**C/C++**

**Strings**

> String literals / const data such as { char \*p = “gaurav”; } is stored in read only data segment in C, therefore returning them from function is fine and doesn't get destroyed. sizeof gives size of pointer and strlen gives exact length of string.

> While modifying the above string literal, compiler won’t complain but on execution gives SIGSEGV, but if you declare it const char \*p, then you’ll get compilation error.

> String array – char [] s = “poled”; - is stored on stack.

> Using string to initialize a character array (mentioned in previous statement) will automatically add extra '\0' to it. Hence, sizeof(str) = strlen(str) + 1 (Verified)

> Calculating size of non-null terminated string ( char a[] = { ‘a’, ‘a’, ‘a’ }; ) is UB (prints 3 in gcc though)

> Declarations:

>> char s2[4] = { 'a','a','a', 0 }; // good if string MUST be 3 chars long

>> char \*s2 = "aaa"; // if you don't need to modify the string after creation

>> char s2[]="aaa"; // if you DO need to modify the string afterwards.. sizeof = 4 and strlen = 3

> You will get compilation error for 🡪 char x[3] = "abc";

**Pointers**

> void pointer couldn't be dereferenced. Cast it appropriately to use it like for integer where v is void pointer \*(int\*)(v). Otherwise \*v will give error (verified)

> Basically, void is an incomplete type and sizeof(void) and void pointer +/- 1 shouldn't make sense but for sizeof(void), gcc returns 1 and also allows increment operations on void pointer.

on void pointer which is not allowed by most compilers. Also, sizeof(void\*) is 4/8 bytes depending on machine, but, standard-wise, it is undefined.

> void type is allowed to declare a variable. You’ll get compilation error (verified)

> Compiler prevents converting volatile or const int\* to int\* but vice versa is allowed. You’ll get compilation error even if you use static cast.

**Enums**

> Members of non-class enums can be accessed without adding operand along with scope-resolution operator.

> We can create objects of enum as well

enum Gohana { FIRST, SECOND };

Gohana g = Gohana::SECOND; g = SECOND will work too.

Why enum class had to be introduced?

> Two enumerations cannot share the same names (you’ll get compilation error)

> Enums are not type-safe: These can be converted implicitly to int. But, for enum class, you have to use ‘static\_cast’

> 'enum class' is also called scoped-enumeration

enum class has more control over underlying type; it may be any integral data type, such as char, short or unsigned int, which essentially serves to determines the size of the type.

This is specified by a colon and underlying type following the enumerated type:

eg: enum class eyecolor : char { char,green,blue };

**Structure:**

> You can initialize members directly in structure just like class in C++. (verified with string, int etc..)

> In structure, a bit field variable cannot be static and we can’t take address of it as these might not start at byte boundary.

> A special unnamed bit field of size 0 is used to force alignment on next boundary such as :

int a:0

> It is implementation defined to assign an out-of-range value to a bit field member.

> Need extra operation of bit masking and all, mightn’t be efficient. Bit field storage is implement defined so we don’t know in what way it might be stored. Little-endian or big-endian.

> When we send metadata. Like information about a header, we can use very less information to communicate this thing.

> Used in old games and in present in boost mutex struct for atomicity.

> But they degrade information since they have to be converted to word for register to work on them

> struct and class **can be** mixed. A class can inherit from struct and similarly, a struct can inherit from class.

**Union:**

It is a user defined data type which can contain multiple members, but at a time, only 1 member can contain a value. Unions usually make sense inside a struct that has a flag that indicates which is the "active" member. Size of a union is dependent on size of largest element as well as alignment.

struct ONE\_OF\_MANY {

enum FLAG { FLAG\_SHORT, FLAG\_INT, FLAG\_LONG\_LONG } flag;

union { short x; int y; long long z; };

};

Need:

> Memory saver. That’s why mostly used in embedded systems

**Type Qualifiers and Storage classes:**

> Type qualifiers modify the property of variable while storage class specifier determines where to store the variable, what’ll be its initial value and what’ll be the lifetime. We can’t specify 2 storage class specifier for a given variable.

Storage classes: auto, extern, register, static, mutable

Type Qualifier: const, volatile

**Static**

> The global/static variables which are either uninitialized or '0' initialized are stored in .bss segment and it is done so that

>> There is some dependable starting state. No if checks required in 1st use.

>> They are initialized at compile time since their relative address is known and fixed so no overhead.

>> There is no security related issue

The auto variables are uninitialized because just old memory area of some stack is utilized which consists of random values. Apart from that, for efficiency purpose auto vars are not initialized. There might be array with 100 values unused.

> static variables have to be defined outside the class just because of one definition rule (ODR). If that was allowed every place each instance of class will define a static variable.

> const data (char \*c and const int s[] = { 1, 2.. } etc. is stored in .rodata segment.

> 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. It means access is restricted to given file.

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

> Mutable is also a storage class specifier (because const puts the variable in write-protected **memory area** so we can’t make it type-specifier0. It can be used with functions which are itself declared const. It can be used for synchronization primitive, logger object etc. so that mutex can be used in const function

> Once the static data member has been defined, it exists even if no objects of its class have been created, so they don’t contribute to size of class as well.

> Local static members (defined in a function) are allocated space at the time when function is invoked and are freed when program terminates.

// 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

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.

> 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

**Typedef**

> Use ‘using’ instead of typedef in C++

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

> There is no major difference between ‘using’ and ‘typedef’ except that the alias declaration is compatible with templates, whereas the C style typedef is not. For example, we can have

template<typename T>

using Details = std::unordered\_map<Student, T>;

But in case of typedef, we need to add map decl. in a struct.

Other difference (using brings clarity):

typedef void(\*func\_pointer)(int);

vs

using func\_pointer = void(\*)(int);

> 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 storage class specifier 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. But you can use typedef with specifiers like:

typedef const int cint;

> 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!

**Volatile**

*An object that has volatile-qualified type may be modified outside the scope of program and is intended to prevent compiler from doing any optimization. That object may’ve been tied to some memory address, or can be modified by signal handler or some asynchronously interrupting function. Such variables are not optimized by compiler.*

Contexts where we might need volatile qualifier:

> Reading from a data port (memory-mapped IO) from which we might have to read variables on the fly.

> Variable modified by signal handler.

> Variables shared by multiple threads

> Variable referenced within ISR

There is no optimization performed on volatile variables.

Example 1

unsigned \*p = Address();

a = \*p;

// other operations

b = \*p;

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

Example 2

Another use for volatile is signal handlers.

int quit = 0;

while (!quit) {}

Compiler converts it into an infinite loop even though quit might be changed by a signal handler.

**Access Specifiers:**

> Protected members couldn’t be accessed directly, only via function in both base and derived classes. So, protected members behave like private variables (verified)

**Functions**

**C language:**

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

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

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

> If there are **insufficient arguments** for the format, the behavior is **undefined**. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. . If a function is present as extra argument, it is evaluated but it should return some value (verified)

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

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

> 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.

> 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.

> If return statement is missing in main, by default 0 is returned.

> Declaring function which returns a function pointer:

servedFuncReturnType (\*servingFunc(servingFunctionArg))(servedFunctionArgs)

> Accessing the address of local variable of callee inside the caller, returned via pointer is undefined behavior.

Some facts on malloc() function:

> The malloc() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized.

> If ptr is NULL, no operation is performed.

> **If the multiplication of nmemb and size would result in integer overflow, then calloc() returns an error. This is advantage of calloc. By contrast, an integer overflow would not be detected in the following call to malloc(),** with the result that an incorrectly sized block of memory would be allocated:

> If realloc() fails, the original block is left untouched; it is not freed or moved. Passing a freed pointer to realloc is undefined behavior. If ptr is NULL, the behavior is the same as calling malloc(new\_size).

> By default, Linux follows an optimistic memory allocation strategy. This means that when malloc() returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer.

> If the size of the space requested is zero, the behavior is implementation defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value (tested on codeblocks), except that the returned pointer shall not be used to access an object.

> malloc just reserve the address space. Pages are fetched only when accessed.

> Free store is implemented in the form of linked list

> When you call free, the memory may or may not be given back to the kernel immediately rather it’s added to free blocks.

> malloc\_trim can be invoked to release any memory now

> malloc rounds up sizes internally to the size of two pointers (i.e. 8 bytes for 32-bit, or 16 bytes for 64-bit). If you separate "consistent" allocations in one thread and "varying" ones in another, they're more likely to be kept separate in separate arenas, which might help. Also, malloc divides the heap areas to arenas and each arena has a lock. A multithreaded program can make use of different arenas.

There is a lockless techniques as well. In this, intermediate chunks are allocated to threads.

