**Unformatted learning notes**

**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 won’t complain but on execution gives SIGSEGV.

> String array is stored on stack

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

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

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 in language standard as to what’s the internal representation of floating point numbers..

> Space matters when doing typedef. Use ‘using’ instead of typedef in C++

using vec\_iter = vector<int>::iterator;

> int is updated to unsigned int when comparison or in expression evaluation.

> static variables 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 with

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

//this works

char \*s = "gaurav";

int \*p = (char\*)(s);

printf("%s", p);

**Volatile variables**

In the context of our discussion about “**volatile**“, C language standard quotes i.e. ISO/IEC 9899 C11 – clause 6.7.3

“An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects.”

“A volatile declaration may be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function. Actions on objects so declared shall not be ‘‘optimized out’’ by an implementation or reordered except as permitted by the rules for evaluating expressions.”

**>** volatile is a type qualifier not a storage class specifier.

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 represented below:

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

Neither ISR return a value nor it takes an argument because it is called in async mode and there is nobody there to handle its return value.

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 mismatch is 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

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

**Functions**

> As per C standard C11, all the arguments of printf() are evaluated irrespective of whether they get printed or not.

> If there are insufficient arguments for the format, the behaviour is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

> In C, if a function name is used without parentheses, the reference to the function name simply generates a pointer to the function, which is then discarded

> "An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 through 7 only."

> As per C, A function cannot have a function or Array as return type.

> Unless and until a function is declared virtual in base class and overridden in derived class, base class pointer will call base class method even when it points to derived class object. That method will be part of base class sub-object inside the derived object.

> Default argument in function can be passed either in declaration or definition but not both.

**Pure Functions:**

In C++, a function a said to be pure which returns the same value when given a particular input. e.g strlen() is a pure function while rand() and time() is not.

**References**

A reference is basically an alias of a variable. Any change made via alias is reflected in the original variable

> References should be initialized in constructor initialization list

> Reference can’t be modified, once initialized

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

> An rvalue reference refers to a temporary object, which the user of the reference can (and

typically will) modify, assuming that the object will never be used again.

> There shall be no references to references, no arrays of references, and no pointers to references.

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

**Classes**

If a derived class doesn't implements pure 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 data members and member functions are not accessible from that static function because ‘this’ pointer is not 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

> 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.

**Interface versus Abstract class**

Abstract class can have definition of a function while an interface cannot have the same

If base class function is overridden in derived class then all other functions are hidden

**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 it’s able to access external and static variables purely because of lifetime

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

**Member Variable of a Class**

While defining members of class outside use address outside and pointer inside

->Mutable keyword to be able to change variable inside lambda, another use is when construct by logic is constant but it some things are added that can be changed like debugging symbols

In caching where you have to retain some variable as mutable though lookup of hash table is const by construct

> non-static data member initializers only available in C++11 ( read it as const )

> static variables don't contribute to size of class. So, for empty class with a static member variable, sizeof(A) and sizeof(A’s object) is 1

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

**Methods inside Classes**

> All the functions defined inside the class are implicitly inline.

> Static members are accessible in non-static functions

> MACROS cannot access private variables of the class

> Virtual function cannot be inlined.

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

> A static member function shall not be declared const, volatile, or const volatile because there is no instance to which const or volatile can be applied to in calling that function.

> There shall not be a static and a non-static member function with the same name and the same parameter types

**Constructors/Destructors**

> Move constructor/assignment operator takes non-const reference

> Argument of a copy constructor can have pointer as an argument but that won’t be good because they can be nullptr while reference cannot.

> We can have main() function overloaded inside a class

> Throwing an exception from constructor is the best way to clean-up 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. This is really dangerous.

class A{

private:

B b;

public:

A(B x){

b = x;

}

};

Steps here:

-call B constructor

-call A default constructor.

-assignment operation.

-Destructor for X

> When we call constructor explicitly then compiler creates a temporary object and deletes that immediately

> Deep copy is required when we are copying one object to another and there is involvement of pointers else shallow copy will be made which will create problem if changes are made in 1st object and such changes are reflected in 2nd object too.

> Destructor is called after return statement while variable is copied so destructor cannot change its value

> All data members are sure to be fully constructed before body of constructor starts.

> Constructor cannot be static member function.

> Deleting a void pointer doesn't calls the destructor

C++11 allows the following:

class SomeType

{

int number;

public:

SomeType(int newNumber) : number(newNumber) {}

SomeType() : SomeType(42) {}

};

**Initializer list:**

Advantages of initializer list:

> References and const can be initialized in this.

