C++ Advanced

- ✓ STL Algorithms – all_of, for_each,count_if, sort
- ✓ STL Containers – Sequential, Associative, Adapter
- ✓ Move Semantics – std::move
- ✓ Lambda Function – [ ] ( ) { }
- ✓ C++ Exceptions – try catch throw

Contents

Modern C++

- Expressive
- Fully C++ - lambdas, templates, const etc.
- Readable
- Stack semantics – avoid manual memory management
- C++ and libraries – include what you use

```
std::string → #include <string>

std::shared_ptr → #incldue <memory>

std::vector → #include <vector>
```

Smart pointer

- Stack based object
- Manages memory on heap
- Frees memory when it goes out of scope

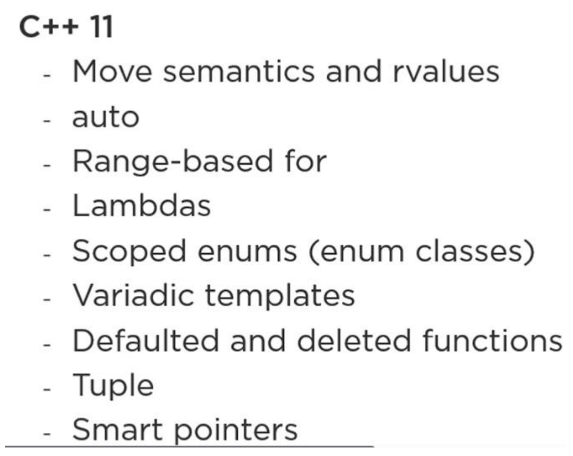
shared_ptr - Reference counted

weak_ptr - “peek” at a shared_ptr without bumping the reference count

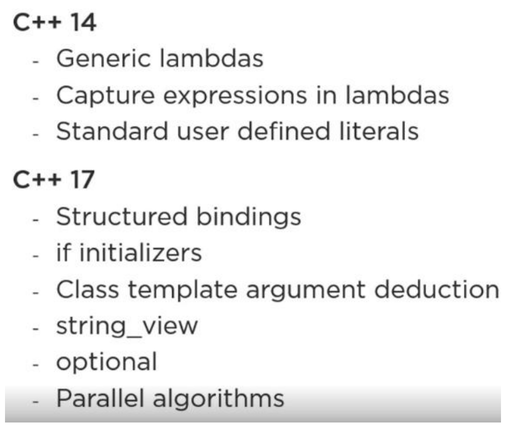
unique_ptr - Non-copyable (use std::move)

const

- A way to commit to compiler that the value won’t change
- Declaration – int const zero = 0;
- Function parameter – int taxes (int const total)
- Modifier on member function – int GetName( ) const;



C++ 11
- Move semantics and rvalues
- auto
- Range-based for
- Lambdas
- Scoped enums (enum classes)
- Variadic templates
- Defaulted and deleted functions
- Tuple
- Smart pointers



C++ 14
- Generic lambdas
- Capture expressions in lambdas
- Standard user defined literals

C++ 17
- Structured bindings
- if initializers
- Class template argument deduction
- string_view
- optional
- Parallel algorithms

STL Algorithms

- Works on all STL collections
- uses iterators
- sorting searching algorithms
- InputIt -> input iterator

all_of any_of none_of	for_each for_each_n	count count_if	copy copy_if

Fill vector with 0 to 4

```
for ( int i=0; i<5; i++)
        v.push_back(i);
```

```
int i = 0;
std::generate_n(std::back_inserter(v), 5 , [&] () { return i++; } ) ;
```

Sum of elements in vector

```
int total = 0;
for(int index = 0; index < 5; index++)
        total += v[index];
```

```
int total = 0;
for ( int elem : v )
        total += elem;
```

```
int total = std::accumulate(begin(v), end(v), 0 );
```

Count number of 3's

```
int count = 0;

for (unsigned int i=0; i<v.size(); i++)
        if( v[i] == 3)
                count++;
```

```
int count = 0;
for ( auto it = begin(v); it!= end(v); it++)
        if( *it == 3 )
                count++;
```

```
int count = std::count(begin(v), end(v), 3);
```