MACROS

> MACROS cannot access private variables of the class

Drawback:

> no type-checking

> Difficult to debug as there is just a simple replacement.

> Side-effects such as in examples of typedef int\* intptr; and #define MAC(x) x\*x and MAC(1+2) is invoked.

> No containment. It can be used by any file.

**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. A single functor declaration for int, bool, float etc.

> It is inlined easily.

Type of functors :

**Generator**: A functor 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.

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

is equivalent to:

increment obj;

transform(arr, arr+n, arr, obj(to\_add));

A function object which returns Boolean value is called predicate. In Unary function, it is simply called as predicate while in Binary function, it is called as Binary predicate.

Here are the two types of transform() algorithms:

> Transforming elements

transform ( source.begin(), source.end(),

destination.begin(), UnaryFunc op )

> Combining elements of two sequences

transform ( source1.begin(), source1.end(), source2.begin()

destination.begin(), BinaryFunc op )

**References**

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

> Internally, it is implemented as const pointer

> References should be initialized in constructor initialization list

> Reference can’t be reseated, 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 modify, assuming that the object will never be used again.

> Taking address of alias gives the same value as what we get while we take address of pointed variable. (verified)

> There shall be no arrays of references. They don't have storage of their own and just reference existing objects. So, It we can’t have arrays of references wherein we need size of objects.

> 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.

> 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;

Usage of references:

> Runtime dispatch without pointers.

> In exception handling – ‘throw by value catch by reference’

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

**Classes**

Default members of class:

class Thing {

public:

Thing(); // default constructor

Thing(const Thing&); // copy c'tor

Thing& operator=(const Thing&); // copy-assign

~Thing(); // d'tor

// C++11:

Thing(Thing&&); // move c'tor

Thing& operator=(Thing&&); // move-assign

};

> If a derived class doesn't implement pure virtual function of base class, then it also becomes abstract and will throw error in case its object creation is tried. This condition holds even if we define that pure virtual function outside the class.

> A class declaration can contain static object or pointer of self-type. Declaring an object of self-type will return error – incomplete type <class\_name>

> 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;

Here ‘ptr’ is just a pointer and points to nowhere in particular unless it is assigned a value via class object such as:

A a;

printf(“%d”, a.\*ptr); // will print value of int\_var

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

> Non-member function cannot have cv-qualifier. The const modifier at the end of the function declaration applies to the hidden ‘this’ parameter for member functions.

> Non-static data member initializers (all types) only available in C++11 (even without const)  
> Initialization of static data member without const is an error. And if used with const, without constexpr int is allowed and with constexpr – double, float is also allowed, but not string & alike

> Then why ‘static const int’ is allowed? We can take address of ‘static const int’ declared in class only if it is defined outside the class otherwise we’ll get error. It is because such variables are optimized out by compilers and are not stored as memory objects.

> **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 (verified)

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

> 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, but normal class members are not accessible in static functions.

> 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.

> If a given data member is initialized both in member initialization list as well as in body, the latter takes precedence.

> 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.

> Virtual function cannot be inlined because compiler doesn’t know function of which class is going to be 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 declared, but undefined virtual function will generate compiler error when creation of the object is tried because when compiler creates a virtual table it needs entries of virtual function. **>** A const object can only call const functions (not any other) because in that case “const this” is passed as pointer else err is thrown. (verified)

> 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. Need to be used in move constructors especially.

> Overload is possible if one argument is reference and another one is const reference, but it’s not possible if one is taking normal parameter and other one is taking reference/const reference.

> When a function is declared as const, it can be called on any type of object, const object as well as non-const object but modification to class variable is not allowed. (verified)

> private member functions/variables too are inherited but not accessible (verified via size)

> If virtual function is declared private, it won’t be accessible via runtime dispatch. It may or mayn’t be public in derived class. Take this as vice-versa too.

**Constructors/Destructors**

> Move constructor/assignment operator takes **non-const** reference since we are gonna modify the source

> We will get compiler error in case we try to pass lvalue to function accepting rvalue reference. To get past that error in case you know that that lvalue you are not gonna use, pass std::move(lvalue\_object) to that function.

> Constructors are never inherited.

> Your assignment operator is technically inherited, but then it's hidden by the default copy assignment operator in the derived class

> 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.

> 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

> 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

> **Derived class constructor is implicitly deleted if base class constructor is private.**

> In multiple inheritance, when we use ‘virtual’ keyword, the default constructor of grandparent class is called even if the parent classes explicitly call parameterized constructor. To make it happen call parameterized constructor from derived class

> We won't get compilation error if we don't provide const in user defined copy constructor.

> We can have return statement inside both ctor and dtor but return type is not allowed.

> There can only one destructor in a class with classname preceded by ~, no parameters and no return type.

> 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

> We don’t need virtual keyword in destructor of derived class.

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

> Constructor cannot be static member function.

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

class SomeType

{

int number;

public:

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

SomeType() : SomeType(42) {}

};

**> You can call parameterized base class constructor by normal derived class constructor.**

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

> 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 constructor or 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**

> Virtual constructors don't make sense:

To call a virtual function, we need to have its entry in vtable which is setup by constructor. Now, we intends to call a function which is supposed to present in the table which it has to create.

> Compiler creates a copy constructor if we don't write our own. Compiler creates 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

> 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.

> Virtual constructor is not possible but we can create it using following technique -> static Base \*Create(int id); Delegate the creation of object to base class from User class so appropriate type of Derived class is created.

**Initializer list is required for initialization of**;

> non-static const data members:

> reference members:

> member objects which do not have default constructor

> the parameterized constructor of the base class can only be called using Initializer list

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

**Initialization in Constructor required:**

**>** When there is memory allocation.

> If we want to assign specific values to array

> To initialize a struct.

> **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.

> 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 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.

**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 {

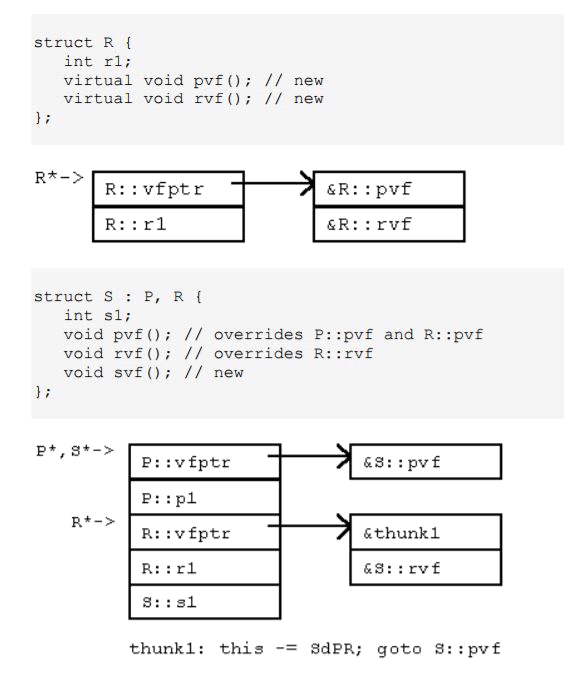
int p1;

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 (S\*)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. This is done via typeinfo data just above each sub-vtable. 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. We have typeinfo above each vtable. In case of multiple inheritance, first parent and child contents are clubbed and referred via same vptr.

**Copy Elision:**

Elision means removal. Copy Elision is technique of memory-optimization which prevents unnecessary copying of objects especially when object is returned by value from a method

C c2 = C(42); // Theoretically, copy-ctor should be called but compiler just calls the constructor -> C (int c){}. Show this example as when returning from function.

// To prevent copy-elision, pass -fno-elide-constructors while compiling

**Primitive data types**

Expressions such as the given below might fail because -1 = 11111111111:

**for** (unsigned int i = -1; i < ARRAY\_SIZE(arr) - 1; ++i)

> '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'

> Similarly, signed int is promoted to ‘unsigned int’ in case other operand is of type ‘unsigned int’. So, result of an expression involving an int and an unsigned int would be non-negative.

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.

We cannot have array of void data type

In C, arr, &arr, &arr[0] gives same output.

arr and &arr[0] is of type int\* while

&arr is of type pointer to array

int \*p = &arr will give error

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

**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.

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

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

Placement new

It can be used when we are allocating memory from an already allocated pool.

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

// placement new in buf

int \*pInt = new (buf) int(3);

We need to explicitly call destructor in case object is created via placement new.

Advantages:

> Optimized use of memory

> There’s no danger of allocation failure since the memory has already been allocated

> In case of ternary operator, return type is determined by exp2; e.g int m = exp

1?exp2:exp3;

> 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.

> Overloaded conversion operators must be a member method and shouldn’t specify return type though value of desired time will be returned.

