**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 (how numbers will be stored). In case of floats/double value-representation is undefined in language standard as to what should be 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;

> There is no difference between ‘using’ and ‘typedef’

> 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 each instance of class will define a static variable.

Take care of expression such as :

**for** (unsigned int 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

**Type Qualifiers:**

**Volatile**

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.

> Type qualifiers modify the property of variable while storage class specifier dictates where to store the variable, what’ll be its initial value and what’ll be the lifetime.

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

> Example showing compiler behavior for volatile variables:

const int local = 10;

int \*ptr = (int\*) &local;

printf("Initial value of local : %d \n", local);

\*ptr = 100;

printf("Modified value of local: %d \n", local);

With optimizations turned off, the code will print 10, 100 but when optimizations are turned on, the code will print 10, 10. Hence, when compiler tries to optimize it will store the value of const in some fixed location which is used for every access. change the variable declaration to:

const volatile int local=10 and then we will have 10, 100 as the resultant values because value wasn't optimized.

> In C, return type of getchar(), fgetc() and getc() is int (not char). So it is recommended to assign the returned values of these functions to an integer type variable.

**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.

> "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.

> By default, return type of function is 'int' unless 'void' is explicitly specified.

> If the return type of 'main()' function is not 'int' or is 'void', C will give only warning, but C++ will throw error.

> '++i', 'i' and 'i++' inside 'printf()' - Go from right to left, wherever i++ is encountered the value is what it is but in case of ++i and i, the value will be whatever was at the end of parsing the printf arguments.

> We can have function overloading even if input parameter is of same type but one should be normal type and another one is of type reference to same type

> When a function is declared as const, it can be called on any type of object, const object as well as non-const objects.

> Whenever an object is declared as const, it needs to be initialized at the time of declaration. however, the object initialization while declaring is possible only with the help of constructors.

**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.

**Functors**

Functor is a class with overloaded () function. Advantages of functors are:

> It can maintain state. (create multiple function objects and each has its own state)

> Better equipped for templates.

> It is inlined easily.

> ISO C++ forbids declaration of `operator()' with no type (it should’ve a return type)

> Also, if we have to perform a given piece of code by iterating over some data, then functions like for\_each(), transform() etc. separates iteration from calculation.

Types:

**Generator**: A function object which is called with no argument.

**Unary Function**: A unary function object is called with one argument.

**Binary function**: A binary function is called with two arguments.

The line,

transform(arr, arr+n, res, increment(to\_add));

is the same as writing below two lines,

// Creating object of increment

increment obj(to\_add);

// Calling () on object

transform(arr, arr+n, arr, obj);

A function object which is called with one argument is called predicate. In Unary function, it is simply called as predicate while in Binary function, it is called as Binary predicate.

**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. Because references are not objects. They don't have storage of their own, they just reference existing objects. For this reason it doesn't make sense to have arrays of references.

> We cannot have reference to pointer

> The compiler created temporary objects cannot be bound to non-const references. So, returning an auto object from function and assigning to another

will raise error

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

**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.

> A class declaration can contain static object or pointer 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.

**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

Example:

int A::\*ptr = &A::int\_var;

->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 or requirement of logger inside const function.

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

> You can use non-const variable in const function provided that you are not modifying the variable, just accessing the data.

> Non-static data member initializers only available in C++11 (even without const)  
> Initialization of static data member without const is error in all versions of C++

> ISO C++ forbids in-class initialization of non-const static member.

> 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

> A static variable **DEFINED** in class should only be of const integral type.

> Linker won’t give error for static variable not being defined in case static variable isn’t used.

> Non-scoped enum or enum without class, if used in a class, should be defined and forward declaration is not possible in such case while the same is not true for ‘enum class’

> The initialization order of member variables is determined by their order of declaration, not their order in the initialization list.

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

**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 because compiler doesn’t know function of which class is going to ve invoked.

> 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

> A virtual function declared in a class shall be defined, or declared pure in that class, or both. A declared, but undefined virtual function will generate compiler error because when compiler creates a virtual table it needs entries of virtual function.

