Programming in Modern C++

Module M50: C++11 and beyond: General Features: Part 5: Rvalue and Move/2

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Understood the difference between Copying and Moving
- Understood the difference between Lvalue and Rvalue
- Learnt the advantages of Move in C++ using
  - Rvalue Reference
  - Move Semantics
  - Copy / Move Constructor / Assignment
  - Implementation of Move Semantics

Module Objectives

- To learn to implement move semantics in user-defined classes
  - Challenges
  - Workaround using `std::move`
- To learn the use and implementation of `std::move`
- To put all the pieces together in a project to code move-enabled UDTs

Module Outline

Recap of Copy vs. Move and related Concepts

Recap of Copy vs. Move and related Concepts

Sources:

- **Module 49**: *C++11 and beyond: General Features: Part 4: Rvalue and Move/1*
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++

Copying vs. Moving: Recap (Module 49)

- C++ has always supported copying object state:
  - *Copy* constructors, *Copy* assignment operators
- C++11 adds support for requests to *Move* object state for performance improvement. Examples show the benefits of Move in several contexts including:
  - Return by value from functions
  - Appending to a full vector
  - Swapping two variables
  - Choice of Swallow Copy in place of Deep Copy when possible
  - ...
- C++11 adds the following for related optimization
  - Rvalue Reference
  - Move Semantics through *Move* constructors, *Move* assignment operators

Lvalues, Rvalues and Rvalue References: Recap (Module 49)

- **Lvalues** *are generally things we can take the address of*:
  - In C, Expressions on *left-hand-side (LHS)* of an assignment
  - Named objects - variables
  - Legal to apply address of (&) operator
  - Lvalue references (`T&`)
- **Rvalues** *are generally things we cannot take the address of*:
  - In C, Expressions on *right-hand-side (RHS)* of an assignment
  - Unnamed (temporary) objects - expressions, return by value from functions, etc.
  - Rvalue references (`T&&`) identify objects that may be moved from
- Important for overloading resolution using Lvalues / Rvalues
- Lvalues may bind to *lvalue references*
- Rvalues may bind to *lvalue references to* `const`
- Rvalues may bind to *rvalue references to non-*`const`
- Lvalues may *not* bind to *rvalue references*

Move Semantics: How to code?

Move Semantics: How to code?

Sources:

- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++

Move Semantics: How to code?

- Move operations take source's value, may change the source, but leave the source in valid state:

```
class MyResource { // Representative resource class with move support
    char *str = nullptr; // Resource pointer
public:
    // Move: Constructs by moving resource from rvalue reference source (rr)
    MyResource(MyResource&& rr) noexcept : str(rr.str) // Take the value of source rr
    { rr.str = nullptr; } // Changed but valid state set for the source

    // Move: Assigns by moving resource from rvalue reference source (rr)
    MyResource& operator=(MyResource&& rr) noexcept {
        delete [] str;      // Release the current value
        str = rr.str;       // Take the value of source rr
        rr.str = nullptr;   // Changed but Valid state set for the source
        return *this;       // Return the assigned object
    }
...
```

- Easy for built-in types like pointers above. Gets tricky for UDTs

Move Semantics: How to code?

- Like Copy Assignment, MyResource's move operator= may fail for move-to-self:

```
MyResource r;
r = std::move(r); // undefined behavior! std::move changes r to an rvalue reference
...               // std::move will be discussed later in this module

MyResource *p1, *p2;
...
*p1 = std::move(*p2); // undefined if p1 == p2
```

- Fix is simple like the guard used in copy assignment:

```
MyResource& MyResource&::operator=(MyResource&& rr) noexcept {
    if (this != &rr) {
        ...    // Do the move as above
    }
    // else assert(this != &rr); // un-comment to disallow move-to-self
    return *this;
}
```

Move Semantics: How to code?