> We can do A a = b; if we have conversion operator in B for A

> cout << (bmf == 38) ?15 : 10; will print either 0 or 1, because '<<' has high precedence than '?'

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

> Allocating a buffer using malloc() and free’ing it using delete operator is undefined behavior.

When tried, no compilation error was observed.

> delete without '[]' in case of array or with ‘[]’ in case of primitive type doesn’t give any error. It’s a logical bug.

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

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

Multiply without operator -

#define A x

#define B y

char arr[A][B];

use sizeof

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 comma operator (lowest precedence) 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 and doesn't evaluates the expression

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

**Casting**

Casting is a programming technique that helps change 1 type to another for specific purposes:

**Upcasting**

> Upcasting can cause object slicing when a derived class object is passed by value as a base class object and you won’t be able to call derived class functions with that base class object.

**Downcasting**

It 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**.**

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

> *Static Cast won’t allow casting between unrelated data type...like pointer to character and pointer to integer. 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;

\*p = 5; // run-time error: stack corruption

Actual use of static case:

> Integer to float conversion in mul. and div. operations.

> converting from buffer to object. Key data to key object in C++

> It's about how much type-safety you want to impose.

When you write (bar) foo (which is equivalent to reinterpret\_cast<bar> foo if you haven't provided a type conversion operator) you are telling the compiler to ignore type safety, and just do as it's told.

**Dynamic cast**

It’s a kind of typecasting which can be applied **only** when the source and the target classes are polymorphic (has implicit or explicit virtual functions) else use static\_cast.

ptr\_a = dynamic\_cast<A\*>(ptr\_b); // successful if ptr\_b was initially pointing to A object

**Actual use:**

// Below function handles both debug and error messages.

void DebugLog::LogMessage(Message \*msg) {

DebugMessage \*debug;

ErrorMessage \*err;

if(debug = dynamic\_cast< DebugMessage \*>(msg))

// debug message

else if(err = dynamic\_cast< ErrorMessage \*>(msg))

// xyz message

}

> 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 in dynamic cast and compilation error in case of static\_cast and no issue in C-style cast.

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

> On failure, dynamic\_cast will return nullptr in case of pointers and exception (std::bad\_cast) in case of references.

> dynamic\_cast and typeid are the RTTI operators.

**Re-interpret cast**

It is used when you want to convert data from type1 and type2 which are not related, like pointer to integer.

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

> Assigning float\*/int\* to void\* won’t emit any warning.

> Assigning address of integer to float pointer will emit error, casting will do fine.

**Const cast**

It generally use to either remove constness or add constness to a variable. While static\_cast can do non-const to const it can't go other way around. The const\_cast can do both ways.

**Use of const cast:**

1. When you want to implement T& SomeClass::foo() as well as const T& SomeClass::foo() const. To avoid code duplication, you can apply const\_cast to return value of one function from another.

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\*

3. When you have const parameters used in your API but want to use legacy APIs

int x = 10;

const int\* y = &x; // allowed

cout<<"old value is"<<\*y<<"\n";

int\* z=const\_cast<int \*>(y); // fails without const\_cast

\*z=100; // fails in case we just do \*y=100

cout<<"new value is"<<\*y;

Base operations:

Base \*bb = new Base;

Base \*bd = new Derived;

Derived \*dd = new Derived;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operations | C-Style cast | Static\_cast | Dynamic\_cast | Comments |
| dd = bb  (Downcasting) | Success. Calls base | Success but undefined. Calls base | Success. Fails at runtime | Compilation fails without casting |
| bb = dd  (Upcasting) | Success. Calls derived | Success. Calls derived | Success. Calls derived | Succeeds in all |
| dd = bd | Success. Calls derived | Success. Calls derived | Success. Calls derived | Compilation fails without casting |
|  |  |  |  |  |

**Runtime Polymorphism (using Virtual Functions)**

Polymorphism in general sense means deciding which piece of code to be executed based on something. Runtime polymorphism or late binding is a phenomenon in which the object whose method is invoked is decided at runtime.

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. It can be used to provide common functionality for its derived classes.

> <typeinfo> is used for RT polymorphism. It uses typeid() operator 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.

> 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 work.

> 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. (verified)

> Virtual pointer is inherited by derived classes.

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

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

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

> Apart from virtual functions, we can have function pointers to achieve RT polymorphism. For example, we have fprintf as polymorphic function wherein handle could be returned from fopen or input could be stdout as well.

> Polymorphism only applies when methods have the exact same signature (same number and types of parameters and same return type)

> If parent class has function that takes long and that is not overridden in derived class, but we have integer in derived class. Then call with to func(int) will call parent class.

**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 copied 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.

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 throw 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.

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

> 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).

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

Using keyword:

class Derived : Base {

using Base::greet;

void greet(string s) {

cout << "Hello from " << s << endl;

// Instead of recursing, the greet() method

// of the base class is called.

greet();

}

};

> 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.

**Function Overloading**

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

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\_;

};

**Operator Overloading**

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

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.

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 delete(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;

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

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

**Friend function:**

> We can access private members in a friend function and ofcourse, it’ll be via an object.

> While defining friend function outside the class ... 'friend' keyword and ‘class name’ has to be dropped

> default arguments are not allowed in friend function while it is declared inside a class. though, you can add it in definition. Because as per the standard, a friend function declaration with default args should be the definition too.

> Defining friend function in class is of no use (not accessible)

> Base class friend functions and its constructor/destructor are not inherited inside the derived class.

Examples of friend functions usage can be:

> Overloading cout and cin for objects

> **Making test class as a friend of normal class.**

**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.

Disadvantages of templates:

> Syntax is complex

> They are hard to validate.

> Slow to compile

> Ambiguity resolution

> Error messages are incomprehensible.

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);

template <>

class MyContainer <double, 4> { // Full specialization

....

template <int n>

class MyContainer <double, n> { // Partial specialization

....

template <class T, int n> // Partial specialization (adds something to the base template parameter list: '\*')

class MyContainer<T\*,n> {

....

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.

Random Notes :

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

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};

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.(verified)

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++.

> 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.

> 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.

> 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.

> 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.

> 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];

}

> 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 static function to it. 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];

Why can't we declare a static variable within a structure in C?

> In C, a struct is a contiguous chunk of memory with fields. A static variable cannot be created without changing that (as to implement a static you need to refer to a single memory location from all structs of that type), and that would be a big difference in complexity without much benefit.

**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 behavior is the order in which the arguments to a function are evaluated.

**SOLID Design Principles**

SOLID refers to list of design principles that need to be followed for maintainable, extensible and loosely coupled code. Following are the principles this acronym describes:

> Single Responsibility: Every class should implement a given functionality and should be self-contained or, in other words, class should have only 1 job to perform.

> Open closed principle: It states that objects or entities should be open for extension but closed for modification.

> Liskov Substitution: This means that, in inheritance hierarchy, if S is a subtype of T, then objects of type T may be replaced with objects of type S. Bad case is when Square is defined as type of rectangle.

> Interface seggregation: Interfaces for different clients should be different.

> Dependency Inversion: The higher level layer should not be directly dependent on low level layer, rather they should depend on abstractions.

An example could be a ‘PasswordReminder’ class which depends on some ‘MySQL’ connection. In this case we shouldn’t have ‘MySQL’ class contained in ‘PasswordReminder’ class that connects to database, rather create an interface ‘DBConnectionInterface’ that could be used.

If we talk about usage of virtual functions, then virtual function is an example of 2nd design principle (Open to extend and closed for change)

**PImpl idiom**

When we make a change in header file, then the code containing that header file needs to be recompiled, increasing the compilation time and additional overhead. Using PImpl we separate interface from actual implementation so that we have an extra struct/class that is actually implementing the methods of a class implementation of which we need. E.g. –

// header file:

class Cat {

private:

class CatImpl; // Not defined here

CatImpl \*cat\_; // Handle

public:

Cat(); // Constructor

~Cat(); // Destructor

// Other operations...

Purr();

};

// cpp file

class Cat::CatImpl {

Purr();

... // The actual implementation can be anything

};

Cat::Cat() {

cat\_ = new CatImpl;

}

Cat::~Cat() {

delete cat\_;

}

Cat::Purr(){ cat\_->Purr(); }

CatImpl::Purr(){

printf("purrrrrr");

}

We can use this idiom in case we are writing code for multiple targets.

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

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

> 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.

> The destructor is called before member variables are destroyed.

> The base constructor executes first because the base class has to be initialized before the derived constructor, otherwise the derived constructor could access data members of the base class that haven't been initialized.

The destructors execute in reverse order for a similar reason: if the base destructor were to execute first then the derived destructor could access resources from the base class that are no longer available (freed memory, closed files etc.)

