UNIVERSITY
OF WOLLONGONG
AUSTRALIA

CSCI251
Advanced Programming

Moving and Advanced Function

Outline

Object Oriented Programming in C++

Move semantics

1. **L-value and R-value**
2. Move constructor
3. Move assignment
4. Move function

RAII

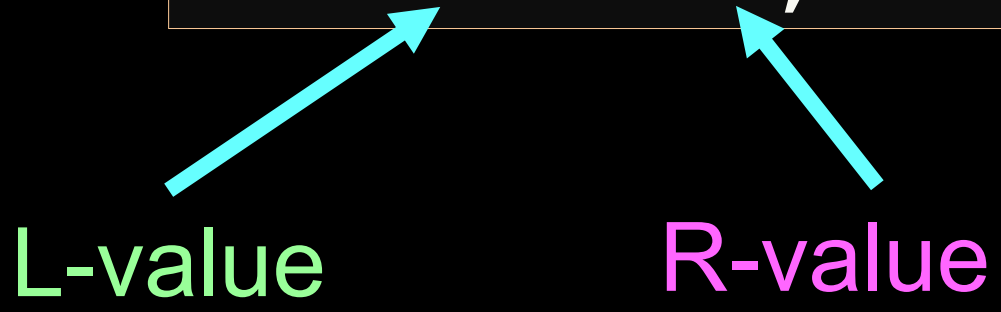
Lambda expression


L-Value and R-Value

L-value is variable that points to a specific memory location. On the left of the assignment.

R-value is a value that is temporary and short lived

```
int i = 6;
```



Source:
https://www.youtube.com/watch?v=PNRju6_yn3o
ttps://www.internalpointers.com/post/understanding-meaning-lvalues-and-rvalues-c

L-value and Rvalue example

Are the assignments below legal?

```
int a=42;
int b=43;

a=b; b=a;

int c=a+b;
a +b =42;

int *p=&i;
int *p1=&43;
```

“Reference”, do you recall?

```
1    int i = 6;        // L-value is i and R-value is 6
2    int & ref = i;    // reference to i (an alias)
3    int && refref = i; //
```

```
int i

L-value is i and R-value is 6

an rvalue reference cannot be bound to an lvalue C/C++(1768)

View Problem (Alt+F8)    No quick fixes available
```

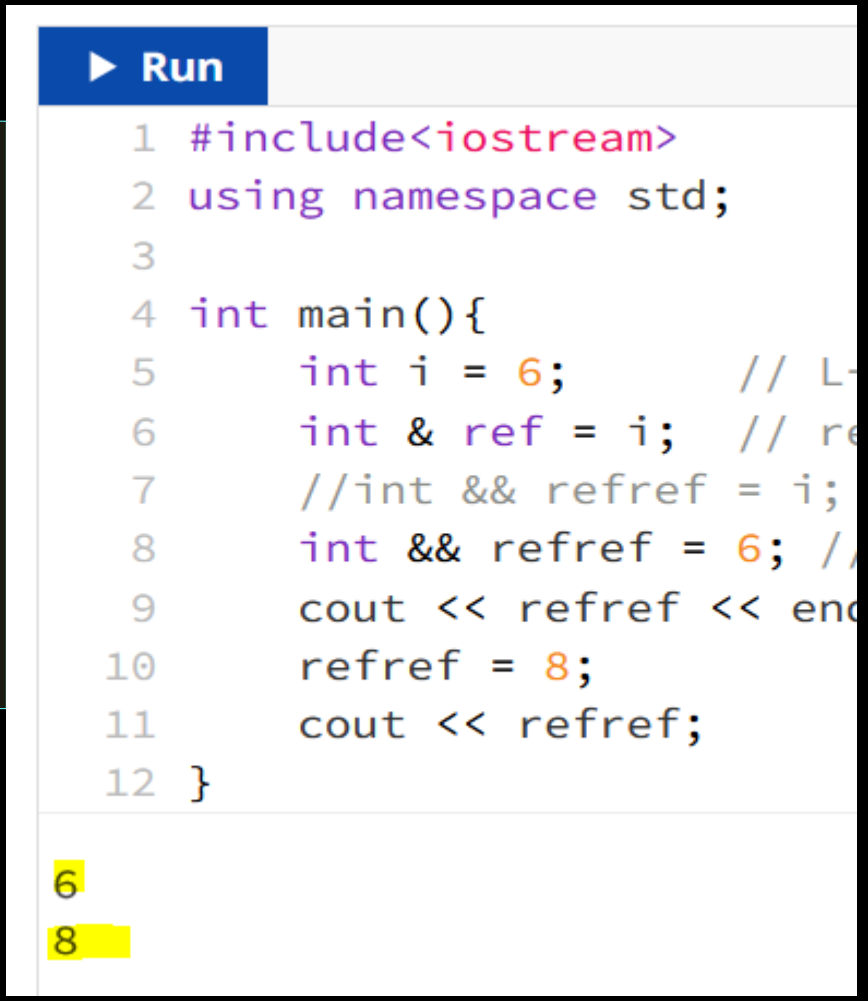
In C++, && sign means you want an R-value reference