> 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.

> Constructors and assignment operators are never inherited.

> Since C++ supports implicit conversion, call to a function whose type differs only by the return type is ambiguous. float x = fn(int/double) is ambiguous.

> Self-assignment check is required for conditions in which we have: a = a; and we have delete[] statement in the beginning to wipe out the contents.

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

**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.

> using “{}” in constructor initialization list prohibits narrowing while “()” allows the same.

> 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. That is really dangerous.

> 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 ‘void\*’ is undefined. Hence, deleting a void pointer mightn't calls the destructor

> Delegating constructors are only allowed in C++11 as:

class SomeType

{

int number;

public:

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

SomeType() : SomeType(42) {}

};

> 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 you want to call a superclass constructor with an argument, you must use the subclass's constructor

> 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.

> 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.

> The thing is that it's known at compile time how the function will be called: via virtual table or just will be a usual call.

**Initializer list is required**;

> For initialization of non-static const data members:

> For initialization of reference members:

> For initialization of member objects which do not have default constructor

> For initialization of base class members: the parameterized constructor of the base class can only be called using Initializer List.

> When constructor’s parameter name is same as data member

> **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. For this, we need to call destructor explicitly.

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

> If you do not mention a variable in a class's initialization list, the constructor will default initialize it before entering the body of the constructor you've written. This means that option 2 will lead to each variable being written to twice, once for the default initialization and once for the assignment in the constructor body. Hence, initialization list in constructor is efficient.

**Layout in a class**

In C++, this “sharing inheritance” is (unfortunately) called virtual inheritance and is

indicated by specifying that a base class is virtual.

**C++ under the hood:**

In the implementation of Visual C++ compiler and most other compiler, derived class members are simply appended to the base class members.

struct P {

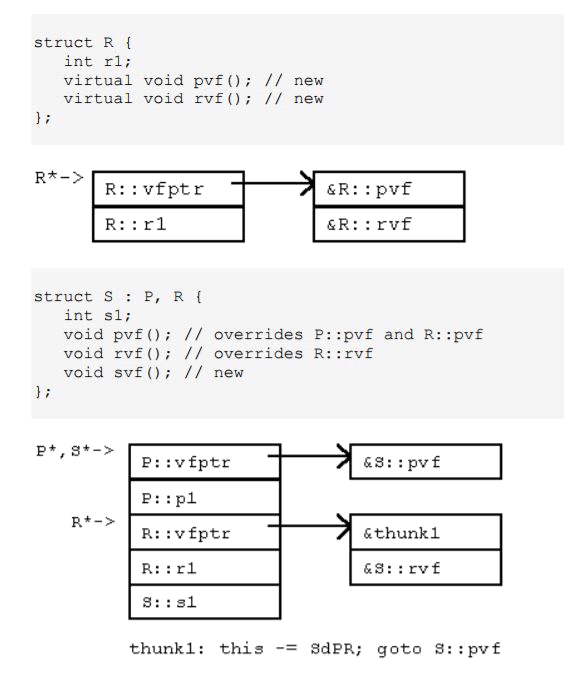
int p1;

void pf(); // new

virtual void pvf(); // new

};

Below is the layout of class in case of multiple inheritance:



Casting derived class pointer to base class will make the casted base class pointer to point to base class instance inside that derived class.

The expression (R\*)ps does not point to the same part of the class as does (P\*)ps. Since the function S::pvf() expects to receive an S\* as its hidden this parameter, the virtual function call itself must automatically convert the R\* at the call site into an S\* at the callee. Therefore, S’s copy of R’s vftable’s pvf slot takes the address of an adjuster thunk, which applies the address adjustment necessary to convert an R\* pointer into an S\* as desired.

Casting one pointer to another in inheritance hierarchy takes the pointer-to-be-cast to the corresponding location.