> "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." <--

> Advantage of templates:

Type-safe, Supports generic programming, Improvement over MACROS, Expanded at compile time so not much overhead.

> Disadvantage of templates:

Emitted error message are unreadable. You might encounter ambiguity issues.

List of C++11 features:

**Lambda Expressions**: Anonymous functions that are declared at call point and are explicitly inlined.

**Automatic Type Deduction**; auto x = 0

**Uniform Initialization Syntax**: int a{9};

**Range-based for loop**

**Deleted and Defaulted Functions**- like deleted ctors in Singleton

**nullptr**; int x = nullptr; //compilation failure

**Delegating Constructors**: A constructor may call another constructor in initializer list.

**Move Symantics and r-value references**: Optimized memory usage

**STL Enhancements**: unordered\_set, unordered\_map, unordered\_multiset, and unordered\_multimap

**Multi-threading features –** thread

**Synchronization primitives –** mutex, condition\_variable

**New Smart Pointer Classes**: Only auto\_ptr was present in older version (verified)

**New C++ Algorithms** - all\_of(), any\_of() and none\_of()

**noexcept specifier** - Like we use for new operator

**Tuple**

You can return multiple values from function using tuple

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

int a, b;

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

where fun() is

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

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

}

**Features introduced in C++14**

> **Auto type deduction in lambda expression**.

Earlier, the parameters specified need to be with concrete type like (int a, int b)

auto lambda = [](auto x, auto y) {return x + y;};

> **make\_unique** has been introduced similar to make\_shared added in C++11

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

> **Tuple addressing via type**

The std::tuple type introduced in C++11 allows an aggregate of typed values to be indexed by a compile-time constant integer. C++14 extends this to allow fetching from a tuple by type instead of by index.If the tuple has more than one element of the type, a compile-time error results.

tuple<string, string, int> t("foo", "bar", 7);

int i = get<int>(t); // i == 7

int j = get<2>(t); // Same as before in C++11: j == 7

string s = get<string>(t); // Compile-time error due to ambiguity

> **Variable templates**

In prior versions of [C++](https://en.wikipedia.org/wiki/C%2B%2B), only functions, classes or type aliases could be templated. C++14 allows the creation of variables that are templated.

template<typename T>

constexpr T pi = T(3.141592653589793238462643383);

> **Auto return type deduction**

We now no need to write cumbersome return decltype as in:

template <typename T>

auto foo(T value) -> decltype(value.bar())

{

return value.bar();

}

**Digit separators**

auto integer\_literal = 1'000'000;

auto floating\_point\_literal = 0.000'015'3;

**Numerical literal in binary form**:

Int x = 0b1010 or 0B100001

> [[deprecated]] keyword can be added before a function to warn the user

> **Shared timed mutex**

**> Expression allowed in parameter list of lambda function**

**> The attribute [[deprecated]]**

> **Generic lambdas**

In C++11, lambda function parameters need to be declared with concrete types. C++14 relaxes this requirement, allowing lambda function parameters to be declared with the auto type specifier.[7]

auto lambda = [](auto x, auto y) {return x + y;};

**C++17 features**

> Auto type deduction for class templates. So, based on type

std::string\_view

std::filesystem

std::optional

> New auto-type deduction

auto a{3} initially was interpreted as initializer-list, but now is interpreted as int

> inline specifier for variable which when applied to static integer in class helps getting away with its definition outside the class.

> Nested namespaces:

namespace A {

namespace B {

namespace C {

int i;

}

}

}

The code above can be written like this:

namespace A::B::C {

int i;

}

> std::byte type

STL

Ref 1 # C++ for C Programmers (Coursera)

Ref 2 # Effective Modern C++ (Scott Meyers)

Sequence Containers: implement data structures which can be accessed in a sequential manner.

* vector
* list
* deque
* array(Introduced in C++11)
* forward\_list( Introduced in C++11)

Container Adaptors : provide a different interface for sequential containers.

* Queue (uses deque internally)
* priority\_queue (uses vector internally because of binary heap representation)
* stack (uses deque internally for efficiency purpose)

Associative Containers : Implement data structures that store key-value pairs. Both sorted and unsorted versions are present. Sorted ones can be searched in O(log n) time while unsorted can be searched in O(1) complexity.

* set
* map
* bitset
* multiset
* multimap
* unordered\_map
* unordered\_set
* unordered\_multimap
* unordered\_multiset

**<vector>**

It represents a contiguous area of allocation.

Difference between size(), max\_size() and capacity();

Size() tell us the number of objects allocated in vector. Capacity tell us the current size of space allocated and max\_size() tells us the maximum size a vector can be allocated space (depending upon the limitation imposed by system)

The performance will take a hit if we are going to insert large number of elements, so call vector.reserve() function.

vector<int> vec(2);

Vec.push\_back(12); // size of vector is now 3

**<array> (Since C++11)**

Fixed size data structure and don’t use any allocator

Empty array holds size of 1 so that initializer is valid.

Iterators – begin(), end(), rbegin(), rend(), cbegin(), cend(), crbegin(), crend()

size and max\_size returns the number of elements in array. max\_size is there only for consistency.

It also provides front() and back() functions.

arr.fill(<val>) fills the array with value ‘val’

 since C++ 17, it is allowed to omit the type and size.

std::array myArray { 9, 7, 5, 3, 1 }; // The type is deduced to std::array<int, 5>

**<deque>**

It refers to doubly-ended-queue

It grows dynamically from both sides with constant insertion and deletion time from both ends. Though insertion and deletion from middle takes O(n). Deque allocates its storage in the form of blocks of predefined size (generally 512 bytes). So, deque grows linearly as a complete block is allocated in one go

It uses an allocator object to dynamically handle its storage needs.

When assignment operator is used, LHS deque takes the size of RHS deque.

Also, if exception is thrown, container remains in valid state.

Iterators – begin(), end(), rbegin(), rend(), cbegin(), cend(), crbegin(), crend()

size() and max\_size() are not similar in case of deque.

shrink\_to\_fit (C++11) s a request to container implementation to modify its size to the value of number of elements in it.

It provides front(), back(), push\_front(), pop\_front(), push\_back() and pop\_back() functions

It also provides emplace(), emplace\_front() and emplace\_back() functions (All C++11)



**<forward\_list>**

Just for efficiency of 1 word, forward\_list lacks the size() member function and standardization agency thought it should be as matchable as hand-written linked list.

distance(list.begin(), list.end()) algorithm (in <iterator> with its begin and end

iterator before\_begin()

The iterator returned shall not be dereferenced: It is meant to be used as an argument for member functions [emplace\_after](http://www.cplusplus.com/forward_list::emplace_after), [insert\_after](http://www.cplusplus.com/forward_list::insert_after), [erase\_after](http://www.cplusplus.com/forward_list::erase_after) or [splice\_after](http://www.cplusplus.com/forward_list::splice_after)

Iterators – begin(), cbegin(),before\_begin(), cbefore\_begin(),end(), cend(), cbegin() [**Reverse iterators not available in forward\_list as the name suggests**]

It provides front()

Utility functions: remove(), remove\_if(), unique() [removes duplicate values], merge()[Merge sorted lists], sort(), reverse(), splice\_after()

**Note**: splice() takes element from second forward\_list (Either of the complete list, range of elements or single item) and add to first forward\_list. Merge does sorted merge. Both the operations, merge and splice, modify the second list.

**<list>**

size – pre-C++11, it was mentioned that complexity of size() function should be O(1) but that wasn’t required. Hence, some implementation used std::distance(first, last) function which makes the complexity of operation as O(n)

gcc 4.8 has O(n) [using distance function] while gcc 5.0 has O(1)

splice(Iterator position, Container c); removes elements from the container c to the calling function. The overloads of splice includes transferring just 1 element, range of elements or complete list which is the default operation. But in C++11, this was made strictly O(1) by language standard.

list1\_name.splice (iterator position, list2)

list1\_name.splice (iterator position, list2, iterator i)

list1\_name.splice (iterator position, list2, iterator first, iterator last)

**Difference between vector and deque**

Vector is guaranteed to provide contiguous storage of data while deque doesn’t though both provide index level access of data. You can insert data in front in case of deque but it isn’t possible in case of vector. Due to uncertainty of data storage deque are somewhat less efficient as compare to vector. We don’t have push\_front() for vector as that might result in moving every single element one element back resulting in O(n) complexity. So, for ‘n’ elements, it’ll result in O(n^2) complexity.

**Difference between deque and list**

deque provides random access iteration as it is implemented in the form of vector. Also, erasing an element from deque invalidates iterator while list doesn’t. Also, deletion a node from deque take linear time in case of deque and O(1) in case of list, but to reach to the element to be deleted, the time complexity will be O(n) for both list and deque. Unlike vectors, contiguous storage allocation may not be guaranteed..