Remove the 3's

```
auto v2 = v;
for( unsigned int index = 0; index < v2.size(); index++)
        if(v2[index] == 3 )
                v2.erase(v2.begin() + index );
```

```
auto v3 = v ;
for (auto it = begin(v3); it != end(v3); it++)
        if ( *it == 3 ) v3.erase(*it);
```
// wrong – will fail. When we delete through an iterator, the iterator is invalidated

```
auto v4 = v;
auto endv4 = std::remove_if(begin(v4), end(v4), [] (int elem){ return (elem == 3); } ) ;
v4.erase(endv4,end(v4);
                or
v4.erase(std::remove_if(begin(v4),end(v4),[](int elem){return (elem==3);}),end(v4));
```

```
sort(begin(v4), end(v4)) // c++ 20 : sort(v4)

bool allpositive = std::all_of(begin(v4),end(v4),[] (int elem){return elem >=0 ;});

string s { "Hello I am a sentence"};

auto letter = find ( begin(s), end(s), 'a'); // find first a

auto caps = std::count_if(begin(s), end(s), [](char c){return (c!=' ') && (toupper ( c ) == c ); });
```

STL Containers

Vector

- Grows itself when new item added
- Can traverse with an iterator or random access with [ ]
- Cleans itself when goes out of scope.
- While Resizing, does copying of elements. Faster than normal copy.
- Uses move semantics.
- Getting to particular element is fast.
- Consecutive memory location.
- Iterator++ is next memory location calculation.
- Elements inside vector are kept on heap.

```
vector<int> v { 0 , 1, 2 };
v.push_back(-2)
int i = v[2];
v[2] = 5;
for (int i : v )
      cout << i;
```

Array - std::array

List - Implements linked list

- Less copying. Insertion in middle does not need to move other items.
- More expensive on traversal. Iterator ++ is an indirection.
- Never assume list is faster. Test and decide

Associative Containers

- map
- multimap
- unordered_ map
- unordered_multimap

- set
- multiset
- unordered_set
- unordered_multiset

Sequential Containers

vector	array
list	forward_list
dequeuer	span

Container Adapters

- stack
- queue
- priority_queue

Common Member Functions:

- size
- capacity
- clear
- insert

begin( ) and end ( )

- For begin and end of container
- Can be used for C-style array also

Move Semantics

Some objects have a pointer to data somewhere else.

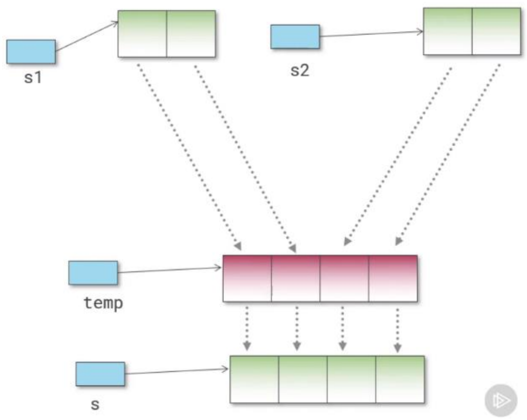
Copying this data to another object take time.

If we don't need the original object anymore, we can move it instead of copying.

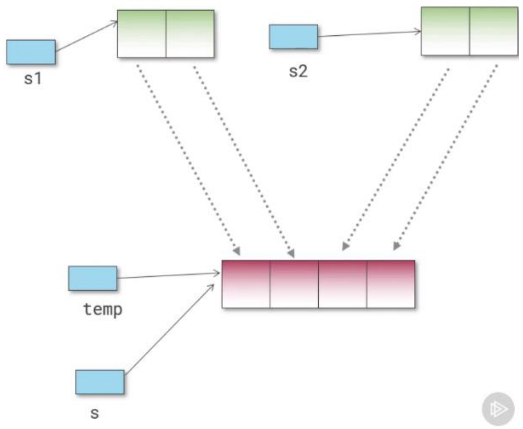
Temporaries:

string s = s1 + s2 + s3 + s4;

Temporaries like (s3 + s4) have no purpose later. Moving is huge **speedup**.



string s = s1 + s2;



R-value Reference

There is a class, called : Resource. A function can take Resource parameter multiple ways:

1. Resource x – An instance of Resource
2. Resource *px – a pointer to instance of Resource
3. Resource &rx – a reference to instance of Resource
4. Resource && rrx – **an rvalue reference** to disappearing instance

x = 3
x = a + b

x is l-value.

a+b or 3 are r-value.