> Prevents assignment operator to be called which unnecessary create temporary

> You can call base class constructor with arguments in initializer list of derived class constructor.

> To initialize a value to its default, use default initializer {}

> Don’t use the same name in both a scope and an enclosing scope and prefer the {} - initializer syntax for declarations with a named type;

try catch in initializer 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.

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

> A function-level try/catch automatically re-throw the exception.

> You can refer a member of class inside the constructor because storage for the corresponding object has been allocated, though you might get indeterminate value.

>While defining virtual destructor, you have to do it in base class

> Making a destructor private will result into compiler error if object is allocated on stack.

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

> If move constructor or assignment operator is explicitly declared then no copy constructor and assignment operator are generated

> If constructor fails, throw an exception.

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

A virtual call is a mechanism to get work done given partial information. In particular, "virtual" allows us to call a function knowing only any interfaces and not the exact type of the object. To create an object you need complete information. In particular, you need to know the exact type of what you want to create. Consequently, a "call to a constructor" cannot be virtual.

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

> 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 constructor/destructor are not inherited inside the derived class.

> In case of diamond inheritance, the lowest child calls the topmost class constructor first and then subsequent derived class’ constructors are executed.

> **If the most derived class defines its own function which is defined in class ‘B’ and ‘C’, then only ‘D’’s function is called and the call will be ambiguous when definition of that function is not overridden in the most derived class.**

> **Virtual friend function idiom**: When a virtual function takes a base class reference as one of its parameter, it can act as if it is dynamically bound after it takes derived class objects as its arguments

> The only time we have to call the destructor explicitly is when we have allocated memory using placement new. Placement new can be used when we are allocating memory from an already allocated pool.

> In C++11 you can call another constructor from a given constructor while the same is not possible in C++03

> We should put data members in constructor body if the data members require ‘this’ pointer or we have to assign specific values to array.

> **Named Constructor Idiom**: When you have constructors with same signature but different type of argument then that will result in ambiguity. To resolve that, create static methods.

**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 (except when mentioned with std::nothrow) in which case it throws exception std::bad\_alloc.

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

> delete without '[]' in case of array doesn’t give any error it’s a logical bug BTW.

> delete operator works only for objects allocated using operator new

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 function sea(), in function stack, while returning value and while assigning to x.

But compiler optimize it as

**Foo t;**

**sea(&t, 1, 2)**

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

**ptr = new Foo(i, j);**

**//do with ptr;**

**return;**

**}**

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

**RAII**

Resource Acquisition Is Initialization

“*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.

**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.

> Static variables have internal linkage by default.

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

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

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

> Mutable is also a storage class specifier, which allows const member function and object to modify its value.

> Once the static data member has been defined, it exists even if no objects of its class have been created.

> Static data members are initialized and destroyed exactly like non-local objects

// in namespace or global scope

int i; // extern by default

const int ci; //static by default in C++ but not in C in which you need ‘static’ keyword

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 relaxed 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.

> typedef declares a type alias. It is a storage class purely as a syntactic sugar to prevent following declaration:

typedef static int int32; because static is a qualifier which 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. such as:

So, typedef is a syntactic sugar introduced just to prevent static aliasing

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 to same port, compiler might optimize it and do b = a;

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

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

They are initialized just before the execution of program

**Operators**

int a[] = { 10, 20 };

The expression ++\*p has two operators of same precedence, so compiler looks for associativity. Associativity of ++ (prefix) and \* (indirection) 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“.

Some operators return by value, some by reference. In general, an operator whose result is a new value (such as +, - (binary) 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

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

int Add(int x, int y)

{

    if (y == 0)

        return x;

    else

        return Add( x ^ y, (x & y) << 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 ,

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

> sizeof is an operator and can be used without parentheses.

> sizeof operator is evaluated at compile time

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

> The sizeof operator doesn't evaluates the expression, it just return size of operand

**Casting**

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

> Static Cast won’t allow casting between unrelated data type...like pointer to integer or integer to pointer. But it’s allowed in C-casting. e.g. –

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;

> 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, you will get NULL pointer.

> dynamic\_cast uses typeinfo from vtable to determine the validity of dynamic\_cast. typeinfo is placed just before the vtable.

> 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 mechanism.

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.

> Mutex implementation underneath is provided by hardware atomic instructions compare-and-swap idiom. The thread first checks the global mutex value ( a variable created for a given mutex ) if that is ‘0’ it makes it ‘1’ and thread can proceed else thread will be blocked, put in scheduling queue and wait for mutex flag to be turned ‘0’ again.

