front(“Moe”, 24); // add to front.

**Note:** Main usage of the Deque is usage of the Palindrome. (like madam, radar)

**List and Forward List: (std::list and std::forward\_list) (#include <list>)**

a) Sequence Containers.

b) Non-Contiguous in memory

c) No direct access to the elements

d) its very efficient for inserting and deleting the elements once an element is found.

**Note:**  List is by default bi directional , whereas forward\_list is unidectional.

e) Dynamic Size ( i) list of elements, list is bi drectional(doubly linked list))

f) Direct element access in not provided.

g) Rapid insertion and deletion of the elements anywhere in the container (constant time).

f) All iterators are available and invalidate when corresponding element is deleted.

**Initialization and Assignment:**

a) std::list<int> l {1,2,3,45};

b) std::list<int> ll (10,100); // ten 100s

c) std::list<std::string stooges { std::string(“Larry”), “Moe”, std::string(“Curly”));

d) l = {2,4,6,8,10};

e) std::list<int> l {1,2,3,4,5}; , std::cout << l.size(); //5

std::cout << l.max\_size; // a very large number

std::cout << l.front(); //1

std::cout << l.back(); //5

f) elements can be inserted at both front and back. Even elements can be popped at both and front.

Person pl {“Larry”, 18};

Std::list<Person> l;

l.push\_back(pl);

l.pop\_back();

l.push\_front(Person {“Larry”, 18});

l.pop\_front(); //remove element

l.emplace\_back(“Larry”, 18); //Add to back efficient.

l.emplace\_front(“Moe”, 24);

**std::list – Methods that use the iterators: (We can use range based for loop to iterate through the list).**

std::list<int> l {1,2,3,4,5,};

auto it = std::find(l.begin(),l.end(), 3);

std::cout << \*it ; //3

it++;

std::cout << \*it ; //4

it--;

std::cout << \*it ; //3

**Note:** We can use range based for loop also to iterate through the list.

l.insert(it, 10) // 1,2, 10, 3, 4, 5

l.erase(it);// Erases 3 as iterator points to 3. Now ouput lookslike 1,2,10, 4,5

l.resize(2) // Now output of list becomes, 1,2

l.resize(5); // Now output will be 1 2 0 0 0 //Default remaining 3 elements will be initialized to zero.

**Forward List:** header file (#include <forward\_list> , std::forward\_list)

a) Dynamic Size

i) List of elements

ii) list unidirectional (singly-linked list)

iii) Less overhead than a std::list

b) Direct element access is not provided.

c) Rapid insertion and deletion of elements anywhere in the container(constant time).

d) Reverse iterators are not available. Iterators invalidate when corresponding element is deleted.

e) Only “front() i.e first element is accessed or inserted, no back() is available)

**Ex:**

Std::forward\_list<int> l {1,2,3,4,5};

~~Std::cout << l.size();~~ //Not available

Std::cout << l.max\_size(); // Returns how many max number of elements can be stored. A very large number

Std::cout << l.front() ; // 1

~~Std::cout << l.back(); // Not available~~

Person p1 {“larry”, 18};

Std::forward\_list <Person> l;

l.push\_front(p1); // Add p1 to the front

l.pop\_front(); // Remove p1 from the front

l.emplace\_front(“Moe”, 24) // add to front

auto it = std::find(l.begin(), l.end(),3);

l.insert\_after(it,10); // 1 2 3 10 4 5

l.emplace\_after(100) // 1 2 3 100 10 4 5

l.erase\_after(it) // Erases the 100, new: 1 2 3 10 4 5

l.resize(2); //1 2

l.resize(5); // 1 2 0 0 0

**#include <iterator> // for std::advance is defined in the iterator.**

**Template Specialization:**

Template in C++is a feature. We write code once and use it for any data type including user defined data types. For example, sort() can be written and used to sort any data type items. A class stack can be created that can be used as a stack of any data type.  
*What if we want a different code for a particular data type?* Consider a big project that needs a function sort() for arrays of many different data types. Let Quick Sort be used for all datatypes except char. In case of char, total possible values are 256 and counting sort may be a better option. Is it possible to use different code only when sort() is called for char data type?  
*It is possible in C++ to get a special behavior for a particular data type. This is called template specialization*.

*// A generic sort function*

*template <class T>*

*void sort(T arr[], int size)*

*{*

*// code to implement Quick Sort*

*}*

*// Template Specialization: A function*

*// specialized for char data type*

*template <>*

*void sort<char>(char arr[], int size)*

*{*

*// code to implement counting sort*

*}*

**How does template specialization work?**  
When we write any template based function or class, compiler creates a copy of that function/class whenever compiler sees that being used for a new data type or new set of data types(in case of multiple template arguments).  
If a specialized version is present, compiler first checks with the specialized version and then the main template. Compiler first checks with the most specialized version by matching the passed parameter with the data type(s) specified in a specialized version.

