Programming in Modern C++

Tutorial T10: How to optimize C++11 programs using Rvalue and Move Semantics?

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

Tutorial Objectives

- To understand optimization by copy elision
- To understand copy / move optimization by Rvalues and Move Semantics

Tutorial Outline

Optimizing C++11 Programs

Sources:

- Move semantics and rvalue references in C++11, cprogramming, Alex Allain, 2019
- Copy elision, wikipedia

Optimizing C++11 Programs

- C++ has always produced fast programs
- Unfortunately, until C++11, there has been an obstinate wart that slows down many C++ programs:
  - the *creation of temporary objects*
- Sometimes these temporary objects can be optimized away by the compiler by *copy elision*[1] (the return value optimization, for example). But this is not always the case, and it can result in expensive object copies
- Copy elision (or omission) depends primarily on identification of rvalues by the compiler and can be optimized away
- In addition to what the compiler can do, we can reduce copies by explicitly marking rvalues in the code by Rvalue references and by providing the move operations along with the copy operations (if needed)
- We first elucidate some common scenarios of copy elision that the language standard specifies and the compiler exploits for optimization
- Next we show through a small sorting project how the programmer can expose good move opportunities for the compiler to optimize copies

[1] compiler optimization technique that eliminates unnecessary copying of objects

Copy Elision

Sources:

- Copy elision, cppreference
- Copy elision in C++, geeksforgeeks, 2017
- Copy elision, wikipedia

Copy Elision

- In C++ programming, **copy elision**[2] refers to a *compiler optimization technique that eliminates unnecessary copying of objects*
- The C++ language standard generally allows implementations to perform any optimization, provided the resulting program's observable behavior is the same as if, that is pretending, the program were executed exactly as mandated by the standard.
- Beyond that, the standard also describes a few situations where copying can be eliminated even if this would alter the program's behavior
  - the most common being the *return value optimization*
  - Another widely implemented optimization, described in the C++ standard, is when a temporary object of class type is copied to an object of the same type. As a result
    - *copy-initialization* is usually equivalent to *direct-initialization* in terms of performance, but semantically,
    - copy-initialization still requires an accessible copy constructor
- The optimization cannot be applied to a temporary object that has been bound to a reference

[2] *elision* is the omission of a sound or syllable when speaking (as in I'm, let's)

Copy Elision: Copy Initialization

Copy Elision: Copy Initialization

- What will be the output?

```
#include <iostream>
int n = 0;
struct C {
    explicit C(int) { }
    C(const C&) { ++n; } // the copy constructor has a visible side effect
};                        // it modifies an object with static storage duration
int main() {
    C c1(42);      // direct-initialization, calls C(int). c1 is lvalue
    C c2 = C(77); // copy-initialization. C(77) by C(int) is rvalue, skips C(const C&)
    std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 0
}
```

- Interestingly both GCC-C++ and MSVC++ and print 0 even in debug build
- Copy constructor `C::C(const C&)` is not even invoked
- If you think this is because `C::C(const C&)` does not do anything meaningful for the object, check the next version

Copy Elision: Copy Initialization

- What will be the output?

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" << i << ' '; }
};
int main() {
    C c1(42);      // direct-init., calls C(int). c1 is lvalue                  // 42
    C c2 = C(77); // copy-init. C(77) by C(int) is rvalue, skips C(const C&) // 77
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 0
} // ~77 ~42
```

- `C::C(const C&)` is just not invoked!
- Yet, if you comment the copy constructor and explicitly delete it (`C(const C&)` = delete;) so that no free copy constructor is provided, C++11 will give `error: use of deleted function 'C::C(const C&)'`

Copy Elision: Copy Initialization

- Let us construct an object from an lvalue

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" <<  i << ' '; }
};
int main() {
    C c1(42);     // direct-init., calls C(int). c1 is lvalue                // 42
    C c2 = C(77); // copy-init. C(77) by C(int) is rvalue, skips C(const C&) // 77
    C c3 = c1;    // copy-init., calls C(const C&) as c1 is lvalue           // 43
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 1
} // ~43 ~77 ~42
```