- Next let us attempt to build a *move-enabled* MyClass that use an object of MyResource as a resource data member:

```
class MyClass { MyResource mRrc; // Resource object
public:
    MyClass(MyClass&& c) noexcept : // Move Constructor
        mRrc(c.mRrc) // Compiles, but actually copies using MyResource(MyResource&)
    { ... }
    MyClass& operator=(MyClass&& c) noexcept { // Move Assignment
        if (this != &c)
        { mRrc = c.mRrc; } // Compiles, but actually copies using
        return *this;      // MyResource::operator=(MyResource&)
    }
    ...
};
```

- c.mRrc is an lvalue, because it has a name
  - Lvalue-ness / Rvalue-ness orthogonal to type!
    - ints can be lvalues or rvalues, and rvalue references can, too.
  - mRrc initialized by MyResource's *copy* constructor / assignment

Move Semantics: How to code?

- Next let us try to extend the design on an inheritance hierarchy where MyClass ISA MyClassBase

```
class MyClassBase { public:
    MyClassBase(const MyClassBase&);      // Copy Constructor
    MyClassBase(MyClassBase&&) noexcept; // Move Constructor
    ...
};

class MyClass: public MyClassBase { public:
    MyClass(MyClass&& c) noexcept // Move Constructor
    : MyClassBase(c) // Compiles, but actually copies using MyClassBase(MyClassBase&)
    { ... }
    ...
};
```

- c is an lvalue, because it has a name
  - Its declaration as MyClass&& not relevant!
- Similar issue will happen for MyClass::operator=(MyClass&&)

Move Semantics: How to code?

- **How to solve the problems in the coding of move semantics for UDTs?**
- In general, in a UDT, there are two kinds of constituent objects:
  - Data Member like `c.mRrc` `// MyClass c;`
  - Base Class part like `(MyClassBase&)c` `// MyClass: public MyClassBase`
- While we need to copy or move, we use the constructor or assignment operators of the underlying classes:
  - Construction: `mRrc(c.mRrc)` and `MyClassBase(c)`
  - Assignment: `mRrc = c.mRrc` and `MyClassBase::operator=(c)`
- For copy, we use the copy constructor / assignment operator
- For move, we need to use the move constructor / assignment operator to optimize resource handling
  - This means for the above four instances of the respective operators, the sources (`c.mRrc` or `c`) must be rvalues. But they are available by name as lvalues
- Hence, **we need a mechanism to convert an lvalue to an rvalue**
- `std::move` in `<utility>` provides for this

std::move

Sources:

- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Scott Meyers on C++
- Quick Q: What's the difference between std::move and std::forward?, isocpp.org
- On the Superfluousness of std::move – Scott Meyers, isocpp.org, 2012
- std::move, cppreference.com

Explicit Move Requests

- To request a move on an lvalue, use `std::move` from `<utility>`:

```
class MyClassBase { ... };
class MyClass: public MyClassBase { public:
    MyClass(MyClass&& c) noexcept : // Move Constructor
        MyClassBase(std::move(c)), // Request Move for base class part
        mRrc(std::move(c.mRrc))    // Request Move for data member
    { ... }
    MyClass& operator=(MyClass&& c) noexcept { // Move Assignment
        if (this != &c) {
            MyClassBase::operator=(std::move(c)); // Request Move for base class part
            mRrc = std::move(c.mRrc);             // Request Move for data member
        }
        return *this;
    }
    ...
};
```

- `std::move` turns lvalues into rvalues
  - The overloading rules do the rest

Explicit Move Requests

- std::move uses *implicit type deduction*
  Consider:

```
template<typename It>
void someAlgorithm(It begin, It end) {
    // permit move from *begin to temp

    // static_cast version
    auto temp1 =  static_cast<typename std::iterator_traits<It>::value_type&&>(*begin);

    // C-style cast version
    auto temp2 = (typename std::iterator_traits<It>::value_type&&)*begin;

    // std::move version
    auto temp3 = std::move(*begin);
    ...
```