R-value demo

```cpp
#include<iostream>
using namespace std;

int main(){
    int i = 6;       // L-value is i and R-value is 6 (copy)
    int & ref = i;  // reference to i (an alias)
    int && refref = i; //
    int && refref = 6; // NO PROBLEM
    refref = 8;
    cout << refref;
}
```

&& assign to R-value reference (line 8)
You can think about it as an alias of R-value



Inefficiency

```
vector<int> getReverse(const vector<int>& original) {
        vector <int> reversed;
        for (int i=original.size()-1; i >= 0; ++i) {
                reversed. push_back(original.at(i));
        }
        return reversed;
}

Somewhere in main...
vector<int> k = {2, 4, 6, 8};
vector<int> r = getReverse(k);
```

Before C++11

- Vector is constructed in the function and returned in a temporary object.
- The returned data is **copied** to the object getting the assignment.
- Then the returned object is destructed.

```
vector<int> k = {2, 4, 6, 8};
vector<int> r = getReverse(k); // copy assignment
```

After C++11

- Instead of copying...
- **Pilfer / steal** the dynamic parts from the temporary object and use in the object getting the assignment.
- Instead of making a temporary copy then destructing the temporary,
  - Use what is already there and would be destructed anyway.
- So if we had a vector with 1,000,000 elements...
  - instead of copying 1,000,000 elements (time consuming)
  - Point to those existing 1,000,000 elements and unpoint the temporary object.

copying one pointer is much faster than copying 1,000,000 elements!

Move Constructor Declaration

MyClass (MyClass&& source) ;

&& indicates that it is an Rvalue, i.e., a temporary object

Move Constructor Definition

```
MyClass: :MyClass (MyClass&& source) {
      // pilfer dynamic resources from source
      // set source dynamic resources to nullptr
}
```

Move Assignment Declaration

MyClass& operator=(MyClass&& source);

&& indicates that it is an Rvalue, i.e., a temporary object

Move Assignment Definition

```
MyClass& operator=(MyClass&& source) {
    if (this != &source) {
        // delete old data from this / Ths
        // piller resources from source
        // set pointers in source to nullptr
    }
    return *this;
}
```

This should look familiar! Recall copy assignment.

Recall

Move (~~Copy~~) Assignment

1. Delete old data
2. ~~Allocate new memory~~
3. ~~Copy data from source~~
4. Pilfer resources from source
5. Set pointers in source to nullptr

Move (~~Copy~~) Constructor

1. ~~Allocate new memory~~
2. ~~Copy data from source~~
3. Pilfer resources from source
4. Set pointers in source to nullptr

Move function (std::move)

- It makes use of a standard library function move that converts, or casts, from an lvalue to an rvalue reference
- The function move is part of the standard namespace

```cpp
#include<iostream>
using namespace std;

int main(){
    int i = 6;       // L-value is i and R-value is 6
    int & ref = i;   // reference to i (an alias)
    //int && refref = i; //
    //int && refref = 6; // NO PROBLEM
    int && refref = move(i); // NO PROBLEM
    cout << refref << endl;
    refref = 8;
    cout << refref;
}
```

```
6
8
```

Move function or move constructor

- Move is like we change the ownership. Because it creates the alias of R-value
- Move can be significantly more efficient than a copy and delete and it’s a major reason why C++11 STL containers are improvements over C++98/C++03 ones.