> Cond-Wait mechanism works this way only. threads are made to wait in a kernel scheduling queue for a given mutex (which is assigned a value)

**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 an entity that prevents memory leaks and actual work is done by mutex.

lock\_guard<mutex> m;

> Difference between lock\_guard and unique\_lock is that lock\_guard doesn’t have function named lock while unique\_lock provides it. In a function, when you have to repeatedly lock and unlock a mutex, use unique\_lock.

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

B b;

A &a = b;

**POSIX Semaphores:**

A semaphore is an object with an integer value that we can manipulate

with two routines; in the POSIX standard, these routines are sem wait()

and sem post()

sem\_t m;

sem\_init(&m, 0, X); //initialize the semaphore to X (resource count)

int sem\_wait(sem\_t \*s) {

// decrement the value of semaphore s by one

// wait if value of semaphore s is negative

}

int sem\_post(sem\_t \*s) {

// increment the value of semaphore s by one

// if there are one or more threads waiting, wake one

}

The thread/process will wait till value of semaphore is > 0

**Polymorphism (including Virtual Functions)**

**Definition:**

When derived class overrides the base class method by redefining the same function, then if client wants to access redefined method of derived class through a pointer from base class object, then you must define this function in base class as virtual function.

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**

> Just declaring (and not defining it) virtual function will give ‘typeinfo’ error.

> Upcasting slices the object because u won’t be able to call derived class functions when used with base class pointer.

> Virtual function when called from base class constructors only point to the base class, because till then derived class constructor 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 destructor options.

> If you want to make base class abstract but there is no pure virtual function you can make the destructor pure and virtual that’s 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 using base class pointers in case of runtime dispatch only those member functions are accessible which are defined in base class.

**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.**

class A { virtual void display(){} };

class B : public A { void display(){} };

A \*ab = new B;

A \*aa = new A;

B \*b1 = dynamic\_cast<B\*>(aa); // returns NULL

B \*b2 = static\_cast<B\*>(aa); // Successful but undefined

B \*b1 = dynamic\_cast<B\*>(ab); // Works fine

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

> Virtual pointer is inherited by derived classes.

> Within C++, a polymorphic class is one that contains either an inherited or declared virtual function.

> Virtual table is created in each class which has either a virtual function or inherits from a class that has a virtual function. The base class pointer points to a derived class object when runtime polymorphism is realized. Hence, vtable of derived class is referred.

> The derived class inherits vtable of base class as well (basically implementation defined)

> A call to virtual function from base class constructor while that constructor is invoked via polymorphism will call base class virtual function because derived won't be created by that time.

> \*vptr is associated with each object

> this pointer is implicit to the object and can check the behaviour just before constructor

> Problem with Double check locking is Sequence re-ordering, normal sequence will be

-Allocated memory.

-Create object in it

-Make pointer point it to that memory

But if compiler decides to flip statements 2 and 3 and one thread decides to stop after 3(Make pointer 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

**VTABLE**

VTable or virtual table is a table containing function pointer entries of virtual functions declared in a given class.

Virtual tables for all base classes are inherited and entries of overridden functions are replaced by the derived functions.

class B1 {

public:

void f0() {}

virtual void f1() {}

int int\_in\_b1;

};

class B2 {

public:

virtual void f2() {}

int int\_in\_b2;

};

class D : public B1, public B2 {

public:

void d() {}

void f2() {} // override B2::f2()

int int\_in\_d;

};

and the following piece of C++ code:

B2 \*b2 = new B2();

D \*d = new D();

g++ 3.4.6 from GCC produces the following 32-bit memory layout for the object b2:[nb 1]

b2:

+0: pointer to virtual method table of B2

+4: value of int\_in\_b2

virtual method table of B2:

+0: B2::f2()

and the following memory layout for the object d:

d:

+0: pointer to virtual method table of D (for B1)

+4: value of int\_in\_b1

+8: pointer to virtual method table of D (for B2)

+12: value of int\_in\_b2

+16: value of int\_in\_d

Total size: 20 Bytes.

virtual method table of D (for B1):

+0: B1::f1() // B1::f1() is not overridden

virtual method table of D (for B2):

+0: D::f2() // B2::f2() is overridden by D::f2()

D \*d = new D();

B1 \*b1 = d;

B2 \*b2 = d;

While d and b1 will point to the same memory location after execution of this code, b2 will point to the location d+8 (eight bytes beyond the memory location of d). Thus, b2 points to the region within d which "looks like" an instance of B2, i.e., has the same memory layout as an instance of B2.