Why stack uses deque?

It appears that the way deque is usually implemented is a variable size array of fixed size arrays. This makes growing faster than a vector (which requires reallocation and copying), so for something like a stack which is all about adding and removing elements, deque is likely a better choice. Similary, deque is used for queue as well. We can use List as well.

priority\_queue requires indexing heavily, as every removal and insertion requires you to run pop\_heap() or push\_heap(). This probably makes **vector** a better choice there since adding an element is still amortized constant anyways.

**Iterators**:

Iterators are classes that are used to access members of the container classes, and can be used in a similar manner to pointers. For example, one might use an iterator to step through the elements of a vector. There are several different types of iterators:

> input\_iterator: Read values with forward movement. These can be incremented, compared, and dereferenced.

> output\_iterator Write values with forward movement. These can be incremented and dereferenced.

> forward\_iterator: Read or write values with forward movement. These combine the functionality of input and output iterators with the ability to store the iterator’s value.

> bidirectional\_iterator: Read and write values with forward and backward movement. These are like the forward iterators, but you can increment and decrement them.

> random\_iterator: Read and write values with random access. These are the most powerful iterators, combining the functionality of bidirectional iterators with the ability to do pointer arithmetic and pointer comparisons.

> reverse\_iterator: Either a random iterator or a bidirectional iterator that moves in reverse direction.

Each of the container class is associated with an iterator.

When you are in a template and a parameter has exactly type T&& for some deduced type T, then what you might get when instantiating the template is not an rvalue reference. Indeed, the parameter of function can bind to both lvalues and rvalues. On a side note, auto&&works similarly.

> When an array is passed as an argument to function which accepts value by reference, then the parameter in function call is not a pointer rather an array – int (&)[array\_size]

// return size of an array as a compile-time constant. (The

// array parameter has no name, because we care only about

// the number of elements it contains.)

template<typename T, std::size\_t N> // see info

constexpr std::size\_t arraySize(T (&)[N]) noexcept // below on

{ // constexpr

return N; // and

} // noexcept

\* Mentioning noexcept turns an exception throw into a call to std::terminate().

• During template type deduction, arguments that are references are treated as non-references, i.e., their reference-ness is ignored.

• When deducing types for universal reference parameters, lvalue arguments get special treatment.

• When deducing types for by-value parameters, const and/or volatile arguments are treated as non-const and non-volatile.

• During template type deduction, arguments that are array or function names decay to pointers, unless they’re used to initialize references.

**Use of override keyword:**

When there are large number of classes in inherited hierarchy and you think that you might misspell the name of overriding function, append override so that compiler can know that this function must be overriding a function. For example: A common mistake is to spell SetUp() as Setup() with a small u - Use override in C++11 to make sure you spelled it correctly. In case the function doesn’t override anything, you’ll get compilation error.

**Note:**

c\*begin/end() iterators are included in C++11

<initializer\_list> is a parameter introduced in constructor of containers in C++11

**auto keyword**

array and function names decay into pointers for non-reference type specifiers.

const char name[] = “Gaurav”;

auto& arr2 = name; // arr2 type is const char (&)[13]

C++11 allows a variable declaration in 4 forms:

* auto a = 23;
* auto a(23);
* auto a{23};
* auto a = {23};

The first 2 statements declares an integer while the last 2 declares an initializer-list of 1 element as of type: std::initializer\_list<int>

Hence, when auto variable encounters braces, it expects an initializer-list of homogeneous elements. The following statement throws error:

auto x5 = { 1, 2, 3.0 }; // can't deduce T for std::initializer\_list<T> (data doesn’t resolve to a single type)

If similar initializer-list is passed to function template, deduction fails.

* auto in a function return type or a lambda parameter implies template type deduction, not auto type deduction.

In C++14, we can omit trailing return type for both functions and lambdas while in C++11 it was possible only for lambdas.

auto, when used as a return type, strip off reference. So, when a function which returns a reference with return type deduction left to auto will fail at the following:

deque<int> q;

auto try(deque<int>& q, int i);

try(q, 12) = 23; // r-value returned, the assignment to which is not allowed

working declaration:

decltype(auto) try(q, 12);

Given name of expression, decltype gives us the type of parameter passed.

decltype can be used in trailing return type, recently introduced in C++11.

For example: This code gives error because a and b are used before their type is declared.

template<class T>

decltype(a\*b) mul(T a, T b){

return a\*b;

}

while this works fine;

template<class T>

auto mul(T a, T b) -> decltype(a\*b){

return a\*b;

}

Notes:

If we are using auto to declare a reference implicitly, then we should put it as 'auto&'

**l-values, r-values and move semantics**

An lvalue is an expression e that may appear on the left or on the right hand side of an assignment, whereas an rvalue is an expression that can only appear on the right hand side of an assignment.

An lvalue is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator. An rvalue is an expression that is not an lvalue.

The return value of a function is an l-value if and only if it is a reference

This is because we have to support chaining, such as

a = b = c;

It means that the assignment operator will have to return a reference.

We want assignment operation to work like this:

// [...]

// swap m\_pResource and rhs.m\_pResource

// [...]

These are called move semantics.

**rvalue references**

If X is any type, then X&& is called an rvalue reference to X. For better distinction, the ordinary reference X& is now also called an lvalue reference.

X x;

X foobar();

foo(x); // argument is lvalue: calls foo(X&)

foo(foobar()); // argument is rvalue: calls foo(X&&)

Things that are declared as rvalue reference can be lvalues or rvalues. The distinguishing criterion is: if it has a name, then it is an lvalue. Otherwise, it is an rvalue.

Here is an example of something that is declared as an rvalue reference and does not have a name, and is therefore an rvalue:

X&& goo();

X x = goo(); // calls X(X&& rhs) because the thing on

// the right hand side has no name

Notes:

> The T&& in the the templated functions do not necessarily denote an rvalue reference, it depends on the type that is used to instantiate the template.

> A move constructor doesn't take const argument, after all it's going to move that

**Smart Pointers**

// include this header to use C++ smart pointers.

#include <memory>

Four types

1. std::auto\_ptr (deprecated C++11 onwards because it didn’t support move semantics)

auto\_ptr<int> p1(new int(42));

auto\_ptr<int> p2 = p1; // at this point p1 is a bad pointer

Here, unique\_ptr will throw compilation error but auto\_ptr won’t.

2. std::unique\_ptr

Copy constructor and assignment operator of unique pointer are deleted, but it is possible to return unique\_ptr from a function, since that will be moved.

auto a = std::make\_unique<Class>(//parameters);

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

std::unique\_ptr<int> p2 = p1; //Compile error(Use of deleted function)

std::unique\_ptr<int> p3 = std::move(p1); //Transfers ownership. p3 now owns the memory and p1 is rendered invalid.

p3.reset(); //Deletes the memory.

p1.reset(); //Does nothing.

std::shared\_ptr to std::unique\_ptr is not allowed.

3. std::shared\_ptr

std::shared\_ptr manages two entities:

the control block (stores meta data such as ref-counts, type-erased deleter, etc) and the object being managed

A shared\_ptr can be created from unique\_ptr with make\_shared function call as below:

std::shared\_ptr<int> s\_ptr{std::move(u\_ptr)};

allocate\_shared<class\_name> can be used when custom allocator is required.

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

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

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

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

Problem with shared\_ptr & unique\_ptr:

void main( )

{

int\* p = new int;

shared\_ptr<int> sptr1( p);

shared\_ptr<int> sptr2( p );

}

The program will crash

4. std::weak\_ptr

Basically used to check the validity of smart pointer if it’s deleted. Shared ownership can be retrieved using weak\_point.lock() function call. It helps resolving cyclic references.

Suppose class A has a shared pointer that points to class B which in turns has another pointer that points to A. The following code will create a cyclic reference and results in memory leak when control goes out of this scope

shared\_ptr<B> sptrB( new B );

shared\_ptr<A> sptrA( new A );

sptrB->m\_sptrA = sptrA;

sptrA->m\_sptrB = sptrB;

When sptrA and sptrB goes out of scope then pointers won’t be deleted because each has one referent in other’s class.

which can only be resolved if shared\_ptr is replaced by weak pointer since that will not increase the reference count.

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

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

{

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

if(p2) // As p2 is initialized from a weak pointer, you have to check if the memory still exists!

{

//Do something with p2

}

} //p2 is destroyed. Memory is owned by p1.

p1.reset(); //Memory is deleted.

std::shared\_ptr<int> p3 = wp1.lock(); //Memory is gone, so we get an empty shared\_ptr.

if(p3)

{

//Will not execute this.

