Unformatted learning notes

Table of contents

**Object Types**

**Functions**

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

**Strings**

String literals / const data is stored in read only data segment in C, therefore returning them from function is fine and doesn't get destroyed.

> While modifying the string literal, compiler wont complain but on execution gives SIGSEGV. In case of string array it is stored on stack

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

> Global/Static either uninitialized or '0' initialized both are stored in .bss segment

Compiler removes /\* \*/ but not // comment

Variables inside enums are treated as constants and using an enum value common which is common in two enums will create conflict.

> Bitwise operators are not allowed on floating point numbers because bitwise operations are allowed on numbers which are value-represented. In case of floats this is undefined

> Expressions are evaluations when done

> Space matters when doing typedef

> int is updated to unsigned int when comparison or in exprssion evaluation

> Statics have to be defined outside the class just because of one definition rule. If that was allowed every place where class is declared will get initialization

Take care of expression such as : **for** (i = -1; i < ARRAY\_SIZE(arr) - 1; ++i)

Because here macro will return size of type size\_t and condition mght fail ( -1 ~ 11111111111)

#error

macro specified if there is no argument that matches wirth

fseek() function should be preferred over rewind since it gives a sort of return value at least

wx”, “wbx”. When x is used with w, fopen() returns NULL if file already exists or could not open

**Volatile variables**

Hardware registers in peripherals : Like reading from a data port from which we might have to read variables on the fly

Variable referenced within ISR

Variables shared by multiple threads

> No optimization is performed on volatile variables

To pass value by reference to a thread use stf::ref as rep. below

std::thread t1 { functor, std::ref(var1), std::ref(var2) };

ISR cannot return a value

They cannot be passed parameted because they are called async and there is no ep

swap nibbles in byte

( (x & 0x0F)<<4 | (x & 0xF0)>>4 );

manacher Algorithm

Find longest palindrome from a string

Approach 1 : For every character move in both the directions and compare till incompare found.

ternary search tree is special case of trie with less child pointers and in which there is common prefix and same is used in auto completion feature of the web browsers/spell checks

C++ mangles the function so whent a linking to any C library it fails.

Size of blank class name is 1 so that address of 2 objects are different.

Types of boolean results are different in C and C++

**References**

alias of variable..any change will reflect in this 2

basically to show operator overloading

> References should be initialized in function initialization list

> Reference cant be modified

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

Static functions can be duped using fn pointers

reference cannot be assigned consts

Its ok to return reference of static variable from a function also any modifications made to the returned variable will be reflected in that variable.

**Classes**

If a derived class doesn't implements virtual function of base class, then it also becomes abstract and will throw error in case its object creation is tried.

> Static member functions can be called by the class object, but static members and static functions are not accessible from that static function because when called nu object its just called as global function and no this pointer is passed.

**>** A const object can only call const functions (not any other ) because in that case “const this” is passed as pointer.

> A class declaration can contain static object of self type

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

Constructors/Destructors

Argument of a copy constructor can have ptr as an argument but that wont be good because they can be nullptr while reference cannot

We can have main() fn overloaded inside a class

> Throwing an exception from constructor is the best way to cleanup in case object is not fully constructed

> Throwing an exception from destructor is dangerous, should be avoided and might result into the crash in case stack unwinding is in place..In that case 2 exceptions are propagating..really dangerous

class A{

B b;

public:

A(B x){

b = x;

}

};

steps here

-call B constructor

-call A default ctor

-assignment operation

-Destructor for X

* When we call constructor explicitly then compiler creates a temporary object and deletes that immediately
* One ctor cannot call another, that is because the order of member objects in the class body is critical

try catch in initialization list

class Foo

{

Foo() try : \_str( "text of string" )

{

}

catch ( ... )

{

std::cerr << "Couldn't create \_str";

// now, the exception is rethrown as if we'd written

// "throw;" here

}

};

> C++ is strongly type in case of exception handling also, a catch block written for a char cannot handle an integer

> Using this in constructor is not a problem

> All data members are sure to be fully constrcuted before body of ctor starts

> constructor cannot be static member function

> You cannot initialize static member in initializer list because it has to be defined outside.

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

* Deleting a void pointer doesn't calls the destructor
* Making a destructor private will result into compiler error if object is allocatedon stack

> An inherited protected member cannot be initialized by the derived class

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

**Methods Inside Classes**

In fact, all the functions defined inside the class are implicitly inline

MACROS cannot access private variables of d class

Virtual function cannot be inline

> static member function cannot be const and volatile because these keywords are referred for an object while static function don't deal with them.

Advantages of initializer list:

> References and const can be initialized in this.

> Prevents assignment operator to be called which unnecessary create temporary

Optimization + Memory Issues/Solutions

Copy Elision : prevents unnecessary copying of objects:

C c1(42); // direct-initialization, calls C::C(42)

C c2 = C(42); // copy-initialization, calls C::C( C(42) )

new operator never returns NULL it throws exception std::bad\_alloc

std::set\_new\_handler

> Compiler will do nothing on -> delete p if p is NULL and same is with free()

this as a parameter is passed to ctor

> delete without '[]' in case of array dont give any err its a logical bug BTW.