(\*(\*(d[+0]*/\*pointer to virtual method table of D (for B1)\*/*)[0]))(d) */\* Call d->f1() \*/*

(\*(\*(d[+8]*/\*pointer to virtual method table of D (for B2)\*/*)[0]))(d+8) */\* Call d->f2() \*/*

Virtual table is not created unless and until virtual function is defined (not declared)

In Diamond inheritance problem, the problem subsides only after both 2nd level derived class inherits the base virtually.

In case of multiple inheritance, The leaf node class contains virtual pointer of both base classes.

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

Functions

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

2 functions - 1 contains a variable as argument and another containing the same const argument are not considered as overloaded because only copy is passed while in case of pointer they are different i.e. char\* and const char\* are different hence overloaded function as arguments of such type is allowed

**Exception Handling**

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. If you catch by value and in case derived class exception is thrown and handled by base class, then thrown object will be sliced. To keep it correct, catch by reference of base class type.

Static order initialization fiasco depends on the dependency among 2 static variables 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.

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

> std::range\_error, std::overflow\_error etc. You can define your own exception classes descending from std::runtime\_error, as well as you can define your own exception classes descending from std::exception.

> runtime\_error has constructor while exception has no.

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

**C Internals:**

>A signed char is guaranteed to be signed and an unsigned char is guaranteed to be unsigned but a normal ‘char’ definition is implementation defined.

> ‘void’ can only be used to specify that a function returns nothing and that a type of pointer pointing to some unspecified type.

1 = sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)

It is extremely unwise to assume that the size of an int is the same as the size of a pointer; many machines (‘‘64-bit architectures’’) have pointers that are larger than integers.

> .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.

> Dangling pointers are those which point to invalid memory locations. Example: Address of local variable of function whose frame has been removed.

> 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 to store variable from register to local memory while the reverse is called filling.

> Difference between typedef and define:

typedef int\* int\_p1;

int\_p1 a, b, c; // a, b, and c are all int pointers.

#define int\_p2 int\*

int\_p2 a, b, c; // only the first is a pointer!

Compiler created temporary objects cannot be bound to non-const references

To find offset of struct use offsetof(struct, variable); from <stddef.h>

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 initialized ones are while in case of c++ error is generated ;)

> 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 an 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();

> In case of ref. var. you can’t 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 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; }

// ...

};

When you apply the const qualifier to a non-static member function, it affects ‘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\*

**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 )

**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.**

**Operator Overloading**

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

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 constructor and destructor.

In C++, following are the general rules for operator overloading.

1) Only built-in operators can be overloaded. New operators cannot be created.

2) Arity of the operators cannot be changed.

3) Precedence and associativity of the operators cannot be changed.

4) Overloaded operators cannot have default arguments except the function call operator () which can have default arguments.

5) Operators cannot be overloaded for built in types only. At least one operand must be used defined type.

6) Assignment (=), subscript ([]), function call (“()”), and member selection (->) operators must be defined as member functions

7) Except the operators specified in point 6, all other operators can be either member functions or a non-member functions.

8) Some operators like (assignment) =, (address) & and comma (,) are by default overloaded.

9) Operators like scope resolution ( :: ), member access ( . ), pointer to member (.\*), Ternary (?:) cannot be overloaded.

**Reason**: Scope resolution and member access work on names rather than values. sizeof needs to find the size at compiler time.

> We can overload [][] either by using another within a class or use (x,y) type of constructor.

Example of iterator overloading

class LinkedListIterator {

public:

bool operator== (LinkedListIterator i) const;

bool operator!= (LinkedListIterator i) const;

void operator++ (); // Go to the next element

int& operator\* (); // Access the current element

private:

LinkedListIterator(Node\* p);

Node\* p\_;

friend class LinkedList; // so LinkedList can construct a LinkedListIterator

};

class LinkedList {

public:

void append(int elem); // Adds elem after the end

void prepend(int elem); // Adds elem before the beginning

...

LinkedListIterator begin();

LinkedListIterator end();

...

private:

Node\* first\_;

};

To overload new operator we have to pass parameter of type size\_t while in case of delete we have to pass void\*

void\* operator new(size\_t);

void operator free(void\* );

Example: Fred\* p = (Fred\*) operator new(sizeof(Fred));

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 constructor by having a Create method as static

vptr is related to object while vtable is related to class

> Returning reference from operator overloading allows chaining and returning value rather than reference 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 principal 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

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

**Access specifiers are checked at compile time**

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

Implicit type conversion doesn’t happen for primitive types

The catch(...) must be the last catch block else error will be thrown.