Copy Elision: Copy Initialization

- Using `-fno-elide-constructors` option to disable copy-elision:

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" <<  i << ' '; }
};
int main() {
    C c1(42);     // direct-init., calls C(int). c1 is lvalue                   // 42
    C c2 = C(77); // copy-init. C(77) by C(int) is rvalue, skips C(const C&) // 77 78 ~77
    C c3 = c1;    // copy-init., calls C(const C&) as c1 is lvalue              // 43
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 2
} // ~43 ~78 ~42
```

Copy Elision: Return Value Optimization (RVO)

Copy Elision: Return Value Optimization (RVO)

- Similar behaviour would be observed through function return by direct construction:

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" <<  i << ' '; }
};
C f(int i) {
    return C(i); // directly constructed object by C(int): C(i) is rvalue
} // rvalue C(i) is to be copy constructed by C(const C&) to be returned as rvalue. Skipped
C g(int i) {
    C c(i);   // directly constructed object: c is lvalue needs C(int)
    return c; // return object constructed from c by C(const C&) to be returned as rvalue
}
int main() {
    f(19);        // f(19) is rvalue - unused and destructed                  // 19 ~19
    g(35);        // f(19) is rvalue - unused and destructed                  // 35 ~35
    std::cout << n << std::endl; // prints 0 if the copy was elided, 1 otherwise // 0
}
```

Copy Elision: Return Value Optimization (RVO)

- Similar behaviour is also observed if the return value is used in initialization without being discarded – however, the destruction order changes:

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" <<  i << ' '; }
};
C f(int i) {
    return C(i); // directly constructed object by C(int): C(i) is rvalue
} // rvalue C(i) is be copy constructed by C(const C&) to be returned as rvalue. Skipped
C g(int i) {
    C c(i);   // directly constructed object: c is lvalue needs C(int)
    return c; // return object constructed from c by C(const C&) to be returned as rvalue
}
int main() {
    C c1 = f(19); // copy-init. f(19) by C(int) is rvalue, skips C(const C&) // 19
    C c2 = g(35); // copy-init. g(35) by C(int) is rvalue, skips C(const C&) // 35
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 0
} // ~35 ~19
```

Copy Elision: Return Value Optimization (RVO)

- Using `-fno-elide-constructors` option to disable copy-elision:

```
#include <iostream>
int n = 0;
struct C { int i;
    explicit C(int i) : i(i) { std::cout << i << ' '; }
    C(const C& c) : i(c.i) { std::cout << ++i << ' '; ++n; }
    ~C() { std::cout << "~" <<  i << ' '; }
};
C f(int i) {
    return C(i); // directly constructed object by C(int): C(i) is rvalue
} // rvalue C(i) is to be copy constructed by C(const C&) to be returned as rvalue. Skipped
C g(int i) {
    C c(i);   // directly constructed object: c is lvalue needs C(int)
    return c; // return object constructed from c by C(const C&) to be returned as rvalue
}
int main() {
    C c1 = f(19); // copy-init. f(19) by C(int) is rvalue, skips C(const C&) // 19 20 ~19 21
    C c2 = g(35); // copy-init. g(35) by C(int) is rvalue, skips C(const C&) // 35 36 ~35 37
    std::cout << n << std::endl;  // prints 0 if the copy was elided, 1 otherwise // 4
} // ~37 ~21
```

Copy Elision: Language Specification

Sources:

- Copy elision, cppreference

Mandatory Elision: Copy / Move Operations

- Under the following circumstances, the compilers are required to omit the copy and move construction of class objects, even if the copy / move constructor and the destructor have observable side-effects
- *Objects are constructed directly into the storage where they would be copied / moved to*
- *The copy / move constructors need not be present or accessible*:
  - In a return statement, when the operand is a prvalue of the same class type (ignoring cv-qualification) as the return type:

```
T f() {
    return T();
}

f(); // only one call to default constructor of T
```

  - More are specified for C++17

Non-Mandatory Elision: Copy / Move Operations

- Under the following circumstances, the compilers are *permitted, but not required* to omit the copy and move construction of class objects, *even if the copy / move constructor and the destructor have observable side-effects*
- *Objects are constructed directly into the storage where they would be copied / moved to*
- **This is an optimization**: *even when it takes place and the copy / move constructor is not called, it still must be present and accessible (as if no optimization happened at all), otherwise the program is ill-formed*:
    - In a *return statement*, when the operand is a named object (and not a function or a catch clause param) with automatic storage duration, and which is of the same class type (ignoring cv-qualification) as the function return type. This variant of copy elision is known as **Named Return Value Optimization (NRVO)**
    - In the *initialization of an object*, when the source object is a nameless temporary and is of the same class type (ignoring cv-qualification) as the target object. When the nameless temporary is the operand of a return statement, this variant of copy elision is known as **Return Value Optimization (RVO)**
    - Return value optimization is mandatory and no longer considered as copy elision since C++17

Sorting Objects

Sorting Objects

- To illustrate the effect by copy optimization, we consider a tiny sorting project
- We intend to sort objects of a data class `D` having resource of a class `R`
- We define the following to get started:
  - Resource class `R`
  - Data class `D`
  - A template function `swap`
  - A template function `sort` to bubble sort an array
  - The `main` function to initialize an array and sort it
- We are interested to see the trade-off of move and copy. So we build a statistics support in the code to count the number of constructions and destructions of the resource objects from class `R`
- Initial version works with only Copy operations
- We next add move operations in Data class `D` and move support in `swap` function
- We compare the statistics to show the huge benefit accrued with the move semantics

Sorting Objects: Copy Support

Resource Class, R

Tutorial T10

Partha Pratim Das

Objective & Outline

Optimizing C++11 Programs

Copy Elision

Copy Initialization

Return Value Optimization (RVO)

Language Specification

Sorting Objects

Copy Support

Statistics Support

Move Support

Summary

Project Codes

Problems

Tutorial Summary

- Let us consider a resource class R, and
- A data class D having resource R:

```
struct R { // Resource class - wrapper of int
    int i; // Wrapped resource
    R(int i) : i(i) { }          // Parametric constructor
    R(const R& r) : i(r.i) { } // Copy constructor
    ~R() {}                      // Destructor
};

struct D { // Data class with resource
    R* r; // Resource to be dynamically constructed / destructed

    // ...
};
```

Data Class, D


```
struct D { // Data class with resource
    R* r;                                     // Resource to be dynamically constru. / destru.
    D() : r(nullptr) { }                      // Default constructor - null resource
    D(int i) : r(new R(i)) { }                // Parametric constructor - create resource
    D(const D& d) : r(new R(*(d.r))) { }      // Copy constructor - copy resource
    D& operator=(const D& d) {                // Copy assignment - copy resource
        if (this != &d) {       // Self copy guard
            delete r;           // Free resource
            r = new R(*(d.r)); // Copy resource
        }
        return *this;
    }
    ~D() { delete r; }                        // Destructor - free resource
    friend bool operator>(const D& c1, const D& c2) { // Compare D objects for sorting
        return c1.r->i > c2.r->i;
    }
    friend std::ostream& operator<<(std::ostream& os, const D& d) { // Stream D objects
        os << d.r->i << ' ';
        return os;
    }
};
```

sort Function

- We store N number of D objs in an array
- We sort the array by Bubble Sort in ascending order

```
template<typename T>
void swap(T& a, T&b) { // Swap a and b using copy
    T t = a; // t copy-created from a: two a's
    a = b;   // a copy-assigned from b: two b's, one a destroyed
    b = t;   // b copy-assigned from t: two t's, one b destroyed
} // t destroyed

template<typename T>
void sort(T arr[], int n) { // Bubble Sort for easy analysis
    for (int i = 0; i < n - 1; ++i)
        for (int j = 0; j < n - i - 1; ++j)
            if (arr[j] > arr[j + 1]) {   // Compare by D::operator>
                swap(arr[j], arr[j + 1]); // 3 constr.s and destr.s of R objs with copy
                                          // 0 constr. and destr. of R objs with move
    }
}
```

main Function