- Great convenience by using std::move

Reference Collapsing in Templates

- In C++03, given

```
template<typename T> void f(T& param);
int x;
f<int&>(x);                    // T is int&
```

- `f` is initially instantiated as

```
void f(int& & param);          // reference to reference
```

- C++03's reference-collapsing rule says
  - `T& & => T&`
- So, after reference collapsing, `f`'s instantiation is actually: `void f(int& param);`
- C++11's rules take rvalue references into account:
  - `T& & => T&     // from C++03`
  - `T&& & => T&    // new for C++11`
  - `T& && => T&    // new for C++11`
  - `T&& && => T&& // new for C++11`
- Summary:
  - *Reference collapsing involving a* `&` *is always* `T&`
  - *Reference collapsing involving only* `&&` *is* `T&&`

std::move: Return Type

- To guarantee an rvalue return type, std::move does this:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(MagicReferenceType obj) noexcept {
    return obj;
}
```

  - Recall that a T& return type would be an lvalue!

- Hence:

```
int x;
std::move<int&>(x); // calls remove_reference<int&>::type&& std::move(/*...*/)
                    // => int&& std::move(/*...*/)
```

  - Without std::remove_reference, move<int&> would return int&

std::move: Parameter Type

- Must be a non-const reference, because we want to move its value
- An lvalue reference does not work, because rvalues cannot bind to them:

```
TVec createTVec();                        // as before
TVec&& std::move(TVec& obj) noexcept;     // possible move instantiation
std::move(createTVec());                  // error!
```

- An rvalue reference does not, either, as lvalues cannot bind to them:

```
TVec&& std::move(TVec&& obj) noexcept; // possible move instantiation
TVec vt;
std::move(vt);                          // error!
```

- What std::move needs:
  - For lvalue arguments, a parameter type of T&
  - For rvalue arguments, a parameter type of T&&

std::move: Parameter Type: Solution by Overloading?

- Overloading could solve the problem:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T& lvalue) noexcept {
    return static_cast<std::remove_reference<T>::type&&>(lvalue);
}
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& rvalue) noexcept {
    return static_cast<std::remove_reference<T>::type&&>(rvalue);
}
```

- But the *perfect forwarding problem*[1] would remain:
  - To forward *n* arguments to another function we would need 2^n overloads!
- *Rvalue references* aimed at both `std::move` and *perfect forwarding*

[1] *Perfect Forwarding Problem will be discussed in a later module*

std::move: T&& Parameter Deduction in Templates

- Given

```
template<typename T> void f(T&& param); // note non-const rvalue reference
```

- T's deduced type depends on what is passed to param:
  - **Lvalue** ⇒ T is an lvalue reference (T&)
  - **Rvalue** ⇒ T is a non-reference (T)
- In conjunction with reference collapsing:

```
int x;
f(x);              // lvalue: generates f<int&>(int& &&), calls f<int&>(int&)
f(10);             // rvalue: generates/calls f<int>(int&&)

TVec vt;           // typedef vector<int> TVec;
                   // TVec createTVec();
f(vt);             // lvalue: generates f<TVec&>(TVec& &&), calls f<TVec&>(TVec&)
f(createTVec());   // rvalue: generates/calls f<TVec>(TVec&&)
```

std::move: Implementation

- std::move's parameter is thus T&&:

```
template<typename T>
typename std::remove_reference<T>::type&&
    move(T&& obj) noexcept {
        return obj;
    }
```

- This is almost correct. Problem:
  - obj is an lvalue (It has a name)
  - move's return type is an rvalue reference
  - Lvalues cannot bind to rvalue references
- A cast eliminates the problem to give a correct implementation

```
template<typename T>
typename std::remove_reference<T>::type&&
    move(T&& obj) noexcept {
        using ReturnType = typename std::remove_reference<T>::type&&;
        return static_cast<ReturnType>(obj);
    }
```