C++11 provides the concept of unique\_ptr and shared\_ptr where a unique\_ptr cannot be copied because its copy constructor and assignment operators are explicitly deleted.

get shared\_ptr sp by weak\_ptr wp using lock

shared\_ptr<Object> sp = wp.lock();

pointer.reset(); //It resets the memory pointed by the given smart ptr

Copying and assignment is there in shared\_ptr

class A{};

std::shared\_ptr<A> ptr(new A());

std::shared\_ptr<A> ptr2;

ptr2 = ptr1; //possible with shared\_prt but not with unique\_ptr

ptr2.reset(); //memory still there because ptr1 is pointing to it

std::weak\_ptr<int> wp1 = p1; //p1 owns the memory.

std::shared\_ptr<int> p2 = wp1.lock(); //Now p1 and p2 own the memory.

Use weak\_ptr to check if memory is still deleted or not...Weak ptrs help eliminate circular references.

Use make\_shared to avoid double memory allocations

shared\_ptr<Thing> p(new Thing); // ouch - two allocations

you would write:

shared\_ptr<Thing> p(make\_shared<Thing>()); // only one allocation!

You can use reset() method to return the weak pointer to its empty state

unique\_pointer cannot be assigned because they have private copy-ctor and cop-assn functions.

If u really wants to then use a move construct

like

unique\_ptr<Garbage> p2 = std::move(p1)

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

**RAII**

“*RAII can be shortly explained as "Resource themselves are responsible for their cleanup where construction should be given to an object's constructor and destructor should cleanup the object even in case exception is thrown."*

*It basically includes 2 things -*

*memory resource requirements and (shared\_ptr and unique\_ptr0*

*shared memory access (using mutexes and lock\_guards)*

resource allocation (acquisition) is done during

object creation (specifically initialization), by the constructor, while resource deallocation (release) is done during object destruction, by the destructor. If objects are destroyed properly, resource leaks do not occur.

**Smart Pointers**

A unique\_ptr cannot be copied because its copy constructor and assignment operators have been deleted. Though ownership can be transferred using std::move(ptr)

Example

#include<memory>

class A{};

std::shared\_ptr<A> ptr(new A(2));

std::shared\_ptr<int> p1(new int(5));

std::shared\_ptr<int> p2 = p1; //Both now own the memory.

Std::weak\_ptr<int> pw = p1;

std::shared\_ptr<int> p3 = pw.lock(); //It also hold the reference now

Problem can occur when circular references are generated

p1.reset(); //Memory still exists, due to p2.

p2.reset(); //Deletes the memory, since no one else owns the memory.

Example Smart pointer Class

#include <vector>

template<typename T>

class SP

{

T\* object;

size\_t\* count;

public:

SP(T\* data)

try {

// Use weird try around initializer list to catch new throwing.

// If it does we delete data to stop it from leaking.

:object(data)

,count(data ? new int(1) : NULL)

{ /\* This is the constructor \*/}

catch(...)

{delete data;}

SP():object(NULL),count(NULL){}

SP(SP<T> const& rhs): object(rhs.object), count(rhs.count) {

if (count) {++(\*count);}

}

SP<T>& operator=(SP<T> rhs) // Note implicit copy construction in rhs

{

// Using copy swap idiom for assignment.

// The copy is hidden because the parameter is pass by value.

this->swap(rhs);

return \*this;

}

void swap(SP<T>& rhs) throw()

{

std::swap(object, rhs.object);

std::swap(count, rhs.count);

}

~SP()

{

if ((count) && (--(\*count) == 0))

{

delete count;

delete object;

}

}

};

TODO : circular references removal

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

**Storage Classes**

> Rather than making the variable static in a file, place them in unnamed namespace, they will behave just like a static global with internal linkage only.

> Internal linkage means all the variables and functions that are accessible fron within the file only

> External linkage means all variable, functions which are accessible outside the transaltion unit.

> const variable declared in the class has static linkage by default

// in namespace or global scope

int i; // extern by default

const int ci; // static by default

extern const int eci; // explicitly extern

static int si; // explicitly static

// the same goes for functions (but there are no const functions)

int foo(); // extern by default

static int bar(); // explicitly static

> Names at the top-level namespace scope (file scope in C) that are const and not extern have external linkage in C, but internal linkage in C++.

> The static variables have to be initialized through constants...because these variables are initialized even before main.But this norm has be laxed in C++

> Static variables inside class doesn't contribute to its size, it is stored just like any other global variable

> In a class, constants are allowed when variable are declared using const keyword. Simple const is allowed in c++11 else it was only static const type that can be assigned constant values.

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

int a]2 = { 10, 20 };

The expression ++\*p has two operators of same precedence, so compiler looks for assoiativity. Associativity of operators is right to left. Therefore the expression is treated as ++(\*p). Therefore the output of first program is “arr[0] = 11, arr[1] = 20, \*p = 11“.

**Casting**

**Difference between static\_cast and C-cast**

> Static Cast wont allow casting between unrelated data type...like pointer to integer or integer to pointer.but its allowed in C-casting

> When you are cross casting ( take one base class and 2 derived classes - d1 and d2 ) and if you try cross casting d1 and d2...u'll get NULL ptr.

> dynamic\_cast uses info from vtable to determine

> Re-interpret cast is often used when you want to interface C/C++ API

**Concurrency**

Mutex and semaphores both are synchronization primitives while mutex is based on locking-unlocking mechanism, semaphore is based on signalling

Differences:

>Semaphores can provide sync. Services access to multiple resources while mutex only one

>Mutex is unlocked by the process that locked it while semaphore can be signalled by any other thread.