**Explicit Keyword:**

in C++, if a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor because such a constructor allows conversion of the single argument to the class being constructed.

We can avoid such implicit conversions as these may lead to unexpected results. We can make the constructor explicit with the help of *explicit keyword*. For example, if we try the following program that uses explicit keyword with constructor, we get compilation error.

A constructor that is not declared with the specifier explicit and which can be called with a single parameter (until C++11) is called a *converting constructor*.

Unlike explicit constructors, which are only considered during direct initialization (which includes explicit conversions such as static\_cast), converting constructors are also considered during copy initialization, as part of user-defined conversion sequence.

It is said that a converting constructor specifies an implicit conversion from the types of its arguments (if any) to the type of its class. Note that non-explicit user-defined conversion function also specifies an implicit conversion.

Implicitly-declared and user-defined non-explicit copy constructors and move constructors are converting constructors.

**Copy elision:**

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*

***Why copy constructor is not 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”. So the statement

B ob = "copy me";

should be broken down by the compiler as

B ob = B("copy me");

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:

aashish@aashish-ThinkPad-SL400:~$ g++ copy\_elision.cpp -fno-elide-constructors

aashish@aashish-ThinkPad-SL400:~$ ./a.out

Constructor called

Copy constructor called

If “-fno-elide-constructors” option is used, first default constructor is called to create a temporary object, then copy constructor is called to copy the temporary object to ob.

**Const key words: (In class):**

a) Pass args to class member methods as const.

b) We can also create const objects.

c) What happens if we call member functions on const objects.

d) const-correctness

**Creating a const object:**

a) villain is a const object, so its attributes cannot change.

Const Player villain {“ravan”, 100,15}; // So we cannot change any value here.

What happens if we call member methods on const objects.

Void displayplayername(const Player& p){

Cout << p.getname() << endl;

}

Displayplayername(villain); //Error (Here compiler assumes that get\_name method migh change value of the Player obects data).

Solution for above problem is: declar getname function as const as below

Class Player{

Private:

Public:

Std::string getname() const;

}

**Variadic function templates in c++:**

Variadic templates are template that take a variable no. of arguments. Variadic function templates are functions which can take multiple no. of arguments.

**Syntax for a variadic function template:**

template <typename arg, typename… Args)

return\_type functionname(arg var1, args… var2)

*// C++ program to demonstrate working of*

*// Variadic function Template*

*#include <iostream>*

*using namespace std;*

*// To handle base case of below recursive*

*// Variadic function Template*

*void print()*

*{*

*cout << "I am empty function and "*

*"I am called at last.\n" ;*

*}*

*// Variadic function Template that takes*

*// variable number of arguments and prints*

*// all of them.*

*template <typename T, typename... Types>*

*void print(T var1, Types... var2)*

*{*

*cout << var1 << endl ;*

*print(var2...) ;*

*}*

*// Driver code*

*int main()*

*{*

*print(1, 2, 3.14, "Pass me any "*

*"number of arguments",*

*"I will print\n");*

*return 0;*

*}*

The variadic templates work as follows :  
The statement, **print(1, 2, 3.14, “Pass me any number of arguments”, “I will print\n”);**is evaluated in following manner :  
Firstly, the compiler resolves the statement into

cout<< 1 <<endl ;

print(2, 3.14, "Pass me any number of arguments",

"I will print\n");

Now, the compiler finds a print() function which can take those arguments and in result executes the variadic print() function again in similar manner :

cout<< 2 <<endl ;

print(3.14, "Pass me any number of arguments",

"I will print\n");

Again, it is resolved into the following forms :  
(\*)

cout<< 3.14 <<endl ;

print("Pass me any number of arguments",

"I will print\n");

(\*)

cout<< "Pass me any number of arguments" <<endl ;

print("I will print\n");

(\*)

cout<< "I will print\n" <<endl ;

print();

Now, at this point the compiler searches for a function overload whose match is the empty function i.e. the function which has no argument.

This means that, all functions that have 1 or more arguments are matched to the variadic template and all functions that with no argument are matched to the empty function.

**Parameter Pack:** A template parameter pack is a template parameter which accepts zero or more template arguments(non-types, types or templates).. A function parameter pack is a function parameters that accepts

zero or more function arguments.

A variadic class template can be instantiated with any number of template arguments:

Ex:

**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 2 arguments; int an float

Tuple<0> error; //error,: 0 is not a type

A variadic function template can be called with any no. of arguments.