**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) ) only 1 ctor call

> On a standards-conforming **C++** implementation, no. The ordinary form of **new will** never **return NULL** ; if allocation fails, a std::bad\_alloc exception **will** be thrown (the **new** (nothrow) form **does** not throw exceptions, and **will return NULL** if allocation fails).

> 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;**

**}**

**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. Further helpful use of unnamed namespaces is that we can encapsulate class as well.

> 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.

> Mutable is also a storage class specifier (because it restricts the variable to be put in **read-only memory**), 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

> 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++

> C allows a global variable to be declared again when first declaration doesn’t initialize the variable but C++ doesn't allow the definition even if 1st just declares the variable.

> In C, static and global variables are initialized by the compiler itself. Therefore, they must be initialized with a constant value, but variable initialization is allowed in C++.

> C allows a global variable to be declared again when first declaration doesn’t initialize the variable.

> 'char' or 'short' types are promoted to 'int' before any arithmetic operation is performed on them. In case they are prefixed unsigned, they'll be promoted to 'unsigned int'

> sizeof 'long' maybe '4' or '8' depending on the architecture while size of 'long long' is '8' bytes

> To check the platform, use sizeof(size\_t); rathern 'int' or 'long'

> We can’t use any modifiers in float data type. If programmer tries to use it ,the compiler automatically gives compile time error.

> Only long modifier is allowed in double data types. We cant use any other specifier with double data type. If we try any other specifier, compiler will give compile time error.

> 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.

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. Such variables cannot be global because global variables exist all of the time and it’s not possible to permanently reserve space for a register global variable.

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

They are initialized just before the execution of program.

**Const expression is required because compiler doesn’t guarantees sequence of initialization.**

e.g. int z = (sizeof(int)\*2)

int x = z/3;

Compiler may consider ‘z’ as 0 and initialize ‘x’ first

**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.

You should return by reference if you are using += and return by value if it is +, - etc. But if you want to support chaining, then also you should return by reference.

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

> Test \*ptr = new Test; will first call new operator and then the constructor

**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.cpp:8:10: warning: initialization from incompatible pointer type

int \*p = (int\*)&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

**Use of const cast:**

1. Changing the member variable of that class by doing away its const qualifier

2. const\_cast can be used to pass const data to a function that doesn’t receive const.

casting won’t happen if the type of cast is not same as original object. like char\* to int\*

**Polymorphism (including Virtual Functions)**

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 you ensures that it is safe, because there may be another directly derived classes that base pointer might be pointing to

> *dynamic\_cast* cannot be used when classes/objects are not polymorphic type, use *static\_cast*

> **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.

> <typeinfo> is used for RT polymorphism. It uses typeid() function which can be used to compare the related type.

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

> 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.

> 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.

> 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**.** For Example:

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.

> 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

> When we use ‘virtual’ keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

**VTABLE**

VTable or virtual table is a table containing addresses 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.

If a class inherits from 2 base classes, then vptr of this class and vptr of 1st base class is same and shared.

Suppose class Base has 5 virtual functions: virt0() through virt4()

class Base {

public:

virtual int virt0();

virtual int virt1();

virtual int virt2();

virtual int virt3();

virtual int virt4();

};

Following is virtual table generated at compile time:

FunctionPtr Base::\_\_vtable[5] = {

&Base::virt0, &Base::virt1, &Base::virt2, &Base::virt3, &Base::virt4

};

A hidden pointer is added to each class containing virtual functions and initialized in constructor

class Base {

public:

FunctionPtr\* \_\_vptr;

Base():\_\_vptr(&Base::\_\_vtable[0])

}

The functions overridden in derived class replace the ones in base classes

// virt3 and virt4 are inherited as-is

FunctionPtr Der::\_\_vtable[5] = {

&Der::virt0, &Der::virt1, &Der::virt2, &Base::virt3, &Base::virt4

???? ????

};

Base\* p;

The following call:

p->virt3(); // is translated to below call

p->\_\_vptr[3](p);

It's basically 2 load's call. One loads v-pointer and second gets the word at r1+3\*4(Considering word size as 4).