```
int main() { // To populate and sort an array of D objs having R obj resources
    const int N = 10;            // Size of array and number of elements
    D arr[N];                    // Defa. initialization of array - use D::D() calls N times

    // Assignments of array elements with D objs having R obj resources
    // Fill with a strictly decreasing sequence for worst case of Bubble Sort
    for (int i = N - 1; i >= 0; --i)
        arr[i] = D(N - i); // Construct by D::D(int), assign by D::operator=(const D&)
                           // construct / destruct R objs
    for (int i = 0; i < N; ++i) // Print array before sorting. 10 9 8 7 6 5 4 3 2 1
        std::cout << arr[i]; std::cout << std::endl;
    sort(arr, N);                // Sort array in ascending order
    for (int i = 0; i < N; ++i) // Print array after sorting. 1 2 3 4 5 6 7 8 9 10
        std::cout << arr[i]; std::cout << std::endl;
}
```

- To get an estimate for the resource construct. and destruct., we build a worst-case for Bubble Sort, that is, populate arr in strictly descending order. Being sorting, this is dominated by swap
- Clearly in the worst case number of swaps $= \sum_{i=1}^{N-1} i = \frac{N*(N-1)}{2}$. Hence number of (unnecessary) resource constructions and destructions $= 3 * \#$ of swaps $= \frac{3*N*(N-1)}{2}$

Sorting Objects: Statistics Support

Sorting Objects: Statistics Support

Resource Class R with Statistics

- To count the exact number of constructions and destructions of R objects, we add three static counters in R
- We also add a static method stat() to print the statistics at anytime from anywhere

```
struct R { // Resource class
    int i; // Wrapped resource
    R(int i) : i(i) { ++nCtor; }               // Parametric constructor
    R(const R& r) : i(r.i) { ++nC_Ctor; } // Copy constructor
    ~R() { ++nDtor; }                          // Destructor

    static unsigned int nCtor;   // Count of direct construction of R objects
    static unsigned int nC_Ctor; // Count of copy construction of R objects
    static unsigned int nDtor;   // Count of destruction of R objects

    static void stat(std::string s) { // Print R object statistics
        std::cout << s /* Banner message */ << "R obj Created = " << R::nCtor <<
            " R obj Copy Created = " << R::nC_Ctor << " R obj Destroyed = " << R::nDtor <<
            std::endl;
    }
};
```

Resource Class R with Statistics

- Static counters of R are globally instantiated and initialized with 0's.
- We also add helper class Stat whose constructor and destructor calls R::stat(). Next we globally instantiate an object extremeStat of Stat
  - Being global static, extremeStat is constructed before main() is called and is destructed after main() returns
  - Hence the statistics in printed before calling main() and after returning from main()

```
// Instantiations of static R objects in global namespace
unsigned int R::nCtor = 0;   // Count of direct construction of R objects
unsigned int R::nC_Ctor = 0; // Count of copy construction of R objects
unsigned int R::nDtor = 0;   // Count of destruction of R objects

struct Stat { // Helper class to print R objects statistics
    Stat() { R::stat("Program Start: "); } // Construct before main(), initial stat
    ~Stat() { R::stat("Program End: "); }  // Destruct after main(), final stat
} extremeStat;
```

main Function with Statistics

```
int main() { // To populate and sort an array of D objs having R obj resources
    const int N = 10;           // Size of array and number of elements
    D arr[N];                   // Defa. initialization of array - use D::D() calls N times
    R::stat("Array Defa: ");    // Statistics after Defa. initialization of array
    // Assignments of array elements with D objs having R obj resources
    // Fill with a strictly decreasing sequence for worst case of Bubble Sort
    for (int i = N - 1; i >= 0; --i)
        arr[i] = D(N - i); // Construct by D::D(int), assign by D::operator=(const D&)
                           // construct / destruct R objs
    R::stat("Array Init: ");    // Statistics after assignment of array
    for (int i = 0; i < N; ++i) // Print array before sorting
        std::cout << arr[i]; std::cout << std::endl;
    sort(arr, N);               // Sort array in ascending order
    R::stat("Array Sort: ");    // Statistics after sorting of array
    for (int i = 0; i < N; ++i) // Print array after sorting
        std::cout << arr[i]; std::cout << std::endl;
} // Statistics after destruction of array elements by s.~Stat()
```