Ex:

template <typename… Types>

void f(Types … args);

f();

f(1);

f(2,1.0)/

In a primary class template, the template parameter pack must be the final parameter in the template parameter list. In a function template, the template parameter pack may appear earlier in the list provided that all following parameters can be deduced from the function arguments, or have default arguments:

template<typename... Ts, typename U> struct Invalid; // Error: Ts.. not at the end

template<typename ...Ts, typename U, typename=void>

void valid(U, Ts...); // OK: can deduce U

// void valid(Ts..., U); // Can't be used: Ts... is a non-deduced context in this position

valid(1.0, 1, 2, 3); // OK: deduces U as double, Ts as {int,int,int}

**Static Library(.a file) Vs Dynamic Library(.so files):**

When a C program is compiled, the compiler generates object code.

a) After generating the object code, the compiler also invokes linker. One of the main tasks for linker is to make code of library functions (eg printf(), scanf(), sqrt(), ..etc) available to your program.

b) A linker can accomplish this task in two ways, by copying the code of library function to your object code, or by making some arrangements so that the complete code of library functions is not copied, but made available at run-time.

**Static Linking and Static Libraries** is the result of the linker making copy of all used library functions to the executable file. Static Linking creates larger binary files, and need more space on disk and main memory. Examples of static libraries (libraries which are statically linked) are, ***.a*** files in Linux and ***.lib***files in Windows.

**Static Linking and Static Libraries** is the result of the linker making copy of all used library functions to the executable file. Static Linking creates larger binary files, and need more space on disk and main memory. Examples of static libraries (libraries which are statically linked) are, ***.a*** files in Linux and ***.lib***files in Windows.

**This Pointer:**

a) It’s a reserved key word.

b) Contains the address of the current object, so it’s a pointer to the object.

c) Can only be used in class scope.

d) All member access is done via this pointer.

e) Can be used by the programmer.

a) To access data member and methods.

b) To determine if 2 objects are same or not.

c) Can be dereferenced (\*this) to yield current object.

**std::tuple:**

It’s a class template , std::tuple is a fixed size collection of heterogeneous values. It’s a generalization of the std::pair

**Template parameters**

Types...

-

the types of the elements that the tuple stores. Empty list is supported.

**Member functions**

(constructor)

constructs a new tuple  
(public member function)

operator=

assigns the contents of one tuple to another  
(public member function)

swap

swaps the contents of two tuples  
(public member function)

**Non-member functions**

make\_tuple

creates a tuple object of the type defined by the argument types  
(function template)

tie

creates a tuple of lvalue references or unpacks a tuple into individual objects  
(function template)

forward\_as\_tuple

creates a tuple of forwarding references  
(function template)

tuple\_cat

creates a tuple by concatenating any number of tuples  
(function template)

std::get(std::tuple)

tuple accesses specified element  
(function template)

operator==operator!=operator<operator<=operator>operator>=operator<=>

(removed in C++20)(removed in C++20)(removed in C++20)(removed in C++20)(removed in C++20)(C++20)

lexicographically compares the values in the tuple  
(function template)

std::swap(std::tuple)

(C++11)

specializes the std::swap algorithm  
(function template)

**Helper classes**

tuple\_size

obtains the size of tuple at compile time  
(class template specialization)

tuple\_element

obtains the type of the specified element  
(class template specialization)

std::uses\_allocator<std::tuple>

(C++11)

specializes the std::uses\_allocator type trait  
(class template specialization)

ignore

placeholder to skip an element when unpacking a tuple using tie  
(constant)

**Deduction guides(since C++17)**

**Notes**

Until N4387 (applied as a defect report for C++11), a function could not return a tuple using copy-list-initialization:

std::tuple<int, int> foo\_tuple()

{

return {1, -1}; // Error until N4387

return std::tuple<int, int>{1, -1}; // Always works

return std::make\_tuple(1, -1); // Always works

}

 Kubernetes Documentation:

https://dzone.com/articles/kubernetes-in-production-best-practices-to-follow?edition=645291&utm\_medium=email&utm\_source=dzone&utm\_content=Kubernetes%20in%20Production&utm\_campaign=

**GDB TOOL:**

a) Most important step to debug a binary with gdb is, Compile your code/source file(.c or .cpp) with the -g option as below:

g++/gcc shankar.cpp -o shankar -g

b) Once the binary is generated, run the command **“gdb <binary>”,** to go the next step, one more option “add a argument -q” to hide the some trivial information like : “**gdb <binary> -q”**

c) After the above step , if you want to add a Break Point, add using the command “b main”(break option at main) or “**b <functionname>**” or “**b <linenumber**>”

d) Give command **“run” or “r”** to start executing the binary, it will execute till the “Break point” , after that you can give bt, which will give the “Frames” info. Give command **“list”** command to see the actual source code. and give **“continue”** command to go the next execution.