Why we use move in C++

- Copy is expensive. It uses the memory
- Move is not a copy. It assigns to the R-value reference. It saves the use of memory.

https://www.artima.com/articles/a-brief-introduction-to-rvalue-references
https://www.srcmake.com/home/move-semantics

An example of COPY constructor

```
Calling Default constructor
Calling Copy constructor
Calling Destructor
Calling Destructor
```

```cpp
#include <iostream>
#include <vector>
using namespace std;

class A{
  int *ptr;
public:
  A(){
    // Default constructor
    cout << "Calling Default constructor\n";
    ptr = new int ;
  }
  A (const A & obj){
    // Copy Constructor
    // copy of object is created
    ptr = new int;
    // Deep copying
    cout << "Calling Copy constructor
  }\n";
  ~A(){
    // Destructor
    cout << "Calling Destructor\n";
    delete ptr;
  }
};

int main() {
  vector <A> vec;
  vec.push_back(A());
  return 0;
}
```

An example of MOVE constructor

```
Calling Default constructor
Calling Move constructor
Calling Destructor
Calling Destructor
```

```cpp
#include <iostream>
#include <vector>
using namespace std;
class A{
  int *ptr;
public:
  A(){
    // Default constructor
    cout << "Calling Default constructor\n";
    ptr = new int ;
  }
  A( const A & obj){
    // Copy Constructor
    // copy of object is created
    this->ptr = new int;
    // Deep copying
    cout << "Calling Copy constructor\n";
  }
  A ( A && obj){
    // Move constructor
    // It will simply shift the resources,
    // without creating a copy.
     cout << "Calling Move constructor\n";
    this->ptr = obj.ptr;
    obj.ptr = nullptr;
  }
  ~A(){
    // Destructor
    cout << "Calling Destructor\n";
    delete ptr;
  }
};
int main() {
  vector <A> vec;
  vec.push_back(A());
  return 0;
}
```

Practice 1


RAII

RAII

- Resource Acquisition Is Initialization or RAII, is a C++ programming technique 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.

http://en.cppreference.com/w/cpp/language/raii
https://www.srcmake.com/home/raii

RAII

- So?
  - Binding the resource to the lifetime of the object before use means that the resources are tidied up when they go out of scope.
  - The resource is freed up correctly when the object is destroyed.
- Memory leaks were probably the earlier instance we looked at in wanting to make sure we appropriately tidy up.
- See https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#e6-use-raii-to-prevent-leaks

RAII

- Problem: Memory leaks occur when heap allocated (new keyword) variables/objects don't get deleted because of unexpected problems. See the code on the right box.

```
Allocated a new variable on the heap.
Program finished.
```

```cpp
#include <iostream>
#include <stdexcept>
using namespace std;

void MemoryLeak()
    {
    int* i = new int(42);
    cout << "Allocated a new variable on the heap." << endl;
    // if an exception occurs then
    throw std::invalid_argument("This function ends on this line.");
    cout << "We never get to this line of code!" << endl;
    // The 'i' int is still on the heap, and won't be removed
    // from RAM. This is a memory leak.
    delete i;
    }
int main()
    {
    // Call the MemoryLeak function. It can throw exceptions, so put
 it in a try catch.
    try
        {
        MemoryLeak();
        }
    catch(const std::invalid_argument& e) { }
    cout << "Program finished." << endl;
    return 0;
    }
```

RAII

- Solution: Encapsulate variables in classes, and design the destructors to deallocate the resource from memory when the object itself is deleted.
- The best way to make this happen is with smart pointers.
  - Unique Pointers
  - Shared Pointers

Unique Pointers

- Unique pointers offer good RAII technique for objects you only want to have one pointer to. (Meaning, only one pointer is allowed access to the variable/object.)