- ✓ l-value is some address, some memory location
- ✓ r-value is something temporary

```
Move Assignment Operator

Resource& operator= (Resource&& r);

Resource& Resource::operator= (Resource&& r){
        if( this != &r ){
                name = std::move(r.name);
                r.name.clear();
        }
        return *this
}
```

```
Move Constructor

Resource ( Resource&& r)

Resource::Resource(Resource&& r) :
                name(std::move(r.name)){}
```

Rules

- **Pass by value** – If a temporary is passed to function, it might be moved, not copied.
- **Return by value** - The local variable return is about to go out of scope – compiler will move it.
  [Return value optimization. Guaranteed copy Elision.]
- **Is vector slow?**
  – When copying becomes moving, the heuristics change
- **Is it inefficient to build string from many little pieces?**
  – Not if you move the pieces as you go

- ✓ std::move is cast
- ✓ std::move does not move anything

name = std::move (r.name);

- ✓ It causes the compiler to choose the move constructor or move assignment operator. They might move something.

Lambda

- Lambda is an expression – An expression that represents doing something.
- Lambda expression holds code.
- For generic Work. For functional style. For concurrency. For readability.
- Eliminates tiny function

Lambda is a way to pass code off to a function or a way to store some code in a variable and then use it again later.

Tiny Functions

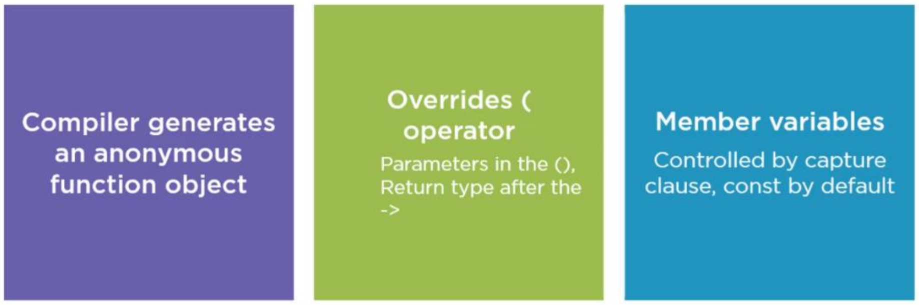
```
auto isOdd = [ ](int candidate){ return candidate % 2 != 0 ; };

bool is3Odd = isOdd(3);

bool is4Odd = isOdd(4);

vector nums {2,3,4,-1,1};

int odds = std::count_if ( begin(nums), end(nums), isOdd)

int odds = std::count_if ( begin(nums), end(nums), [ ](int candidate){ return candidate % 2 != 0 ;})
```

[ ] ( ) { } // a valid lambda

- Capture clause [ ]
- Parameters ( )
- Body { }

How it is implemented

- Compiler generates class and instance of that class using what we put into [ ] ( ) { }.
- It's a class. It has member functions and member variables. One member function is **overloaded function call operator ( )**
- Member variables are **const** by default.



Compiler generates an anonymous function object

Overrides (operator
Parameters in the (), Return type after the ->

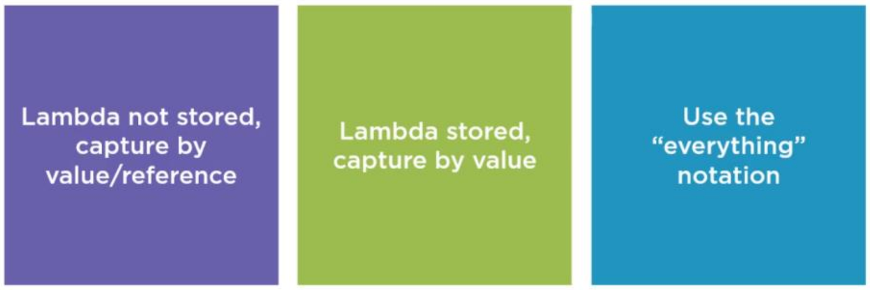
Member variables
Controlled by capture clause, const by default

The Capture

Empty [ ] – captures nothing, use only function parameters.
We can put some variables from calling scope into capture [ ]. Lambda will have access to those variables.