}

> Practical usage of weak\_ptr

Suppose you have Team and Member objects.

Obviously it's a relationship : the Team object will have pointers to its Members. And it's likely that the members will also have a back pointer to their Team object.

Then you have a dependency cycle. If you use shared\_ptr, objects will no longer be automatically freed when you abandon reference on them, because they reference each other in a cyclic way. This is a memory leak.

You break this by using weak\_ptr. The "owner" typically use shared\_ptr and the "owned" use a weak\_ptr to its parent, and convert it *temporarily* to shared\_ptr when it needs access to its parent.

Store a weak ptr :

weak\_ptr<Parent> parentWeakPtr\_ = parentSharedPtr; // automatic conversion to weak from shared

then use it when needed

shared\_ptr<Parent> tempParentSharedPtr = parentWeakPtr\_.lock(); // on the stack, from the weak ptr

if( !tempParentSharedPtr ) {

// yes it may failed if parent was freed since we stored weak\_ptr

} else {

// do stuff

}

// tempParentSharedPtr is released when it goes out of scope

Sample smart\_ptr class

// A generic smart pointer class

template <class T>

class SmartPtr

{

   T \*ptr;  // Actual pointer

public:

   // Constructor

   explicit SmartPtr(T \*p = NULL) { ptr = p; }

   // Destructor

   ~SmartPtr() { delete(ptr); }

   // overloading dereferencing operator

SmartPtr(const SmartPtr& s) = delete;

SmartPtr& operator=(const SmartPtr& s) = delete;

   T & operator \* () { return \*ptr; }

   // overloading arrow operator so that members of T can be accessed

   // like a pointer (useful if T represents a class or struct or

   // union type)

   T \* operator -> () { return ptr; }

};

**Advantages of make\_shared()**

std::make\_shared performs one heap-allocation (with managed object allocated memory along with control block), whereas calling the std::shared\_ptr constructor performs two (One for actual heap allocation and second for control block allocation). When we do 2 allocation we can have benefit when all shared\_ptr ref count hits 0, we can remove raw pointer and control block will be deleted when weak\_ptr goes out of scope. While in case of single allocation both of the allocations have to be vanished in 1 go.

std::make\_shared is exception-safe. Consider the below function declaration:

func(std::shared\_ptr<Object>(new Object()),std::shared\_ptr<Object>(new Object()));

void func(std::shared\_ptr<Object>& obj1, std::shared\_ptr<Object>& obj2){}

Suppose the function execution goes this way

1. Allocate memory for obj1
2. Allocate memory for obj2
3. call shared\_ptr<Object>(obj1);
4. call shared\_ptr<Object>(obj2);

Suppose an exception occurs at line 2. In that case the memory allocated at point 1 will be leaked since shared\_ptr constructor hasn’t been called yet, hence raw pointer is still not owned. To resolve the above issues, replace the shared\_ptr constructor with make\_shared<Object>();

Downside of make\_shared:

Since there's only one allocation, the pointee's memory cannot be deallocated until the control block is no longer in use. A weak\_ptr can keep the control block alive indefinitely.

> make\_shared cannot access protected or private constructor

shared\_ptr maintains certain housekeeping information such as:

A “strong reference” count to track the number of shared\_ptrs currently keeping the object alive. The shared object is destroyed (and possibly deallocated) when the last strong reference goes away.

A “weak reference” count to track the number of weak\_ptrs currently observing the object. The shared housekeeping control block is destroyed and deallocated (and the shared object is deallocated if it was not already) when the last weak reference goes away.

Notes:

* auto\_ptr doesn't work for arrays. When it destroys the object it owns, it uses delete object while unique\_ptr calls delete[] which is correct way to free memory.
* std::shared\_ptr supports array types (as of C++17), but std::make\_shared does not.
* The std::make\_shared is a preferred way to construct a shared\_ptr because it builds the managed object within the control block:
* The weak count does not play any role in deciding the lifetime of the managed object, which is deleted when the reference count reaches zero. However, the control block itself is not deleted until the weak count also reaches zero.
* Control block has the following info – ref count of weak and shared pointer, pointer to managed object, Deleter and Allocator. Only when both refcounts are 0 then the control block is deleted.

> You can have multiple pointers pointing to that memory. It would create a false sense of security if the pointer you specified for the delete got set to null, but all the other pointers did not and it is computationally intense to search for all those pointers. Other reason could be that adding an extra statement might increase computational time.

**Final Keyword:**

C++11 introduced the keyword “final” which can be appended in front of class name to make it underivable as:

class A final {};

Another use of final keyword is to prevent a virtual function from being overridden in derived class.

virtual void myfun() final {}

std::transform function performs operation on all elements present in the set or in other words std::transform applies the given function to a range and stores the result in another range,

e.g. below function adds up 2 arrays:

transform(arr1, arr1+n, arr2, plus<int>());

**constexpr specifier**

constexpr specifies that the value of an object or a function can be evaluated at compile time.

A function be declared as consexpr-

constexpr int product(int x, int y)

1. In C++ 11, a constexpr function should contain only one return statement. C++ 14 allows more than one statements.
2. constexpr function should refer only constant global variables.
3. constexpr function can call only other constexpr function not simple function.
4. Function should not be of void type and some operator like prefix increment (++v) are not allowed in consexpr function

**constexpr vs inline functions:**

In inline functions, expressions are always evaluated at run time. constexpr is different, here expressions are evaluated at compile time.

Difference between const and constexpr -

> const can be used only for member functions while constexpr can be used for both member and non-member functions.

> For constexpr to be applicable on a function, it should meet a certain criteria - function body should be non-virtual, arguments and return type should be literal.

For a function to be fit for use in constant expressions, it must be explicitly declared

> Unlike const, constexpr can be applied to constructors and non-member functions also

constexpr float x = 42.0;

constexpr float y{108};

constexpr float z = exp(5, 3);

constexpr int i; // Error! Not initialized

int j = 0;

constexpr int k = j + 1; //Error! j not a constant expression

constexpr;

template<int N>

class list { };

constexpr int sqr1(int arg) { return arg \* arg; }

const int X = 2;

list<sqr1(X)> mylist1; // OK: sqr1 is constexpr, but we will get error here if function is not prefixed with constexpr

In C++11, constexpr implies const, while in C++14 and C++17 that is not the case. A member function declared under C++11 as

constexpr void f(); needs to be declared as

constexpr void f() const;

**Lambda expression in C++**

C++11 introduced lambda expression to allow us write an inline function which can be used for short snippets of code that are not going to be reuse and not worth naming. In its simplest form lambda expression can be defined as follows:

[ capture clause ] (parameters) -> return-type

{

definition of method

}

Generally return-type in lambda expression are evaluated by compiler itself and we don’t need to specify that explicitly and -> return-type part can be ignored but in some complex case as in conditional statement, compiler can’t make out the return type and we need to specify that.

Following is the sample usage:

// lambda expression to print vector

for\_each(v.begin(), v.end(), [](int i) {

std::cout << i << " ";

});

sort(v.begin(), v.end(), [](const int& a, const int& b){

        return a > b;

});

auto pushinto = [&] (int m)

{

    v1.push\_back(m);

    v2.push\_back(m);

};

// it pushes 20 in both v1 and v2

pushinto(20);

A lambda expression can have more power than an ordinary function by having access to variables from the enclosing scope. We can capture external variables from enclosing scope by three ways:

* Capture by reference
* Capture by value
* Capture by both (mixed capture)

Syntax used for capturing variables:

* [&] : capture all external variable by reference
* [=] : capture all external variable by value
* [a, &b] : capture a by value and b by reference
* [=, &a] : capture all except a by value and a by reference

> A lambda with empty capture clause [ ] can access only those variable which are local to it.

> Only capture close is mandatory in lambda function.

> there is no standard type for lambdas. A lambda’s type is implementation defined, and the only way to capture a lambda with no conversion is by using auto:

auto f2 = [](){};

> Lambdas can be inlined easily. Hence, good performance-wise.

>

**For-Each loop:**

C++11 introduced a concise form of loop statement, especially to reduce code bloating by loops used on iterators. Its syntax is:

for( <data\_type> <var\_name> : <data\_container) {}

Generally, auto keyword with reference is used in place of <data\_type> so that it is automatically deduced.

Notes:

> for(auto a : arr) will call copy constructor. To avoid it use for(auto&a : arr)

**decltype**

In the C++ programming language, decltype is a keyword used to query the type of an expression where it is often difficult, or even impossible, to express types that depend on template parameters.. Introduced in C++11, its primary intended use is in generic programming,

int a = 0;

decltype((a)) b = a;

Since 'a' is parenthesized decltype((a)) is int&

**Initializer Lists:**

Initializer list is a new functionality added to C++11 which is a kind of proxy to the temporary array where a list of given data types is kept inside the brace and used as such in:

>Adding multiple values of type <T> in a vector/list/set.