T&& Parameters in Templates

- Compare conceptual and actual std::move declarations:

```
template<typename T>                        // conceptual
T&&
move(MagicReferenceType obj) noexcept;

template<typename T>                        // actual
typename std::remove_reference<T>::type&&
move(T&& obj) noexcept;
```

- T&& really is a magic[2] reference type!
  - For lvalue arguments, T&& becomes T& => lvalues can bind
  - For rvalue arguments, T&& remains T&& => rvalues can bind
  - For const/volatile arguments, const/volatile becomes part of T
  - T&& parameters can bind *anything*

[2] *Universal Reference will be discussed in a later module*

Move Semantics Project

Move Semantics Project

- To put the pieces of the Move Semantics puzzle together, we present a complete project with classes having resources of POD[3] and UDT
- We also code a resource management helper class (`ResMgr`) to *track the objects (resources) created and released dynamically*. This will help us to *quantify the benefits of move over copy*
- All functions are tracked with messages to understand *object lifetimes under copy and move*
- Using `ResMgr`, we present applications to compare the performance of the *Copy & Move* version of the classes against the *Copy Only* version
- The classes are (skeletons of `MyResource` and `MyClass` used earlier in the module):
  - `ResMgr`: *Resource Management Helper Class*. It has static member functions to `Create()`, `Release()` resources and print statistics (`Stat()`)
  - `MyResource`: *Resource Class with POD resource (`char*`)*. It has usual class members, overloaded output operator and a global function to call and return by value
  - `MyClass`: *Resource Class with UDT resource (`MyResource`)*. It has usual class members, overloaded output operator and a global function to call and return by value
- **The codes may be used to code move semantics in any project you develop**

[3] Plain Old Data (POD) refers to built-in types

Move Semantics: ResMgr: Resource Management Helper Class


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

class ResMgr { // Resource Management class to track creation and release of resources
static unsigned int nCreated, nReleased; // Counters for created & released resources
public:
    ResMgr() { } // Constructor to be called before main
    ~ResMgr() {  // Destructor to be called after main
        cout << "\n\nResources Created = " << nCreated << endl;
        cout << "Resources Released = " << nReleased << endl;
    }
    inline static char *Create(const char *s) // Create a resource from s
    { return (s) ? ++nCreated, strdup(s) : nullptr; } // If s is not null, copy & increment counter
    inline static void Release(char *s) // Release the resource held by s
    { (s) ? free(s), ++nReleased : 0; } // If s is not null, increment counter & free resource
    inline static void Stat() // Print stats for resources created and released
    { cout << " Stat = (" << nCreated << ", " << nReleased << ")\n\n"; }
};
unsigned int ResMgr::nCreated = 0; // Define and initialize Counters
unsigned int ResMgr::nReleased = 0;

ResMgr m; // Static resource manager instance
          // Created before call to main(). Destroyed after return from main()
```

Move Semantics: MyResource: POD Resource Class - *Copy Only*


```
class MyResource { // Representative resource class with copy-only support
    char *str = nullptr; // Resource pointer
public:
    MyResource(const char* s = nullptr) : str(ResMgr::Create(s)) // Param. & Defa. Ctor
    { cout << "Ctor[R] "; } // Creates resource
    MyResource(const MyResource& s) : str(ResMgr::Create(s.str)) // Copy Ctor
    { cout << "C-Ctor[R] "; } // Copy-Creates resource
    MyResource& operator=(const MyResource& s) { cout << "C=[R] "; // Copy Assignment
        if (this != &s) { ResMgr::Release(str); str = ResMgr::Create(s.str); }
        return *this; // Releases and Copy-Creates resource
    }
    ~MyResource() // Destructor
    { cout << "Dtor[R] "; ResMgr::Release(str); } // Releases resource
    friend ostream& operator<<(ostream& os, const MyResource& s) { // Streams resource value
        cout << ((s.str) ? s.str : "null"); return os; // Streams "null" for nullptr (no resource)
    }
};