>

C++11 Concurrency

The technique of acquiring resources in a

constructor and releasing them in a destructor, known as Resource Acquisition Is Initialization

In std::mutex copy constructor and assignment operator are mentioned delete.

mutex m; // used to protect access to shared data

// ...

void f()

{

unique\_lock<mutex> lck {m}; // acquire the mutex m

// ... manipulate shared data ...

or simply

mutex m;

m.lock();

x = x+1;

m.unlock();

}

think of lock\_guard just like a entity that prevents memory leaks and actual work is done by mutex.

lock\_guard<mutex> m;

//Runtime polymorphism can also be implemented through references along with the pointers...

B b;

A &a = b;

**Polymorphism ( including Virtual Functions )**

Pure virtual functions make class to have partial vtable and object of such class which has partial information is meant to be prevented to create object of that particular class.

It is possible to define pure virtual function outside the class.

If you want to provide common function for all derived classes, then make the definition of that pure virtual function outside the class.

> Downcasting is used when you know the exact type of object being pointed by the base class pointer.

> Downcast is not allowed without using dynamic\_cast which u ensures that it is safe, because there may be another directly derived classes that base pointer might be pointing to

> **Downcasting is allowed only between base class and derived class relative.** When there is base class pointer pointing to derived class, it is not allowed when a base class pointer points to another derived class and you try to downcast that base class pointer to another derived class. While static\_cast is allowed without base pointer being pointing to another derived class. **See last point below**

> Upcasting slices the object because u wont be able to call derived class functions when used with base class pointer.

> Virtual function when called from base class ctors only point to the base class, because till then derived class ctor hasn't been called so virtual table hasn't been set up yet..

> When you make the base class destructor virtual, in that case you don't need to override the dtor options.

> If you want to make base class abstract but there is no pure virtual function you can make the dtor pure and virtual thats how things will change.

> Virtual base constructors are always called from the final leaf class. None of the other constructors for the virtual base are called

> Virtual funda starts from the class where first virtual function is defined

> When calling derived class member function sing base class pointers in case of rutime dispatch only those member functions are accessible which are defined in base class.

Here in this example, since a2 doesn't points to B object, dcast is not allowed while static\_cast works fine.

**JUST KEEP IN MIND DOWNCAST IS SUCCESSFUL ONLY WHEN BASE CLASS POINTER WAS POINTING TO DERIVED CLASS WHILE CREATING THE OBJECT, WHILE STATIC CAST JUST SEES THE POLYMORPHIC TYPES OF CLASSES.**

A \*a1 = new B;

A \*a2 = new A;

B \*b1 = dynamic\_cast<B\*>(a2);

B \*b2 = static\_cast<B\*>(a2);

cout<<b1<<" "<<b2<<endl;

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

\*this pointer is implicit to the object and can check the behavior just before construtor

\*Object slicing can be prevented by using ptrs

\*Problem with Double check locking is Sequnce re-ordering , normail sequence will be

-Allocated memory

-Create object in it

-Make ptr point it to that memory

But if compiler decides to flip statements 2 and 3 and one thread decides to stop after 3(Make ptr pointing) in that case object is not created and

second thread if try to enter will think the pointer is valid hence will be fatal because object is still has not been created in the memory yet.

The way is to make each thread call that singleton thread is to call getInstance() by each thread before beginning and cache that object.

Memory mapped I/O where RAM is used to store transmit data from peripheral devices to/from CPU this thing saves time for additional memory fetching

volatile keyword exact use when we are reading from a I/O signal where there is MMIO then accessing that memory might be optimized by the compiler

unsigned \*p = Address();

a = \*p;

b = \*p;

we want p to be changed dynamically because that will refer ro same port, compiler might optimize it and do b = a;

Difference between static\_cast and C-style cast

test.cpp:8:30: error: invalid static\_cast from type ‘char\*’ to type ‘int\*’

int \*p = static\_cast<int\*>(&c);

test.c:8:10: warning: initialization from incompatible pointer type

int \*p = &c;

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

Function template example : T& Array<T>::operator[](int index) {}

To prevent a deadlock maintain the lock order so that 2 thread shouldn't come at once. Have some ways to accessd down critical sections of code

smart ptr count ref

catch by val or ptr

type of itr

dlocking singleton can fail

Use return by value now, because by default std::move will be called

In case of exceptions, The recommended way is to throw by value and catch by reference.

When throwing an exception, throw an object by value. Avoid throwing a pointer, because if you > > throw a pointer, you need to deal with memory management issues

> catch by reference as object slicing will give different result

Use to\_string(int) to convert integer to string and for reverse stoi(string)

delete this wont cause any problem unless and until that is used ur last usage of function

cout << x << y is an example of method chaining.

To find the inorder successor of tree go right then left->left\* in case the node has rt child else use loop where you have to fiund the parent whose left child ius that node. if node lies in left of given node make that as successor

if you go further left then make that also left node else go right->right and no need to modify anything.

Order statistics tree has number of elements in the node in this when treeSize+1 > k. go left and look for treeSize+1-k else go right and look for k +1- (treeSize)

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

static Point rectangular(float x, float y); // Rectangular coord's

static Point polar(float radius, float angle); // Polar coordinates

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

In some cases, mostly for g++ when calling by value

Foo x = sea(1, 2);

Foo sea(int i, int j){

Foo t = new Foo(3, 5);

return t;

}

//As per the standard 3 objects will be created inside sea, in function stack and while assigning to x

But compiler optimizer it as

Foo t;

sea(&t, 1, 2)

void sea(\*ptr, int i, int j){

ptr = new Foo(i, j);

//do with ptr;

return;

}

It is only applied when all path return the same type of variable.