>Returning a set of variables of type <T>

>Passing a list of given data type <T>

Whenever such operation is performed a variable of type std::initializer\_list is created

The universal form based on curly-brace-delimited initializer lists prevents narrowing conversions

int i2 {7.2}; // error : floating-point to integer conversion

int i3 = {7.2}; // error : floating-point to integer conversion (the = is redundant)

**The underlying array is a temporary array of type const T[N]**

> 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;

> Within the initializer-list of a braced-init-list, the initializer-clauses, including any that result from pack expansions (§14.5.3), are evaluated in the order in which they appear.

h(f(), g()); // non-deterministic

h{f(), g()} // definite order

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

}

};

**RAII**

Resource Acquisition Is Initialization

Resource Acquisition Is Initialization or RAII, is a C++ programming technique[1][2] which binds the life cycle of a resource that must be acquired before use (allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection—anything that exists in limited supply) to the lifetime of an object that holds that resource.

*It basically includes 2 things -*

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

*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.

**C++11 Concurrency**

C++11, in its standard template library, provides support of multithreading via thread class and classes for mutual exclusion, including condition variables.

These are:

> **std::mutex**:

The mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. It provides the functions – lock(), try\_lock() and unlock(). try\_lock() returns true in case lock() was successful, else it returns false. In std::mutex copy constructor and assignment operator are mentioned delete.

A condition\_variable needs a unique\_lock.

> **std::condition\_variable:**

The condition\_variable class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until another thread both modifies a shared variable (the condition), and notifies the condition\_variable. condition\_variable calls any of wait(), wait\_for(), or wait\_until() methods to wait for a specified condition to be held true. The wait operations atomically release the mutex and suspend the execution of the thread.

**Example:**

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

// ...

void f()

{

unique\_lock<mutex> lck {m}; // create a unique\_lock

lck.lock(); // or lck.try\_lock()

// Manipulate shared data

lck.unlock(); // If not called, mutex will be automatically unlocked in destructor when exception is thrown or scope is exited. Hence, better than mutex.lock()

**Or we can do this as**:

mutex m;

m.lock();// or m.try\_lock()

// Manipulate shared data

m.unlock();

}

> We can timed our lock using timed\_mutex:

std::timed\_mutex m;

m.try\_lock\_for(chrono::seconds(10));

There is recursive\_timed\_mutex too.

We can defer a lock :

std::unique\_lock lck(m,std::defer\_lock);

We can acquire multiple lock in one go to prevent deadlocks occurring because of sequential locking

std::unique\_lock<mutex> lck1(m1,std::defer\_lock); // make locks but don't yet try to acquire the mutexes

std::unique\_lock lck2<mutex> (m2,std::defer\_lock);

std::unique\_lock lck3<mutex> (m3,std::defer\_lock);

lock(lck1,lck2,lck3);

All these mutexes will be unlocked once control goes out of scope or exception is thrown.

// unique\_lock can be moved, not replicated

std::unique\_lock<std::mutex> guard1(\_mu);

std::unique\_lock<std::mutex> guard2 = guard1; // error

std::unique\_lock<std::mutex> guard2 = std::move(guard1); // ok

// lock\_guard cannot be moved and duplicated

std::lock\_guard<std::mutex> guard1(\_mu);

std::lock\_guard<std::mutex> guard2 = guard1; // error

std::lock\_guard<std::mutex> guard2 = std::move(guard1); // error

scoped\_lock (C++17 only) vs lock\_guard:

Both will be unlocked once control goes out of scope. Scoped\_lock constructor is variadic. It can hold multiple mutexes, but lock\_guard isn’t.

To pass value by reference to a thread use std::ref as represented below:

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

or

std::thread t1 { bind(func, var1, var2) };

We can return a value from thread either through a variable passed by reference or using a std::promise/std::future pair

**Promise:**

It is an object in which the called thread sets the value while the same is read at caller side using future.

promise<T> pr;

// We can either set value or exception depending upon the result pr.set\_value(res);

p.set\_exception(std::current\_exception());

Notes:

The single and important difference is that std::scoped\_lock has a variadic constructor taking more than one mutex. This allows to lock multiple mutexes in a deadlock avoiding way as if std::lockwas used.

{

// safely locked as if using std::lock

std::scoped\_lock<std::mutex, std::mutex> lock(mutex1, mutex2);

}

**Thread Pool:**

To implement a thread pool, we will need a vector/array of threads. Another data structure will be vector/array of functions. To accomplish the latter part, we will use function<void()> type object that can handle everything callable viz. function, function pointer, struct with overloaded ‘()’ operator and lambda expression.

Notes:

* In case a thread terminates while owning a mutex, the result is undefined.
* recursive\_mutex is used in recursive functions.
* std::mutex is neither copyable and copy-assignable nor moveable
* Assignment operation to unique\_lock makes the unique\_lock on LHS to unlock the mutex held by it and takes ownership of the resource on RHS.
* If lock is called by a thread that already owns the mutex, the behaviour is undefined.
* lock() is usually not called directly: std::unique\_lock and std::lock\_guard are used to manage exclusive locking.
* To call a class member function as thread, pass address of that object to satisfy requirement of ‘this’ operator.
* Difference between lock\_guard and unique\_lock is that lock\_guard doesn’t have function lock() while unique\_lock provides it. In a function, when you have to repeatedly lock and unlock a mutex, use unique\_lock. So, in multithreading when you need to lock and unlock the mutex, use unique\_lock

General programming problems:

Heap Corruption Causes:

>> Writing outside the allocated array length

>> Casting to wrong type

>> Pointer pointing to garbage

>> Double free

>> Dangling pointer

>> Custom: Function returning pointer has no declaration.

**New Headers**

**<ratio>**

Declares ratio template and operation on ratio objects.

ratio\_add - Add two ratios (class template )

ratio\_subtract - Subtract ratios (class template )

ratio\_multiply - Multiply two ratios (class template )

ratio\_divide - Divide ratios (class template )

ratio\_equal - Compare ratios (class template )

ratio\_not\_equal - Compare ratios for inequality (class template )

ratio\_less - Compare ratios for less-than inequality (class template )

ratio\_less\_equal - Compare ratios for equality or less-than inequality (class template )

ratio\_greater - Compare ratios for greater than inequality (class template )

ratio\_greater\_equal - Compare ratios for equality or greater-than inequality (class template )

**<type\_traits>**

This header defines a series of classes to obtain type information on compile-time. It contains most of the functions to check the in-built types of objects and mostly used in template metaprogramming

std::is\_integral<T>::value will return true in case T is of integer type

**<functional>**

This header defines the commonly used functors such as : plus<T>(), logical\_and/or/not<T>(), less\_equal<T> (), negate<T> (), greater<T> (), less<T> (), not1<T> (), not1<T>(), bit\_and/xor/or<T>()

equal\_to<T>(),

**Variadic templates**

Variadic template is a template, which can take an arbitrary number of template arguments of any type. Both the classes & functions can be variadic. Here's a variadic class template:

template<typename... Arguments>

class VariadicTemplate;

VariadicTemplate<double, float> instance;

VariadicTemplate<bool, unsigned short int, long> instance;

VariadicTemplate<char, std::vector<int>, std::string, std::string, std::vector<long long>> instance;

template<typename... Arguments>

void SampleFunction(Arguments... parameters);

Here's a function template. The contents of the variadic template arguments are called *parameter packs*. These packs will then be unpacked inside the function parameters. For example, if you create a function call to the previous variadic function template...  
  
SampleFunction<int, int>(16, 24);

**<algorithm>**

// Searches for a number ‘val’ in a sorted sequence ..