MyResource f(MyResource s)      // Global function
{ cout << "f[R] "; return s; } // Uses call-by-value & return-by-value
```

Move Semantics: Application using *Copy Only* MyResource

```
// ResMgr m is constructed here
int main() { // Appl. to check resource behavior for copy-only support through copy ctor & assignment
    MyResource r1{ "ppd" }; // Ctor[R] r1=ppd Stat = (1, 0)
    cout << "r1=" << r1; ResMgr::Stat(); // r1 constructed with parameter

    MyResource r2{ r1 }; // C-Ctor[R] r2=ppd r1=ppd Stat = (2, 0)
    cout << "r2=" << r2 << " r1=" << r1; ResMgr::Stat(); // r2 copy constructed from r1

    MyResource r3{ f(r2) }; // C-Ctor[R] f[R] C-Ctor[R] Dtor[R] r3=ppd r2=ppd Stat = (4, 1)
    cout << "r3=" << r3 << " r2=" << r2; ResMgr::Stat(); // r3 C-Ctor from f(r2): C-Ctor / Dtor for param

    r1 = r2; // C=[R] r1=ppd r2=ppd Stat = (5, 2)
    cout << "r1=" << r1 << " r2=" << r2; ResMgr::Stat(); // r1 copy assigned from r2

    MyResource r4; // Ctor[R] r4=null Stat = (5, 2)
    cout << "r4=" << r4; ResMgr::Stat(); // r4 default constructed

    r4 = f(r3); // C-Ctor[R] f[R] C-Ctor[R] C=[R] Dtor[R] Dtor[R] r4=ppd r3=ppd Stat = (8, 4)
    cout << "r4=" << r4 << " r3=" << r3; ResMgr::Stat(); // r4 C= from f(r3): trace debug to understand
} // m.~ResMgr is called after the destruction of local automatic objects to print the final statistics
// Dtor[R] Dtor[R] Dtor[R] Dtor[R]
// Resources Created = 8 Resources Released = 8 // printed from m.~ResMgr
```

- Note that ResMgr m is a global static object that is *created before and destroyed after* main()
- Track function call messages to understand the lifetimes of object

Move Semantics: MyResource: POD Resource Class

Copy & Move

```
class MyResource { // Representative resource class with copy-and-move support
    char *str = nullptr; // Resource pointer
public:
    MyResource(const char* s = nullptr) : str(ResMgr::Create(s)) // Param. & Defa. Ctor
    { cout << "Ctor[R] "; } // Creates resource
    MyResource(const MyResource& s) : str(ResMgr::Create(s.str)) // Copy Ctor
    { cout << "C-Ctor[R] "; } // Copy-Creates resource
    MyResource(MyResource&& s) noexcept : str(s.str) // Move Ctor
    { cout << "M-Ctor[R] "; s.str = nullptr; } // Moves resource
    MyResource& operator=(const MyResource& s) { cout << "C=[R] "; // Copy Assignment
        if (this != &s) { ResMgr::Release(str); str = ResMgr::Create(s.str); }
        return *this; // Releases and Copy-Creates resource
    }
    MyResource& operator=(MyResource&& s) noexcept { cout << "M=[R] ";  // Move Assignment
        if (this != &s) { ResMgr::Release(str); str = s.str; s.str = nullptr; }
        return *this; // Releases and Moves resource
    }
    ~MyResource() // Destructor
    { cout << "Dtor[R] "; ResMgr::Release(str); } // Releases resource
    friend ostream& operator<<(ostream& os, const MyResource& s) { // Streams resource value
        cout << ((s.str) ? s.str : "null"); return os; // Streams "null" for nullptr (no resource)
    }
};
MyResource f(MyResource s)      // Global function
{ cout << "f[R] "; return s; } // Uses call-by-value & return-by-value
```

Move Semantics: Application using *Copy & Move* MyResource