//So in this case, ptr is created in function itself, no copies...in another moderate optimizations. return by value is changed to pass by pointer.

Static order initialization fiasco depends on the dependency among 2 static vars, to prevent this use "Construct On First Use Idiom", Solution to this is to return the first one which is being used by reference and let the function be called

x().goBowling();

When the program exits all heap is reclaimed by CPU.

If ctor fails, throw and exception

Application of re-throwing and exception is to add traces while it is being trasnsmitted

\*vptr is associated with each object

**Networking**

A socket is identified by :

Client IP : Client Port and Server IP : Server Port

To communicate with the server the client opens a new port on its side, it opens a socket on its side on its IP and waits for the connection

as soon as it gets the connection it creates a new socket combining the address of client and its own address

TCP 3 way handshake takes place in the listen function - Here Hanshake packets are sent in terms of Sync/Sync-Ack/Ack

In these handshakes there is a sequence number that tells whether the packets being sent are in sync or not.

However HTTP can use unreliable protocols such as

The Simple Service Discovery Protocol (SSDP) is a network protocol based on the Internet Protocol Suite for advertisement and discovery of network services and presence information. the User Datagram Protocol (UDP), for example in Simple Service Discovery Protocol

listen on UDP and TCP on the same port

Operator precedence

++ postfix

++ prefix and \* (deref)

**C Internals:**

process segment in following order top to bottom

stack, heap, bss, data, text

.bss all 0's (uninitialized or initialized to 0) along with the static local variables which are not declared

bss segment is non-zero because some of the libraries are present, to see actual use the option -nostdlib or -nostartfiles

use objdump -S ./a.out to see the disassemble code.

Adding a subroutine's entry to the call stack is sometimes called "winding"; conversely, removing entries is "unwinding"

In C++,”this” pointer along with function arguments in the call stack when invoking methods

In pushing order following is stored for a subroutine on a call stack:

“Parameters”, “Return Addresses”, “Locals”

Spilling is the method sto store variable from register to local memory while the reverse is called filling

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

typedef doesn't declare an instance of a variable, it declares a type (type alias actually),

static is a qualifier you apply to an instance, not a type, so you can use static when you use the type, but not when you define the type. Like this..

typedef int int32;

static int32 foo;

The comma operator is evaluated left to right and result is value of right thing

sizeof is an operator because it is mentioned in standards

you cant take address of it, u can use it without parentheses,

sizeof operator is evaluated at compile time

sizeof(func(2)) only prints the sizeof value returned by the function

--------------------------------------------- **TCPL** ------------------------------------------------------------

static variables have internal linkage by default...

You cannot specify storage to class

We cannot take address of register variables and it cannot be global

since auto variables don't exist at program load time they can't be initialized by the runtime startup code

Compiler prevents converting volatile int\* to int\* and vice versa is not allowed

making a variable const automatically makes it static in C++ but not in C

Static local variables are not stored in bss segment

Scope of a name is part of the program where that name can be used

Parameters are same as local variables

register variable informs compiler that the given variable will be heavily use

static variable remains to file only even if defined in header file and used in c file

C standard is defined in such a way that it it possible for it to compiler one file independently...when declared register global in one file, there is now way it can find out that there is another file that has declared that variable as register global

Guarantee - static + global = 0

register + automatic - garbage

For external and static variables, the expression on RHS must be a constant expression

They are initialized just before the execution of program

using string to initialize a character array will automatically add extra '\0' to it

If array size allocated is smaller than initializer in that case warning is displayed and printed characters are which are intialized ones while in case of c++ error is generated ;)

C code, a definition without a storage class was an int definition and you cannot have statements as global

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

Bounded buffer problem. where there is a mutex and 2 semaphores full and empty, there is wait() and signal() function that operates upon these 2 functions

For producer, it will be like:

wait(empty);

acquire(mutex)

...

release(mutex);

signal(full);

for consumer problem:

wait(full);

acquire(mutex)

...

release(mutex);

signal(empty);

another problem for synchronization is reader writer problem many reader can enter the critical section or one writer can write

wait(mx);

if(reader == 1)

wait(wsm);

signal(mx);

...read operation...

wait(mx);

if(reader-1 == 0)

signal(wsm);

signal(mx);

for writer:

wait(wsm);

...write

signal(wsm);

you can access static functions with normal objects

there is nothing called static objects, statics can be ducked by anybody but they cannot anybody.

Usage of static and extern pointers

A static pointer could be used to implement a function that always returns the same buffer to the program, allocating it the first time it is called and example of the same is singleton pattern

char \* GetBuffer() {

static char \* buff = 0;

if ( buff == 0 ) {

buff = malloc( BUFSIZE );

}

return buff;

}

An extern (i.e. global) pointer could be used to allow other compilation units to access the parameters of main:

extern int ArgC = 0;

extern char \*\* ArgV = 0;

int main( int argc, char \*\* argv ) {

ArgC = argc;

ArgV = argv;

...

