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
* arrays
* forward\_list( Introduced in C++11)

Container Adaptors : provide a different interface for sequential containers.

* queue
* priority\_queue
* stack

Associative Containers : implement sorted data structures that can be quickly searched (O(log n) complexity).

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

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

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

• 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.

**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;

}

**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

**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)

2. std::unique\_ptr

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.

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:

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.

}

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

   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 has to be vanished in 1 go.

std::make\_shared is thread-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 ate line 3. In that case the memory allocated at point 2 will be leaked since shared\_ptr constructor hasn’t been called yet, hence object not constructed. To resolve the above issues, replace the shared\_ptr constructor with make\_shared<Object>();

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.

**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, res, 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

**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.

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

**Initializer Lists:**

Initializer list is a new functionality added to C++11 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)

**Function objects** –

These are functions, function pointers and class object that defines operator()

It is of three types

Generators - f() (Functr without any param)

Unary function - f(int r) .............. It is called predicate

Binary function - f(inr , char s)

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 )

**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>

**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

const vs constexpr

const means we promise to compiler that we won’t modify the data while constexpr means the expression which can be evaluated at compile time, provided that the data we’ll feed is constant. It can be variable in case of const.