- **[x, y]** – capture x and y by value. Copies are made. Lambda can be used even when x and y have gone out of scope.
- **[&x, &y]** – capture x and y by reference. No copies, changes affect the originals. Dangling references may be an issue
- **[x = a+1, y = std::move(b)]** – Alias or move capture.
- **[=]** – Copy (by value) all whatever used in lambda.
- **[&]** – Copy (by reference) all whatever used in lambda.
- **Mutable** – Allows to change values captured by reference.

How To Capture



Lambda not stored, capture by value/reference

Lambda stored, capture by value

Use the "everything" notation

```
int x = 3;
int y = 7;
string message = "elements between ";
message += std::to_string(x) + " and " ;
message += std::to_string(y) + " inclusive: ";
for_each(begin(nums),end(nums),
                [x,y,&message](int n){
                if (n >= x && x <= y)
                        message += " " + std::to_string(n);
        }
);
cout << message << endl;
```

```
x = y = 0;
for_each(begin(nums),end(nums),
               [&,x](int element) mutable
{
               x += element;
               y += element;
       }
);
cout << "x = " << x << endl;
cout << "y = " << y << endl;
```

```
message = "";
auto pResource = make_unique<Resource>(", ");

for_each(begin(nums), end(nums),
       [=,&message, p =std::move(pResource)](int n){
       if ( x<=n && n <= y)
              message += std::to_string(n) + p->GetName();
       }
);
```

Note: After moving unique_pointer to lambda, **it becomes empty**. Should not be used afterwards.

Return Type:

- Lambda can return value
- Only a return statement in the lambda
- Return type is inferred by compiler
- If compiler cannot infer, we should specify return type
- Example: [ ] (int n) -> double { }

Parameters: Generally imposed by the place we use it.

Lambda Usability:

- Lambda can be used anywhere function objects (functors) used.
- Sometimes we can use a function pointer.
- Lambdas keep the code where it is used.

Exceptions

Expected Errors – Test for it. Deal with it right there.

Unexpected errors -

- Code that finds it, cannot deal with it. E.g. business layer cannot give message to UI.
- Have function return an indication of trouble
  - Function already returns something?
  - Function cannot return a value? (e.g. constructor)
  - Developer forgets to check the return value?
  - Try, throw, catch

```
try {
// risky stuff
}
catch( out_of_range &oor){
// Handle
}
catch(exception &e){
// Handle
}
```

Exceptions

- Transfer flow of execution
- Deal with trouble as close to the problem as possible
- Developer cannot forget to check return value
- Need to know about stack unwinding
- Wrap code (that may cause problem) in a try block – as small as possible
- Add one or more catch block to handle the exception
- More specific exception catch first, more generics to last
- Catch exceptions by reference – to retain type. catch (Exception &e)
- Else slicing will happen
- Good for catching derived exception
- There is no finally – clean up code is in destructors
- Destructor runs no matter how control leaves the block
- Can throw anything: string, int, object. e.g. : throw "Exception";

std :: exception

- Exception description: e.what()
- marker classes

<u>logic_error</u>		
domain_error, length_error	invalid_argument, out_of_range	runtime_error

```
throw invalid_argument("Number can not be zero");
```

```
vector<int> v;

v.push_back(1);
try{
	int j = v.at(99);
}
catch(out_of_range &e){
	cout << "Out of range" ;
}
catch(exception &e){
	cout << e.what();
}
```

Unwinding the stack

- If exception is thrown in a try,
  - everything in local scope of try, goes out of scope.
  - Destructors run.
  - Control goes to catch block.
- If not in a try
  - everything local to the function, goes out of scope.
  - Control returns to where the function was called from
- Get all the way out of main( ), the user gets a dialog

Exception and Moving

- If a move operation throws, the enclosing operation can not be rolled back.
- Some moving operation in std:: will only call noexcept functions

  Move ctor, move op=, swap

- If move operation are not noexcept, we get a copy instead.
- Mark these noexcept if you can

Exception has a performance cost

- noexcept - Mark function noexcept if no chance of exception.
- Still it throws exception? → program terminates, no stack unwinding.