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
* 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 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.

**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. This is how deque is implemented inside.



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.

**<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’

**<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)

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)

**<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)

**<forward\_list>**

Just for efficiency of 1 word, forward\_list lacks the size() member function.

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)

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.

Note:

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

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

Complexity of forward\_list.size() is O(n) while for list is O(1)

**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)

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.

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:

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

   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 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 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 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.

> 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, 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] : 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.

**For-Each loop:**

C++11 introduced a concise form of loop statement, specially 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. Tto 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 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)

**Advantages of constructor initializer list:**

> References and const can be initialized in this.

> Prevents assignment operator to be called which unnecessary create temporary

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

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

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

> 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.

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

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

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

lock(lck1,lck2,lck3);

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.

**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};

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[](), ... |

**General Notes:**

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

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

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.

**Features introduced in C++14**

> Auto type deduction in lambda expression.

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

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

> 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.[16] If the tuple has more than one element of the type, a compile-time error results:[19]

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

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();

}