Unique Pointers

```
#include <iostream>
#include <memory>
using namespace std;

class srcPizza
    {
    public:
        // Constructor.
        srcPizza()
            { cout << "Full pizza. :) (Constructor)\
n"; }
        // Destructor
        ~srcPizza()
            { cout << "Pizza is gone. :( (Destructor)\
n"; }
    };
```

```
Program started.
42
Full pizza. :) (Constructor)
Pizza is gone. :( (Destructor)
Program finished.
```

```
void DoStuff()
    {
    // Declare a unique pointer like this:
    unique_ptr<int> srcUP(new int(42));
    // The constructor takes a pointer to the object is.

    // You can dereference the unique pointer like a normal poin
ter.
    cout << *srcUP << endl;



    // The destructor of an object is called when the unique poi
nter is deleted.
    unique_ptr<srcPizza> srcUP4(new srcPizza());
    } // srcUP4 gets deleted here, so the destructor of the srcP
izza is called.

int main()
    {
    cout << "Program started." << endl;
    DoStuff();
    cout << "Program finished." << endl;
    return 0;
    }
```

Shared Pointers

- Shared Pointer is the same as unique pointers, except there can be multiple pointers to one variable/object.

Shared Pointers

```
#include <iostream>
#include <memory>
using namespace std;

class srcPizza
    {
    public:
        // Constructor.
        srcPizza()
            { cout << "Full pizza. :) (Constructor)\
n"; }

        // Destructor
        ~srcPizza()
            { cout << "Pizza is gone. :( (Destructor)\
n"; }
    };
```

```
Program started.
42
42
Full pizza. :) (Constructor)
Going out of scope.
Pizza is gone. :( (Destructor)
Program finished.
```

```
void DoStuff()
    {
    // Declare a shared pointer like this:
    shared_ptr<int> srcSP(new int(42));
    // The constructor takes a pointer to the object is.

    // You can make another pointer point to the same thing.
    shared_ptr<int> srcSP2 = srcSP;

    // You can dereference the shared pointer like a normal pointer
.
    cout << *srcSP << endl;
    cout << *srcSP2 << endl;

    // The destructor of an object is called when the shared pointe
r goes out of scope.
    shared_ptr<srcPizza> srcSP5(new srcPizza());
    cout << "Going out of scope.\n";
    } // srcSP5 gets deleted here, so the destructor of the srcPizz
a is called.
int main()
    {
    cout << "Program started." << endl;
    DoStuff();
    cout << "Program finished." << endl;
    return 0;
    }
```

Practice 2


Pairs

Pairs

- Pairs are just mini containers provided by the standard library to group two variables (of any type) together.
- You can initialize a pair with the make_pair function, and you can access the elements by calling the variable name's first and second properties.

```
#include <iostream>
#include <utility>

int main()
    {
   std::pair<int, std::string> p = std::make_pair(5, "hey")
;
    std::cout << p.first << " " << p.second << std::endl;
  return 0;
   }
```


Lambda Expressions

Lambda Expressions

- A lambda expression is way to write anonymous functions

Representation []{}:

{}: function body
(): parameters
[]: capture list

Lambda Expressions []{}
function body {}

passing a function as parameter

```
std::function<return_type(arg1_type, arg2-type...)> obj_name

   //  This object can be use to call the function as below
   return_type catch_variable = obj_name(arg1, arg2);
```

function body {} illustration

```
void print(function<bool()> func){
    string str = func() ? "true" : "false";
    cout << str << "\n";
}
int main(){
    auto isFiveGreaterThanThree = [](){ return 5 > 3; };
    print(isFiveGreaterThanThree) ;
    return 0;
}
```

```
void print(function<bool()> func){
    string str = func() ? "true" : "false";
    cout << str << "\n";
}
int main(){
    //auto isFiveGreaterThanThree = [](){ return 5 > 3; };
    print([](){ return 5 > 3; }) ;
    return 0;
}
```