```
int main() { // Application to check resource behavior for copy-and-move support through
             // copy construction / assignment and move construction / assignment
    MyResource r1{ "ppd" }; // Ctor[R] r1=ppd Stat = (1, 0)
    cout << "r1=" << r1; ResMgr::Stat();

    MyResource r2{ r1 }; // C-Ctor[R] r2=ppd r1=ppd Stat = (2, 0)
    cout << "r2=" << r2 << " r1=" << r1; ResMgr::Stat();

    MyResource r3{ f(r2) }; // C-Ctor[R] f[R] M-Ctor[R] Dtor[R] r3=ppd r2=ppd Stat = (3, 0)
    cout << "r3=" << r3 << " r2=" << r2; ResMgr::Stat(); // r3 M-Ctor from f(r2): C-Ctor / Dtor for param

    r1 = r2; // C=[R] r1=ppd r2=ppd Stat = (4, 1)
    cout << "r1=" << r1 << " r2=" << r2; ResMgr::Stat();

    MyResource r4; // Ctor[R] r4=null Stat = (4, 1)
    cout << "r4=" << r4; ResMgr::Stat();

    r4 = f(r3); // C-Ctor[R] f[R] M-Ctor[R] M=[R] Dtor[R] Dtor[R] r4=ppd r3=ppd Stat = (5, 1)
    cout << "r4=" << r4 << " r3=" << r3; ResMgr::Stat(); // r4 M= from f(r3): Note M-Ctor in f(r3)
}
// Dtor[R] Dtor[R] Dtor[R] Dtor[R]
// Resources Created = 5 Resources Released = 5
```

- **Compared to copy-only, we created and released (8 − 5) = 3 resources less with copy-and-move**

Move Semantics: MyClass: UDT Resource Class - *Copy & Move* (broken!)


```
class MyClass { MyResource mRrc; // Resource object
public:
    MyClass() : mRrc("") // Defa. Ctor
    { cout << "D-Ctor[C] "; }
    MyClass(const MyResource& r) : mRrc(r) // Param. Ctor
    { cout << "Ctor[C] "; }
    MyClass(const MyClass& c) : mRrc(c.mRrc) // Copy Ctor
    { cout << "C-Ctor[C] "; }
    MyClass(MyClass&& c) noexcept : mRrc(c.mRrc) // Move Ctor
    { cout << "M-Ctor[C] "; }
    MyClass& operator=(const MyClass& c) { cout << "C=[C] "; // Copy Assignment
        if (this != &c) { mRrc = c.mRrc; }
        return *this;
    }
    MyClass& operator=(MyClass&& c) noexcept { cout << "M=[C] "; // Move Assignment
        if (this != &c) { mRrc = c.mRrc; }
        return *this;
    }
    ~MyClass() /* Destructor */ { cout << "Dtor[C] "; }
    friend ostream& operator<<(ostream& os, const MyClass& c) // Streams resource value
    { cout << c.mRrc; return os; }
};
MyClass f(MyClass s)            // Global function
{ cout << "f[C] "; return s; } // Uses call-by-value & return-by-value
```

Move Semantics: Application using *Copy & Move* MyClass

```
int main() {
    MyResource r1{ "ppd" }; // Ctor[R] r1=ppd Stat = (1, 0)
    cout << "r1=" << r1; ResMgr::Stat();

    MyClass c1{ r1 }; // C-Ctor[R] Ctor[C] c1=ppd Stat = (2, 0)
    cout << "c1=" << c1; ResMgr::Stat();