}

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

you can return multiple values from function using tuple

i.e

std::tuple<int,int> fun();

int a;

int b;

std::tie(a,b)=fun();

19161212.,()

Some operators return by value, some by reference. In general, an operator whose result is a new value (such as +, -, etc) must return the new value by value, and an operator whose result is an existing value, but modified (such as <<, >>, +=, -=, etc), should return a reference to the modified value.

in order to support operator chaining, the assignment operator must return some value. The value that should be returned is a reference to the left-hand side of the assignment.

You should return by reference if you are using += and return by value if it is +, - etc

Property of modulo operator:

(A+B)%m = A%m + B%m

(A\*B)%m = (A%m \* B%m) %m

Clustered index is index which is sorted and in which multiple column entries are clustered to give multiple data entries

A clustered index determines the order in which the rows of the table will be stored on disk

A non-clustered index has no effect on which the order of the rows will be stored

Add without addition operator - half adder logic

while(y != 0){

carry = x&y; //for subtraction it will be ~x&y

x = x^y;

y = carry<<1;

}

or return printf("%\*c%\*c", x, ' ', y, ' ');

Multiply without operator -

#define A x

#define B y

char arr[A][B];

use sizeof

This another O(n) which relies on the fact that if we n times multiply the matrix M = {{1,1},{1,0}} to itself (in other words calculate power(M, n )), then we get the (n+1)th Fibonacci number as the element at row and column (0, 0) in the resultant matrix.

In a specified execution sequence, certain points are there which are called sequence points which make sure that side effects of previous evaluations are guaranteed to be complete

— The end of the first operand of the following operators:

a) logical AND &&

b) logical OR ||

c) conditional ?

d) comma ,

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

> In C, a code in which function is called before its definition is fine

> Its not mandatory to initialize a \*const variable in C

> in C++, main must return 'int'

> Size of bool and char is different in C and C++

> In case of ref. var. you cant tell whether a given argument is passed by reference or by value, chance of a bug

> Protect object slicing by taking argument as reference parameter

> RT polymorphism is possible with references also like >> Bases &b = d; //where its Derived d

> Parameter declarations that differ only in a pointer \* versus an array [] are equivalent

> Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent.

> C++ allows member methods to be overloaded on the basis of const type

> C++ allows functions to be overloaded on the basis of const-ness of parameters only if the const parameter is a reference or a pointer

> class D : public B {

public:

using B::f; // make every f from B available

double f(double d) { cout << "f(double): "; return d+1.3; }

// ...

};

there are 2 types of caches

instruction and data caches

list<sqr2(X)> mylist2; // Wrong; sqr2 is not constexpr

Virtual function cannot be inline because call is resolved at RT

deleting NULL is undefined behaviour

**Program crashes when used to delete stack variable else it doesn't, might be UB**

If a non-static object is member then declaration of class is incomplete and compiler has no way to find out size of the objects of the class.

Size of an empty class is not zero. It is 1 byte generally. It is nonzero to ensure that the two different objects will have different addresses.

When you apply the const qualifier to a nonstatic member function, it affects the this pointer. For a const-qualified member function of class C, the this pointer is of type C const\*, whereas for a member function that is not const-qualified, the this pointer is of type C\*

delete operator works only for objects allocated using operator new

memory layout

stack -- High address

|

-

^

|

Heap

Uninitialized section

Initialized section ( read-only and read-write)

code section -- Low Address

To learn memory layout remember sorted order from low-high ( CIU )

Compiler creates a copy constructor if we don't write our own. Compiler writes it even if we have written other constructors in class

**To execute code before entering main you should declare a class, define a global object of it and do what you want in its constructor.**

Virtual constructors don't make sense, it is meaningless to the C++ compiler to create an object polymorphically

C++ deliberately specifies that binding a temporary object to a reference to const on the stack lengthens the lifetime of the temporary to the lifetime of the reference itself

We can have "return;" statement both in ctor and dtor

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

B ob = "copy me"; or B ob = B("copy me"); //copy initialization

as

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

BOLD\_\_\_\_BOLD

Function parameters cannot be declared constexpr.

constexpr cannot be applied to the declaration its only for the definition

You can make a class to have virtual ctor by having a Create method as static

It is possible to call destructor for local objects

Destructor is also called for the argument of function

When destructor is called explicitly then object is destroyed immediately

Base class friend functions and its ctor/dtor are not inherited inside the derived class.

vptr is related to object while vtable is related to class

base \*b;

derived d;

b = &d;

b->show(); //first derived class vptr is accessed and then show is searched in the list of functions which is called

Returning reference from operator overloading allows chaining and returning value is inefficient as it will create unnecessary temporaries

private member can be accessed in a friend function and also while defining function outside the class ... 'friend' keyword has to be dropped

The principle reason to make the return type of copy-assignment a non-const reference is that it is a requirement for "Assignable" in the standard.

In summary, the guidelines for the assignment operator are:

Take a const-reference for the argument (the right-hand side of the assignment).

Return a reference to the left-hand side, to support safe and reasonable operator chaining. (Do this by returning \*this.)

Check for self-assignment, by comparing the pointers (this to &rhs) because in case of pointers we actually delete the pointer of the left side which is also the right side so assigning things from right to left won't be fine as it is already deleted

Problem while overloading sizeof is that it is used to measure the size of pointer so decides how much to jump

Scope resolution and member access operators work on names rather than values. C++ has no syntax for writing code that works on names rather than values so syntactically these operators can not be overridden.