Lambda Expressions [](){} parameters ()

passing a function as parameter

```
std::function<return_type(arg1_type, arg2-type...)> obj_name

   //  This object can be use to call the function as below
   return_type catch_variable = obj_name(arg1, arg2);
```

parameters {} illustration

```
void print(int param, function<bool(int)> func){
    string str = func(param) ? "true" : "false";
    cout << str << "\n";
}
int main(){
    auto isFiveGreaterThanThree = [](int num){ return num > 3; };
    print(5,isFiveGreaterThanThree) ;
    return 0;
}
```

Lambda Expressions []{}
Capture list []

passing a function as parameter

```
std::function<return_type(arg1_type, arg2-type...)> obj_name

   //  This object can be use to call the function as below
   return_type catch_variable = obj_name(arg1, arg2);
```

capture list [] illustration

```
void print(int param, function<bool(int)> func){
    string str = func(param) ? "true" : "false";
    cout << str << "\n";
}
int main(){
    int target=3;
    auto isFiveGreaterThanThree = [target](int num){ cout<<"target:"<<target<<endl; return num > target; };
    target=6;
    print(5,isFiveGreaterThanThree) ;
    return 0;
}
```

Lambda Expressions []{}
Capture list []

[x, &y] : pass x by value, y by reference
[&] : pass all needed externals by reference
[=] : pass all needed external by value
[&, x]: pass x by value and others by reference
[=,&z]: pass z by reference and others by value

Lambda Expressions []()->return-type{}

optional : ->return type

```cpp
#include <iostream>

int main()
{
    auto f = []() -> double {
        if (1 > 2)
            return 1;
        else
            return 2.0;
    };
    std::cout << f() << std::endl;
}
```

Lambda Expressions

- A lambda expression is literally just a different way of writing a function. For example, a lambda expression that checks if a number is greater than 2 is the following:

```
auto ckLambda = [](int i)
    {
    return i > 2;
    };
```

Lambda Expressions

- The lambda expression is named "ckLambda", it's type is "auto", and it's equal to that thing on the right.
- The square brackets [] capture any variables you need that exist outside of the lambda expression.
- The parenthesis accept whatever input the lambda expression is given, like a normal function.
- The inside of the lambda function does some work, and possibly returns a value.

Lambda Expressions

- We need to pass lambda expressions to certain STL algorithms to do certain work. For example, to check count how many elements in a vector are greater than 2, we could use a lambda expression and the "count_if" algorithm.

```
vector<int> v{ 1, 2, 3, 4, 5 };

auto lambda = [](int i) { return i > 2; };

int count = count_if(v.begin(), v.end(), lambda
)
```

Lambda Expressions

- For each element of "v", "count_if" will use call the lambda expression on that element. If the lambda expression returns "true", then count_if will add one to it's counter, and give the final answer to the "count" variable.

- It's much simpler to have those lines of code than it would be to write our own loops and other variables. But what are those "v.begin()" and "v.end()" things?

Practice 3


Iterators

Iterators

- Iterators are just special variables that point to a certain element in a container. They seem useless for vectors since vectors can be accessed by index, but other container types (like stacks) don't have indices, and so iterators are used to indicate the place in the container.
- The type of an iterator looks something like this:

```
vector<int> v{ 1, 2, 3, 4, 5 };

std::vector<int>::iterator it = v.begin();
```

Iterators

- An iterator that's for a vector<int> container named "it" is equal to v's "begin()".
- A vector's "begin()" will point to the first element in the vector (in this case, the element 1 (at index 0)), and a vector's "end()" will point to the element after the last element in a vector (in this case, an invisible element after 5, which would be at index 5).
- You can increment an iterator (move the position to the next element) with a simple ++ operation. it++

Iterators

- You can also simply add to a beginning iterator to get the position you want too.

```
std::vector<int>::iterator it = srcVector.begin() + 4;
// Accesses the 5th element in the container.
```

- You can also subtract iterators (which can help you get a position's index).
- iterators just point to the elements in a container.

```
std::vector<int>::iterator  index = it - srcVector.begin();
auto index = it – srcVector.begin();
```