/\* Reference sequences for below operations \*/

vector<int> v{ 1, 3, 5, 8, 9, 23, 45, 79, 98, 101};

bool b = std::binary\_search(ForwardIterator first, ForwardIterator last, const T& val)

**// To perform an operation either on one sequence or collectively on 2 sequence with their result // being stored in another container.**

o\_iterator transform (i\_iterator first1, i\_iterator last1, o\_iterator result, UnaryOperation op);

o\_iterator transform (i\_iterator1 first1, i\_iterator1 last1,i\_iterator2 first2, o\_iterator result, BinaryOperation binary\_op);

**Example:**

transform(v.begin(), v.end(), output.begin(), output2.begin(), [](int i, int j) { return i+j; });

**// It counts the occurrences of x in vector.**

count(first\_iterator, last\_iterator,x)

**// Returns an iterator pointing to the first element in the range [first,last) which has a value not less than ‘x’.**

ForwardIterator = lower\_bound(first\_iterator, last\_iterator, x);

**// Returns an iterator pointing to the first element in the range [first,last) which has a value greater than ‘x’.**

upper\_bound(first\_iterator, last\_iterator, x);

**// Reverses the container**

reverse(BidirectionalIterator first, BidirectionalIterator last)

**// Convert vector to string**

std::ostringstream vts

std::copy(vec.begin(), vec.end(),std::ostream\_iterator<int>(vts, ""));

**// Check if v1 and v2 are permutation of each other**

is\_permutation(v1.begin(), v1.end(), v2.begin()))

**//This programs checks if there is a mismatch between 2 sequences. If there is, what’s the positions of mismatch entity.**

using vecItr = vector<int>::iterator;

pair<vecItr, vecItr> pr = mismatch(v1.begin(), v1.end(), v2.begin());

//**Different types of copy() methods**

copy(start\_iter1, end\_iter1, start\_iter2);

copy\_n(start\_iter1, end\_iter1, num\_elements, start\_iter2);

copy\_if(start\_iter1, end\_iter1, start\_iter2, unary\_predicate);

copy\_backwards(start\_iter1, end\_iter1, start\_iter2);

**// Using for\_each loop**

for\_each(vec.begin(), vec.end(), unary\_predicate);

**// Merges elements from aFirst to aLast and bFirst**

**// to bLast into a result and returns iterator pointing**

**// to first element of result**

OutputItr merge(InputItr1 aFirst, InputItr1 aLast,

InputItr2 bFirst, InputItr2 bLast,

OutputItr result);

**// This function is used to check whether one sorted container elements are including other sorted container elements or not. Returns true if 1st container includes 2nd container else returns false.**

includes(beg1, end1, beg2, end2);

**// set\_union/intersection/difference executes corresponding operations on 2 containers and stores into 3rd container**

set\_union(beg1, end1, beg2, end2, beg3)

set\_intersection(beg1, end1, beg2, end2, beg3)

set\_difference(beg1, end1, beg2, end2, beg3)

**// Search for 2 consecutive elements in container**

adjacent\_find( ForwardIt first, ForwardIt last );

**//Checks if list is sorted**

bool is\_sorted( ForwardIt first, ForwardIt last );

**//Assigns new\_value to all the elements in the range [first, last] that compare to old\_value**

void replace (ForwardIterator first, ForwardIterator last,const T& old\_value, const T& new\_value)

**// It checks for a given property on every element and returns true when each element in range satisfies specified property,**

bool all\_of (InputIterator first, InputIterator last, UnaryPredicate pred);

**// To rotate elements In the container**

void rotate(ForwardIterator first, ForwardIterator middle,

ForwardIterator last)

**//It returns number of occurrences of an element in a given range**

int count(Iterator first, Iterator last, T &val)

**//find\_if returns an iterator to the first element in the range [first, last] for which pred(Unary Function) returns true while find\_if\_not returns**

**iterator if pred returns false**

InputIterator find\_if(InputIterator first, InputIterator last, UnaryPredicate pred);

InputIterator find\_if\_note(InputIterator first, InputIterator last,

UnaryPredicate pred);

**// Remove a given element from container**

ForwardIterator remove (ForwardIterator first,ForwardIterator last, const T& val)

ForwardIterator remove\_if (ForwardIterator first,ForwardIterator last, UnaryPredicate pred);

**// Move elements from one container to another**

OutputIterator move (InputIterator first, InputIterator last, OutputIterator result);

The erase–remove idiom cannot be used for containers that return const\_iterator

std::remove doesn’t actually delete elements, it just shunts non-deleted elements to top, reason being that it uses ForwardIterator and container agnostic

**// Remove with condition**

std::remove\_if(v.begin(), v.end(), unary\_predicate)

**// Remove all occurrences of a given element**

std::remove( v.begin(), v.end(), val)

**Example:**

std::vector<int>::iterator low,up;

low=std::lower\_bound (v.begin(), v.end(), 6); // ^

up= std::upper\_bound (v.begin(), v.end(), 80); //

**// Sum-up all values in container. init specifies the initial value of accumulator**

**accumulate(v.begin(), v.end(), init, [](int i, int j) { return i+j; });**

**std::all\_of, std::any\_of, std::none\_of**

Apply a predicate to a set.

// are all of the elements positive?

all\_of(first, first+n, ispositive());

// is there at least one positive element?

any\_of(first, first+n, ispositive());

// are none of the elements positive?

none\_of(first, first+n, ispositive());

**copy\_n()**

copy\_n() copies one array elements to new array. This type of copy creates a deep copy of array. This function takes 3 arguments, source array name, size of array and the target array name.

// Using copy\_n() to copy contents

copy\_n(ar, 6, ar1);

**iota()**

This function is used to assign continuous values to array. This function accepts 3 arguments, the array name, size, and the starting number

int ar[6] =  {0};

// Using iota() to assign values

iota(ar, ar+6, 20);

Output: 20 21 22 23 24 25

**Notes:**

std::begin() function is introduced in C++11, so that it can work with array and Templates as well.

emplace() function copies in place while in vector, element is copied to some location and then moved

The functional stuff: bind1st, bind2nd, mem\_fun, equal\_to, etc. is pretty useful if for some

You need to specify fn\_name() only in case of function object, in cases of function and lambda expression, no need of parentheses.

Function objects in STL algorithm API:

Adaptable Binary functions:

(plus/minus/multiplies/divides/modulus)<T>(int i, int j)

Adaptable Unary functions:

(negate)<T>(int i)

enable\_if

It allows us to call a specific function overload based on 'type'. Consider below example:

template <bool, typename T = void>

struct enable\_if

{};

// Specialized one

template <typename T>

struct enable\_if<true, T> {

typedef T type;

};

And now we can do things like:

template <typename T>

void do\_stuff(typename enable\_if<std::is\_integral<T>::value, T>::type &t) {

// an implementation for integral types (int, char, unsigned, etc.)

}

template <typename T>

void do\_stuff(typename enable\_if<std::is\_class<T>::value, T>::type &t) {

// an implementation for class types

}

In C++14, it can be written as:

template <typename T>

void do\_stuff(std::enable\_if\_t<std::is\_integral<T>::value, T> &t) {}

template <typename T>

void do\_stuff(std::enable\_if\_t<std::is\_class<T>::value, T> &t) {}

A small description:

enable\_if takes 2 parameters: first one is condition, which, if evaluated to true, mark the function active with its return type specified by the second argument passed to enable\_if

It has ::type as its static const member which refers to the second argument passed to it. enable\_if is preceded by 'typename' keyword

**Concepts:**

Concept is a term that describes a named set of requirements for a type.

Simple Concepts:

|  |  |
| --- | --- |
| Assignable | copy constructor, assignment operator |
| DefaultConstructible | default constructor |
| EqualityComparable | equality and inequality operator |
| LessThanComparable | order comparison with operators <, <=, >=, and > |

A regular type is one that is a model of Assignable, DefaultConstructible, EqualityComparable, and one in which these expressions interact in the expected way. For example, after x = y, we may assume that x == y is true.

Iterator Concepts:

|  |  |  |
| --- | --- | --- |
| Concept | Refinement of | Syntactic requirements |
| InputIterator | Assignable, EqualityComparable | operator\*(), operator->(), operator++(), |
| OutputIterator | Assignable | operator\*(), operator++() ... |
| ForwardIterator | InputIterator, OutputIterator, DefaultConstructible | ... |
| BidirectionalIterator | ForwardIterator | operator--(), ... |
| RandomAccessIterator | BidirectionalIterator, LessThanComparable | operator+(), operator+=(), operator-(), operator[](), ... |

**static\_assert:**

Static assertions are a way to check if a condition is true when the code is compiled.

static\_assert() is executed at compile time. For Example:

static\_assert(sizeof(unsigned int) \* CHAR\_BIT == 32, "Sum not equal");

Advantages:

> Can be used with templates while #error cannot be used.

> Similarly, sizeof operator can be used as that is available compile time.

To check if 2 data types are equal, use --> std::is\_same<T, U>::value

When to customize iterators:

When some transformation needs to be applied or we want to skip elements.

General Notes:

> program crashes as soon as soon as return statement of h() is executed ... -> h has modified the return address.

> Russian Peasent algorithm to multiply 2 numbers -> The idea is to double the first number and halve the second number repeatedly till the second number doesn’t become 1. In the process, whenever the second number become odd, we add the first number to result (result is initialized as 0)

> for a very large image (2D matrix) the row-major traversal is better than column-major traversal because rows are contiguously allocated.