In C++, RTTI (Run-time type information) is available only for the classes which have at least one virtual function

We can make v.fn private but to access it we have to make main as a friend function of that class

Access specifiers are checked at compile time

We need to declare the body of the virtual function in case it is made pure virtual because in that case it will give erro when destruction takes place in reverse order

Prefer not to give default values to a function in inheritance because the value of the function contained inside th base class will be substituted

Implicit type conversion doesn’t happen for primitive types

The catch(...) must be the last catch block.

The process of removing function call entries from fn call stack is called stack unwinding

Constructors are not inherited, however assignment operator is.

**Local classes**

local class is defined in a function and can be used only in that function and not accessible outside.

local class cannot access local variable of function though its able to access external and static variables purely because of lifetime static class members have to be defined in global scope.)

Member functions of a local class have to be defined within their class definition and it cannot have static members also.

nested class can access private members of Enclosing class

for tinyURL typo questions

take 62 possible characters for the URL ... use 10 digit integer id for that URL and store <integer>:<Original URL> in database

// A simple base conversion logic

for (int i=0; i < shortURL.length(); i++)

{

if ('a' <= shortURL[i] && shortURL[i] <= 'z')

id = id\*62 + shortURL[i] - 'a';

if ('A' <= shortURL[i] && shortURL[i] <= 'Z')

id = id\*62 + shortURL[i] - 'A' + 26;

if ('0' <= shortURL[i] && shortURL[i] <= '9')

id = id\*62 + shortURL[i] - '0' + 52;

**Templates**

class templates:

It is possible to have default parameter types in the templates like

template<class T, typename U=char>

and then make a declaration like

Array<int>

Remove class scoping from friend function declaration

using namespace std; is bad because it might happen that there are 2 functions with same name that appear in 2 namespaces then there will a conflict

An identifier can be declared as often as you want.

You cannot have default argument for friend template

We can pass nontype parameters (parameters that are not data types) to class/function templates.

To check if a number is power of 2

If we subtract a power of 2 numbers by 1 then all unset bits after the only set bit become set;

n&(n-1) will give 0. The expression n&(n-1) will not work when n is 0. To handle this case also, our expression will become n& (!n&(n-1))

In the above case n&(n-1) gives '0' in first attempt itself when one bit is set, we can use the same if we have to find out number of set bits by

doing n = n&(n-1) 'x' times till n becomes 0 where x is number of bits set.

Also called nth order statistic or QuickSelect

function select(list, left, right, n)

if left = right // If the list contains only one element,

return list[left] // return that element

pivotIndex := ... // select a pivotIndex between left and right,

// e.g., left + floor(rand() \* (right - left + 1))

pivotIndex := partition(list, left, right, pivotIndex)

// The pivot is in its final sorted position

if n = pivotIndex

return list[n]

else if n < pivotIndex

return select(list, left, pivotIndex - 1, n)

else

return select(list, pivotIndex + 1, right, n)

comma as an operator returns the right operand after executing left side, its use is in for loop where there is inc. dec. of more than 2 indices

if (failure)

return (errno = EINVAL, -1);

The main advantage of paging over memory segmentation is that it allows the physical address space of a process to be noncontiguous

> left shift and right shift when applied on -ve numbers results in undefined behavior and it cannot be applied on floating numbers

The & operator can be used to quickly check if a number is odd or even

To find the missing number XOR of given list and then of 1->n XOR of 2 results gives us the missing number

When you do exit(0) then constructor is not called while it is called on executing return 0 but it will be called when the variable is static

Variable names can be omitted in default arguments

No need of taking address of function to assign it to function pointer

Array of function pointers

In structure, a bit field variable cannot be static

unsigned int i = 1;

char \*c = (char\*)&i;

if (\*c)

printf("Little endian");

else

printf("Big endian")

Studying big and little endian mystery

Normally we ideate everything in terms of big endian, it means we visualize like right end we has LSB. To convert that to little endian just write the number in big endian form and reverse as it is and that is how number will be stored but while calculating in little endian style byte by byte take each tbyte and think of it stored in reverse order like if you stored 12 in big endian.. it will be like

00-00-0c-00

while in little endian it will be like

00-c0-00-00

0000-0000—0011-0000 like this value is 3072

We cannot have array of void data type

3) Array parameters are always passed as pointers, even when we use square brackets.

In C, arr, &arr, &arr[0] gives same output with the middle one interpreted as &arr ( Pointer to array , int \*p = &arr will give error )

A string literal initialization appends '\0'

C++ do array bound checking while gcc doesn't

assignig literal to [] gives one extra space for '\0' while it doesn't when assigning to a pointer

appVirtual funda starts from the class where first virtual function is definedend const to remove warning related to string literal assigned to a pointer

Array parameters are always passed as pointers, even when we use square brackets

You always have to keep 1 character extra while assigning string literal to a constant array

char \*str = "GfG";

In the above line “GfG” is stored in a shared read only location, but pointer str is stored in a read-write memory.

Only and only reference can act as an lvalue in case the variable is returned from a function.

you can do free with realloc passing size as 0 realloc(ptr, 0)

To print some of 2 numbers without any operator use printf width specifiers

Assigning integer to enum results in compiler error directly

you cannot initialize members directly in structure.

Designated Initialization allows structure members to be initialized in any order

Macros can have side effects , macro can undergo name conflicts

const in C cannot be used to build constant expressions.