**Note**: After **selecting a frame (ex: frame 2), then give the command “list”** which will show the corresponding source code.

After listing the source code, then **print the variable values using the command**:

ex: **p n (print value of n) during this time.**

e) After selecting a frame **“info frame”** to give more on Frames.

f) **“info variables”** to list all global and static variables.

g) **“info breakpoints”,** **“info locals”(**gives the local variables information of current frame), **“info functions”**

**h) “info args**” prints the arguments of the current stack frame**.**

**h)** give command **“help info locals”** to get the detailed information of command.

j) “info registers” (or “i r” ) gives the register information. (In output 1st column, register name, 2nd column – register value in hexadecimal, 3rd column/ register value in decimal).

**Note:** “variables which are declared **with “register” key word are stored in the “CPU Registers” they don’t have the address**. Register variables don’t use Virtual Memory , they use directly CPU registers.

register int k = 20;

K) **keyword “n” is next** to go to next line of execution in the gdb or execution. wheras “c” command is used to continue after break point.

l) Command/Keyword “set” is used to set the value of the variable in the gdb.

**Conditional Break Points:**

a) as long as the break point is used, gdb or the debugger always stops at this point.

b) However sometimes it’s useful to tell the debugger to stop at a break point only if some condition is met.

like when the variable has a particularly interesting value.

c) You can specify the breakpoint condition when you set a breakpoint by appending the keyword if to a normal break statement

***break [position] if expression.*** (here “position” can be anything like “function name” or line number

ex: “b 10 if k==15”(break point at line no:10 if the value of the k is 15)

d) If you already set a break point, then use below statement to add the expression to it.

“<condition> bp\_number [expression]”

**Watch Points:**

a) Watch Points are similar to break points.

b) But Watch points are set on variables whereas BPs are set on line numbers/function names.

c) when those variables are read or written, the watch point is triggered and the program execution stops.

d) Whenever we are writing/assigning a value , watch is useful. When we are reading its not useful.

For “Read watch”: Use the command **“rwatch”**

e) For both **Read and write** watch , use the **“awatch”** command.

f) “info breakpoints” shows the break points information also along with Watch point information.

g) How to disable watch point, **“disable 2”** (2 here is watch point/break point information. Delete a break point **“delete 2”**

h) How to enable watch point, “enable 2”

**GDB-TUI: (Text User Interface):**

a) Instead of running binary with gdb command, run it as “**gdbtui <binary**>”, which will open a separate window in the top to view the source code and break points information.

**How to redirect the output of the GDB logging to a file:**

a) Use command, -> set logging on , it will automatically redirect the output to a file called gdb.txt

b) set logging off //Disable logging

c) set logging file file //Change the name of the logging file (instead of default log file gdb.txt)

**You can attach GDB to a running process using the command:**

Steps:

a) just run the “sudo gdb” on linux terminal

b) From gdb terminal, run the command “shell ps -eaf | grep -i <process id>

c) Give the command “attach <processed>

d) Give the command “where” which will give the “back trace of the process where it was at the time of executing this”

e) Detach the process from the gdb, give the command “detach <processed>”

**To check the “Assemble” of the C code:**

a) Use the command “disassemble” from the gdb terminal.

• disassemble main

**Start Command:** Used to execute put a break point at the main and automatically execute the run(. r) command.

**If Multiple files are there in the binary , how to keep break point for a particular file:**

a) break <filename>:<functioname/filenumber>

**MakeFile Directives:**

The **make** program allows you to use macros, which are similar to variables. Macros are defined in a Makefile as = pairs. An example has been shown below −

MACROS = -me

PSROFF = groff -Tps

DITROFF = groff -Tdvi

CFLAGS = -O -systype bsd43

LIBS = "-lncurses -lm -lsdl"

MYFACE = ":\*)"

Special Macros

Before issuing any command in a target rule set, there are certain special macros predefined −

• $@ is the name of the file to be made.

• $? is the names of the changed dependents.

For example, we could use a rule as follows −

hello: main.cpp hello.cpp factorial.cpp

$(CC) $(CFLAGS) $? $(LDFLAGS) -o $@

Alternatively:

hello: main.cpp hello.cpp factorial.cpp

$(CC) $(CFLAGS) $@.cpp $(LDFLAGS) -o $@

In this example, $@ represents *hello* and $? or $@.cpp picks up all the changed source files.

There are two more special macros used in the implicit rules. They are −

• $< the name of the related file that caused the action.

• $\* the prefix shared by target and dependent files.

Common implicit rule is for the construction of .o (object) files out of .cpp (source files).

.cpp.o:

$(CC) $(CFLAGS) -c $<

Alternatively:

.cpp.o:

$(CC) $(CFLAGS) -c $\*.c