    MyClass c2{ f(c1) }; // C-Ctor[R] C-Ctor[C] f[C] C-Ctor[R] M-Ctor[C] Dtor[C] Dtor[R]
                         // c2=ppd c1=ppd Stat = (4, 1)
                         // c2 C-Ctor[C] from f(c1). Calls M-Ctor[C] yet copies resource
    cout << "c2=" << c2 << " c1=" << c1; ResMgr::Stat();

    c1 = f(c2); // C-Ctor[R] C-Ctor[C] f[C] C-Ctor[R] M-Ctor[C] M=[C] C=[R]
                // Dtor[C] Dtor[R] Dtor[C] Dtor[R]
                // c1=ppd c2=ppd Stat = (7, 4)
                // c1 C=[C] from f(c2). Calls M=[C] yet copies resource
    cout << "c1=" << c1 << " c2=" << c2; ResMgr::Stat();
}
// Dtor[C] Dtor[R] Dtor[C] Dtor[R] Dtor[R]
// Resources Created = 7 Resources Released = 7
```

Move Semantics: MyClass: : UDT Resource Class - *Copy & Move* (**fixed!**)

Module M50

Partha Pratim Das


```
class MyClass { MyResource mRrc; // Resource object
public:
    MyClass() : mRrc("") // Defa. Ctor
    { cout << "D-Ctor[C] "; }
    MyClass(const MyResource& r) : mRrc(r) // Param. Ctor
    { cout << "Ctor[C] "; }
    MyClass(const MyClass& c) : mRrc(c.mRrc) // Copy Ctor
    { cout << "C-Ctor[C] "; }
    MyClass(MyClass&& c) noexcept : mRrc(std::move(c.mRrc)) // Move Ctor
    { cout << "M-Ctor[C] "; }
    MyClass& operator=(const MyClass& c) { cout << "C=[C] "; // Copy Assignment
        if (this != &c) { mRrc = c.mRrc; }
        return *this;
    }
    MyClass& operator=(MyClass&& c) noexcept { cout << "M=[C] "; // Move Assignment
        if (this != &c) { mRrc = std::move(c.mRrc); }
        return *this;
    }
    ~MyClass() /* Destructor */ { cout << "Dtor[C] "; }
    friend ostream& operator<<(ostream& os, const MyClass& c) // Streams resource value
    { cout << c.mRrc; return os; }
};
MyClass f(MyClass s)           // Global function
{ cout << "f[C] "; return s; } // Uses call-by-value & return-by-value
```

Move Semantics: Application using *Copy & Move* MyClass

```
int main() {
    MyResource r1{ "ppd" }; // Ctor[R] r1=ppd Stat = (1, 0)
    cout << "r1=" << r1; ResMgr::Stat();

    MyClass c1{ r1 }; // C-Ctor[R] Ctor[C] c1=ppd Stat = (2, 0)
    cout << "c1=" << c1; ResMgr::Stat();

    MyClass c2{ f(c1) }; // C-Ctor[R] C-Ctor[C] f[C] M-Ctor[R] M-Ctor[C] Dtor[C] Dtor[R]
                         // c2=ppd c1=ppd Stat = (3, 0)
                         // c2 C-Ctor[C] from f(c1). Calls M-Ctor[C] and does not copy. FIXED
    cout << "c2=" << c2 << " c1=" << c1; ResMgr::Stat();

    c1 = f(c2); // C-Ctor[R] C-Ctor[C] f[C] M-Ctor[R] M-Ctor[C] M=[C] M=[R]
                // Dtor[C] Dtor[R] Dtor[C] Dtor[R]
                // c1=ppd c2=ppd Stat = (4, 1)
                // c1 C=[C] from f(c2). Calls M-Ctor[C] and does not copy. FIXED
    cout << "c1=" << c1 << " c2=" << c2; ResMgr::Stat();
}
// Dtor[C] Dtor[R] Dtor[C] Dtor[R] Dtor[R]
// Resources Created = 4 Resources Released = 4
```

- **Compared to copy-and-move without `std::move`, we created and released $(7-4) = 3$ resources less with `std::move`**

Module Summary

- Learnt to implement move semantics in UDTs using `std::move`
- Understood the use and implementation of `std::move`
- Studied a project to code move-enabled UDTs