Constructors are not inherited, however assignment operator is.

**RTTI**

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

RTTI or run time type information is a scheme to determine type of an object at run time

following operators are used to implement the same

dynamic\_cast operator is used to check if the Downcasting is possible or not

type\_info class and typeid operator and it returns type\_info class

typeid(int) == typeid(int&); // evaluates to true

Virtual function cannot be inlined

**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 non-type 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.

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 non-contiguous

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

**Little and Big Endian Mystery:**

In big endian the MSB stored first

in little endian MSB (byte) stored last

**To retain - L^3 – LSB stored in lower address for Little Endian**

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 byte 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

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

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) if realloc fails then old memory is kept sane.

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.

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.

To find set bits in an integer:

while(n)

{

count += n & 1;

n >>= 1;

}

and

while (n)

{

n &= (n-1) ;

count++;

}

return count;

Count number of bits to be flipped to convert A to B

Do XOR of 2 numbers and count the set bits in the resultant number.

**How to write your own sizeof operator?**

#define my\_sizeof(type) (char \*)(&type+1)-(char\*)(&type)

 In C, if a macro is not defined, the pre-processor assigns 0 to it by default.

To count the set bits in floating point. typecast the number to char pointer and count the set 1-by-1

Random Notes :

In C++11 it is possible to have in-class member initializer just like int a= 9; and this is better in case we have multiple constructors.

In strcpy the source should be const char\*

C++ allow uniform initializaion for all types of data types --> { }

static\_assert(Expression, String); //if expression failed throw error in the form of string

long long type is defined in C++11

int a = nullptr ; //error nullptr is not integer

C allows partial initializers in array like : int arr[50] = {0,1,2,[47]=47,48,49};

Pointers to functions (§12.5) and pointers to members (§20.6) cannot be assigned to void∗s.

In definition of these arrays, the mention of array size using variable is ok as per C standard but these types of arrays can’t be initialized at the time of definition.

An array whose size is specified as variable can’t be defined out any function.

pre-increment operator can work as l-value but post-increment cannot, because pre-increment returns reference to incremented variable while post-increment returns temporary copy.

Nested class have access to private members of nesting class, but cannot modify it.

Enclosing class cannot access private members of nested class.

% operator cannot be used with floating point numbers in C & C++.

Check range of x between low and high - return ((x-low) <= (high-low));

In C, it is possible to have array of all types except following: void, functions.

cin/cout are a bit slower than printf/scanf because of the synchronization they have to do with stdio functions. It can be prevented by using following function call:

std::ios::sync\_with\_stdio(false);

In unordered\_multiset, equal\_range(val) function returns a pair of type where first iterator points to first position of val and second points to last position of val in data structure.

We can use fill() and fill\_n() function to assign a value to certain range in a vector.

deque allows insertion at front while vector doesn’t

If pid > 0, then it is parent process, else if pid == 0, it is child process, else it is an error.

vector of bool and bitset both pack bool elements.

bitset<T> need exact size while initialization, else it’ll give error.

Within C++, a polymorphic class is one that contains either an inherited or declared virtual function.

strtok() is not re-entrant while strtok\_r() is, because strtok() maintains its state in global variable. hence, it cannot be called by same thread at 2 places.

To overload Iterator in C++, encapsulate the pointer to linked list inside the class iterator.

Extra brackets with function names in C/C++. if we have a macro with same name as function, then extra brackets avoid macro expansion wherever we want the function to be called.

endl flushes the stream... \n doesn't

printing a NULL pointer is undefined behaviour in C, though in gcc it prints (null)

int sum(int a, int b)

{

    char \*p = (char\*)a;

    return (int)&p[b];

}

We can pass \_\_FILE\_\_ in fopen so that a program can print its text.

GCC specific syntaxes :

1. \_\_attribute\_\_((constructor)) syntax : This particular GCC syntax, when used with a function, executes the same function at the startup of the program, i.e before main() function.

2. \_\_attribute\_\_((destructor)) syntax : This particular GCC syntax, when used with a function, executes the same function just before the program terminates through \_exit, i.e after main() function.

void \_\_attribute\_\_((constructor)) calledFirst();

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

**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);

General 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 will try to acquire 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 won’t execute any statement in between a case block and any line coming before that, rest it all works fine

In case size of array is lesser than the initializer gcc won’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 \*\*

**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];