Notes:

> 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, with the vptr of 1st base and derived class shared.

**Exception Handling**

The recommended way is to throw an exception is by value and catch it by reference, because if you throw a pointer, you need to deal with memory management issues. Also, in case derived class object is thrown as an exception object which is handled by base class, then thrown object will be sliced. To keep it correct, catch by reference of base class type.

> 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:**

C/C++ standard guarantees the following:

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

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

> 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”

> 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 in case of gcc and **error** in case of g++and printed characters are which initialized ones are while in case of C++ error is generated ;)

**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();

where fun() is

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

return std::make\_tuple(1, 2);

}

> Adding extra element in tie will result in compilation error.

> 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: Base &b = d; //where its Derived d

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

> ‘*non const*’ and ‘*const*’ member functions are different in C++. Hence, overloading is possible.

> C++ allows member methods to be overloaded on the basis of *‘const’* and *‘volatile’* type only if the const/volatile parameter is a reference or a pointer, otherwise error is thrown.

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; }

// ...

};

> 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 of using operator new:

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

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

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

Standard implementation of Pre and Postfix operator:

T& T::operator++() // pre-increment, return \*this by reference

return \*this;

T T::operator++(int) // post-increment, return unmodified copy by value

T copy(\*this);

++(\*this); // or operator++();

return copy;

Difference between const and constexpr

int age;

cin >> age;

const int myAge{age}; // works

constexpr int someAge{age}; // error: age can only be resolved at runtime

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

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

> We can access private members in a friend function.

> 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

**Templates**

We can have Type Parameters.

\* Templates (only classes and alias templates, no functions or variable templates)

\* Non-type Parameters

We can have any of the below non-type Parameters to the template:

\* Pointers

\* References

\* Integral constant expressions

class templates:

It is possible to have default parameter types in the templates such as

template<class T, typename U=char>

and then make a declaration like

Array<int>

> Remove class scoping from friend function declaration

> You cannot have default argument for friend template

A class name cannot be overloaded i.e two classes, one template and another non-template one cannot be the present in the same file.

> A virtual function cannot be template because template functions are expanded at compile time while virtual concept is all about runtime.

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

> When a template derived class inherits from base template class, then members of base class, if used in derived class, will be unknown to derived class and needs to be explicitly mentioned about their use using <base\_class\_name>:: or else global version will be called.

> Compiler can evaluate constant expression inside templates - buf<char,2-1> to buf<char, 1>

> Templates are expanded at compile-time.

SFINAE (Substitution failure is not an error) and enable\_if

In case of function overloading, compiler will choose the option in which no conversion needs to be done. So, even int will be passed on to const int.

void foo(unsigned i) {

std::cout << "unsigned " << i << "\n";

}

template <typename T>

void foo(const T& t) {

std::cout << "template " << t << "\n";

}

In the above code, calling foo(90); will call the template version.

Look at the below code, when negate(89) is called, the template will pick the non-templated negate and second function in which substitution might fail is ignored because of SFINAE concept.

int negate(int i) {

return -i;

}

template <typename T>

typename T::value\_type negate(const T& t) {

return -t();

}

We can specify a template which takes non-type as parameter as:

template<int N>

class P{};

template<>

class P<90> {};

Example of template->template parameter:

template< template<typename T> class AllocatePolicy>

struct Pool {

void allocate(size\_t n) {

int \*p = AllocatePolicy<int>::allocate(n);

}

};

// pass the template "allocator" as argument.

template<typename T>

struct allocator { static T \* allocate(size\_t n) { return 0; } };

Pool<allocator> test;

> Templates definitions and instantiations should go hand in hand else linker will complain, reason being when both are in 2 different translation units, compiler will not generate any class when it parses the first t. unit which contains the template definition. But when instantiation is encountered, it checks for constructor which won't be there in any translation unit.