Your own sizeof() : -> printf("%d", (int)(&x + 1) - (int)(&x + 0)); for T x;

Heap store and Heap : Both are conceptual names, where free store refers to memory area allocated by new, heap is allocated by malloc/calloc calls.

**Operating Systems**

**MultiThreading**

Thread local storage refers to the variables which are declared static/global but each thread maintains its own copy of that variable. A good example is errno. If a thread executes a system call and an error occurs, in that case it might happen that another thread can override that variable so in that case it makes sense to keep a local copy for the thread. In C++11 this type of TLS (thread local storage ) is now a storage class now which can be declared by prepended the variable type withthread\_local

## Synchronization

## Semaphores should be used in conditions like producer/consumer while condition variables in conditions where one thread is waiting for some condition to become true.

## Check if semaphore also allows broadcasting just like condition variable

Synchronization allows you to control program flow and access to shared data for concurrently executing threads.

The four synchronization models are mutex locks, read/write locks, condition variables, and semaphores.

* Mutex locks allow only one thread at a time to execute a specific section of code, or to access specific data.
* Read/write locks permit concurrent reads and exclusive writes to a protected shared resource. To modify a resource, a thread must first acquire the exclusive write lock. An exclusive write lock is not permitted until all read locks have been released. ( Check in present code )
* Condition variables block threads until a particular condition is true.
* Counting semaphores typically coordinate access to resources. The count is the limit on how many threads can have access to a semaphore. When the count is reached, the semaphore blocks.

thread can create a process

Like process if one thread is blocked another one can run

Thread do not need IPC

Because of the very nature, threads can take advantage of multiprocessors.

Context switching are fast when working with thread

To switch process, CPU has to maintain following things

>process State

>Scheduling info

>memory info

>Shared data

Kernel is not aware of the user level threads. While scheduling it allocates processor to processes not to user level threads. So threads of a process cannot run on multiple processes. Advantages is that no mode switching is required.

Default attributes of a thread :

* It is unbounded
* It is nondetached
* It has a a default stack and stack size
* It inherits the parent's priority
* A thread can be set whether it is cancellable or not and whether its cancellation is synchronized or instantaneous

Multiple threads cannot wait for a thread to terminate

you can detach a thread by calling pthread\_detach(thread\_id)

Thread can get its id by calling pthread\_self();

To compare 2 thread id's int pthread\_equal(pthread\_t tid1, pthread\_t tid2);

pthread\_setschedparam() to modify the priority of an existing thread

int pthread\_setschedparam(pthread\_t tid, int policy,

const struct sched\_param \*param);

Sample code to set priority of thread :

#include <pthread.h>

pthread\_t tid;

int ret;

struct sched\_param param;

int priority;

/\* sched\_priority will be the priority of the thread \*/

sched\_param.sched\_priority = priority;

/\* only supported policy, others will result in ENOTSUP \*/

policy = SCHED\_OTHER;

/\* scheduling parameters of target thread \*/

ret = pthread\_setschedparam(tid, policy, &param);

pthread\_once\_t once\_control = PTHREAD\_ONCE\_INIT;

[pthread\_once](https://computing.llnl.gov/tutorials/pthreads/man/pthread_once.txt)**(**once\_control, init\_routine)

pthread\_mutex\_trylock() can be employed to prevent deadlocks in the program

int pthread\_getschedparam(pthread\_t tid, int policy, struct schedparam \*param) gets the priority of the existing thread.

sched\_yield(); is called by thread to yield its execution in favor of another thread

A thread can set its cancel state as

int pthread\_setcancelstate(int *state*, int \**oldstate*); ( PTHREAD\_CANCEL\_DISABLE/ENABLE, NULL )

int pthread\_setcanceltype(int *type*, int \**oldtype*);

( PTHREAD\_CANCEL\_DEFERRED/ASYNCHRONOUS, NULL )

pthread\_join(pid, &status) is the only way to know whether thread was cancelled successfully or not

pthread\_cancel() is implemented using signals and can result into memory leaks

So to prevent that create routine to flush the threads by pushing cleanup routines which are popped in the reverse order.

Major steps took place when a pthread\_cancel(thread\_id) call is made:

1. Cancellation clean-up handlers are popped (in the reverse of the order in which they were pushed) and called. (See [**pthread\_cleanup\_push**](http://linux.die.net/man/3/pthread_cleanup_push)(3).)

A cleanup may be required if we want to release resources held up by thread like mutexes and other handles. These are not called when a thread makes a normal return even without calling pthread\_exit()

2. Thread-specific data destructors are called, in an unspecified order. (See [**pthread\_key\_create**](http://linux.die.net/man/3/pthread_key_create)(3).)

3. The thread is terminated. (See [**pthread\_exit**](http://linux.die.net/man/3/pthread_exit)(3).)

On Linux, cancellation is implemented using signals. Signals are fired up in the process queue.

pthread\_attr\_\* (&attr, <>); specifies a large number of API's o set various attributes of the thread

pthread\_attr.setscope(&attr, PTHREAD\_SCOPE\_SYSTEM); //bounded

pthread\_attr.setschedpolicy(&attr, SCHED\_OTHER);

pthread\_attr.setstacksize(&attr, stack\_size\_int);

pthread\_attr.setdetachstate(&attr, PTHRED\_CREATE\_DETACHED ); or PTHREAD\_CREATE\_JOINABLE

you should also destroy the attr structure like

pthread\_attr\_destroy(&attr);

Synchronization Objects

* Mutex
* Semaphore
* Condition Variables

TLS or thread local storage is an old mechanism of pthreads but new mechanism on C++11 (where it is a storage class, auto and register has been deprecated ) Here a global/static object is defined but each thread maintains a copy of each of that variable

Following are the associated functions

pthread\_key\_t key;

pthread\_key\_init(&key)

pthread\_setspecific(key, const void \*);

void \*buffer\_ptr = pthread\_getspecific(&key);

**Synchronization Objects**

**Mutexes**

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;//combines bottom 2

pthread\_mutexattr\_t mattr;

pthread\_mutex\_init ( &mutex, &mattr );

pthread\_mutex\_lock(&mutex);

pthread\_mutex\_unlock(&mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mp); //non-blocking but if successful it locks the mutex

**Mutex Attributes:**

int pthread\_mutexattr\_gettype(const pthread\_mutexattr\_t *\*attr*, int *\*type*);

int pthread\_mutexattr\_settype(pthread\_mutexattr\_t *\*attr*, int *type*);

type is PTHREAD\_MUTEX\_DEFAULT/NORMAL/RECURSIVE  
pthread\_mutexattr\_init(&mattr);

pthread\_mutexattr\_destroy();

int pthread\_mutexattr\_setpshared(pthread\_mutexattr\_t \*mattr, int pshared); // mattr scope is either PTHREAD\_PROCESS\_PRIVATE or PTHREAD\_PROCESS\_SYSTEM

pthread\_mutexattr\_getpshared(pthread\_mutexattr\_t \*mattr, int \*pshared);

pthread\_mutex\_destroy(&mutex);

pthread\_kill() will not kill a thread, it just passes a signal to it. If SIGTERM is passed process is killed

**Condition Variables**

Condition variables can be used to atomically block threads until a particular condition is true. Condition variables are **always** used in conjunction with mutex locks:

pthread\_cond\_t condition = PTHREAD\_COND\_INITIALIZER;

pthread\_condattr\_t condattr;

int pthread\_condattr\_init(pthread\_condattr\_t \*cattr);

int pthread\_cond\_init(&condition, &condattr);

pthread\_condattr\_setpshared(&condattr, PTHREAD\_PROCESS\_PRIVATE);

int pthread\_cond\_destroy(pthread\_cond\_t \*cv);

int pthread\_condattr\_destroy(pthread\_condattr\_t \*cattr);

pthread\_cond\_wait(&condition, &mutex)//it can be awakened by below

pthread\_cond\_signal(&condition);

pthread\_cond\_broadcast(&condition);

General way of waiting on a condiion variable

pthread\_mutex\_lock(&mx);

while(condition\_is\_false)

pthread\_cond\_wait(&condition, &mutex);

pthread\_mutex\_unlock(&mx);

Genral way of signalling

pthread\_mutex\_lock(&mx);

pthread\_cond\_signal(&cond);

pthread\_mutex\_unlock(&mx);

You can also block until a specified event occurs. The function pthread\_cond\_timedwait() is used for this purpose. It is prototyped by:

int pthread\_cond\_timedwait(pthread\_cond\_t \*cv,

pthread\_mutex\_t \*mp, const struct timespec \*abstime);

To prevent a deadlock acquire a mutex in same given order like if 1 starts and Ask for mx1 -> mx2 if 2nd process starts it wil try to acuire 2nd which is also claimed by 1st process. That will be a rude idea

**void pthread\_cleanup\_push(void (\****routine***)(void \*),**

**void \****arg***);**

**void pthread\_cleanup\_pop(int** *execute***);**

Clean up handlers are called even if a thread exits. You can specifically call pthread\_cleanup\_pop(int) to pop and execute the current top routine of the stack

pthread\_once\_t once = PTHREAD\_ONCE\_INIT;

pthread\_once(&once, routine);

int a[2][3];

int \*\*p = (int\*\*)malloc(x\*sizeof(int\*));

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

p[i] = (int\*)malloc(sizeof(int));

**Basic Data Types**

switch case works fine. They wont execute any statement in between a case block and any line coming before that, rest it all works fine

&array is an alias for &array[0] and returns the address of the first element in array

o &pointer returns the address of pointer

In case size of array is lesser than the initializer gcc willn't give error instead it will print garbage after the string since it wasn't able to find the last character while in case of g++ error is spewn out

&array+1 points to the end of the whole array while &array[0]+1 points to &array[1]. Both are semantically different even though both of them gives the same address.

But &arrayA has type char (\*)[10] while &arrayB has type char \*\*

In sizeof() operator, operands are not evaluated just their type is identified

static functions can be called from other files with the help of function pointers i.e declare a function pointer and assign it to static function. Now with the help of function pointer static function can be called from other files.

Print if a number is even or odd without condition operator

char arr[2][5] = {"Even", "Odd"};

cout << arr[no%2];

When you apply const to member function that impacts its this pointer

In case of a member function is made const -> const this pointer is passed else simple this is passed since a static function doesn't hold the this pointer, that is not allowed and similar is the case when the function is defined as volatile

static member functions can be called without any object. So vptr (virtual pointer table) can't be made available to them. vptr is set in constructor only.

We can have static variables inside the class that is because it is just declared not defined so it wont add up to the size of the class.