> In case of friend function, compiler will think that these functions are non-template ones and when call is made linking errors are flashed because call was made to template function. To resolve this crap, declare the friend function before class declaration as:

template<typename T> std::ostream& operator<< (std::ostream& o, const Foo<T>& x);

and, in class declaration add <> after friend fuction name

friend std::ostream& operator<< <> (std::ostream& o, const Foo<T>& x);

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 destructor 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 to take address of function to assign it to function pointer

Array of function pointers is possible

In structure, a bit field variable cannot be static

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 )

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\*

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.

> Associativity of postfix ++ is left to right and associativity of prefix ++ is right to left.

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

> static int i = initializer() { return 50; } will fail as 'i' expects a constant literal.

> iscntrl() and isprint() can be used to check if a given character is control character or printable character.

> Variables with register storage class are not allowed to be declared globally, because it's unreasonable to have a variable ocupy register all the time.

> Difference between %i and %d is only in case of scanf in that %i automatically detects the base of input (whether it is hex, oct or dec) while for %d specifier expects integer only. So, 012 would be 10 with %i but 12 with %d.

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

> 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.

> 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.

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

> Also, a function in a base class in diamond inheritance creates problem without virtual public inheritance.

> When we use ‘virtual’ keyword(Virtual base class), the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

> 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];

}

> endl vs '\n': The latter one is just a newline character while former one is a manipulator that flushes the stream after inserting the newline character.

Also, cerr is unbuffered

cout<<endl is equivalent to cout<<'\n'<<std::flush;

**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

**> 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];

> All members of 'struct' are laid out in continuous memory segment. So, it's not feasible to have a static variable since that would amount to one time being not stored along with other members in case when struct is declared and defined in stack.

Undefined/Unspecified/Implementation defined behaviour

**Undefined behaviour**: Anything can happen in all implementations. Example

The effect of attempting to modify a string literal is undefined.

Other examples of undefined behaviour include accessing an array beyond its bounds

**Implementation defined** – Each implementation documents how the choice is made

Example: Size of integer

**Unspecified behavior**: Not documented by implementation.

An example of unspecified behaviour is the order in which the arguments to a function are evaluated.

**Advanced C++ Notes (Conforming to C++11)**

> An implicitly-defined copy constructor would call the copy constructor of its bases while for user-defined ctors we have to explicitly call them else default ctors will be invoked

> Accessing volatile variables on same sequence point results in undefined behaviour

> Do not try to use members of class as Class:: when class is only forwardly declared

> Access is checked at the call point using the type of the expression used to denote the object for which the member function is called". The call point here being a.f(), and the type of the expression is A&. So, even if f() is private in B, it will be called because check is done for A.f()

> Since the lambda is not declared mutable, the overloaded operator() of the closure type will be a const member function

Initialization\_list constructor is greedy, so it consumes : A a{1} even though A(int) is defined.

> extern "C" int x; is a declaration

> extern "C" { int y; } is a definition

> §6.6¶2 Transfer [...] back past an initialized variable with automatic storage duration involves the destruction of variables with automatic storage duration that are in scope at the point transferred from but not at the point transferred to.

label:

A a;

while(1)

goto label;

It will destruct 'a' again and again

> int foobar::x = foo();

For this call, foo() will be searched in class foobar first then outside the namespace

> The reference bound to a temporary extends the lifetime of temporary till the end of scope of reference.

> The point of declaration for a name is immediately after its complete declarator (clause 8) and before its initializer(if any)

Hence, int x = x; is a valid statement.

> Member variables are initialized before the constructor is called. The destructor is called before member variables are destroyed.

> "Default arguments are evaluated each time the function is called."

so, In f(int a = fn()) fn will be called each time f() is called

> When constructor fails/throws exception, then object is not created and no destructor is called.

"If an expression initially has the type “reference to T”, the type is adjusted to T prior to any further analysis." <--

> A virtual function call (10.3) which uses the default arguments gets the arguments value resolved by the static type of the pointer or reference denoting the object. An overriding function in a derived class does not acquire default arguments from the function it overrides.