```
Program Start: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
Array Defa: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0
Array Init: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 0
10 9 8 7 6 5 4 3 2 1
Array Sort: R obj Created = 10 R obj Copy Created = 135 R obj Destroyed = 135
1 2 3 4 5 6 7 8 9 10
Program End: R obj Created = 10 R obj Copy Created = 135 R obj Destroyed = 145
```

Analysis of Statistics

- `Program Start:  R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0`
    - Static object `extremeStat` constructed by `Stat::Stat()` before `main()` is invoked, reports statistics
- `Array Defa:  R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0`
    - `D arr[N]`: `N` = 10 `D` objects are constructed by `D::D()`. As `D::r` is set to `nullptr` in each, no `R` object is constructed
- `Array Init:  R obj Created = 10 R obj Copy Created = 10 R obj Destroyed = 10`
    - `...  = D(N - i)`: `N` = 10 `D` objects are constructed by `D::D(int)`. As `D::r` is set to `new R(i)` in each, `N` = 10 `R` object is constructed
    - `arr[i] = ...`: `D(N - i)` is now *copy assigned* to `arr` elements by `D::D(const D& d)`. Hence, the resource `R` objects is destructed (`delete r`) and constructed (`new R(*(d.r))`) for each
    - Note that `D(N - i)` is an rvalue, yet it is *copy assigned* as there is no *move assignment*
- `10 9 8 7 6 5 4 3 2 1`
    - `arr` before sorting. Filled with a strictly decreasing sequence
- `Array Sort:  R obj Created = 10 R obj Copy Created = 145 R obj Destroyed = 145`
    - `sort(arr, N);`: Being the worst case of bubble sort, $\frac{3*N*(N-1)}{2} = \frac{3*10*(10-1)}{2} = 135$ `R` objects are constructed by `R::R(const R&)` and destructed by `R::~R()` for 45 `swap`s. Note that `t`, `a`, and `b` in `swap` are lvalues
- `1 2 3 4 5 6 7 8 9 10`
    - `arr` after sorting in increasing order
- `Program End:  R obj Created = 10 R obj Copy Created = 145 R obj Destroyed = 155`
    - `int main() { ...  }`: Remaining `N` = 10 `D` objects destructed by `D:: D()` with `delete D::r`. Static object `extremeStat` destructed by `Stat::~Stat()` after `main()` returns reports

Sorting Objects: Move Support

Data Class D with Move Support

- To minimize copies, we provide move operations in class D to be able to move rvalues whenever possible

```
struct D { // Data class with resource
    R* r;                                      // Resource to be dynamically constru. / destru.
    D();                      // Default constructor - null resource
    D(int i);                 // Parametric constructor - create resource
    D(const D& d);            // Copy constructor - copy resource
    D& operator=(const D& d); // Copy assignment - copy resource
    ~D();                     // Destructor - free resource

    D(D&& d) : r(d.r) { d.r = nullptr; } // Move constructor - move resource, ownership
    D& operator=(D&& d) {                // Move assignment - move resource, ownership
        if (this != &d) { // Self move guard
            r = d.r;      // Move resource
            d.r = nullptr; // Take ownership
        }
        return *this;
    }
};
```

- We again run the program and gather statistics

Analysis of Statistics: Move Support in Class D

- Here is the statistics with move support. We note the changes:
- `Program Start: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0`
- `Array Defa: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0`
- `Array Init: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 0`
  - `... = D(N - i)`: `N` = 10 `D` objects are constructed by `D::D(int)`. As `D::r` is set to `new R(i)` in each, `N` = 10 `R` object is constructed
  - `arr[i] = ...`: `D(N - i)` is now *move assigned* to `arr` elements by `D::D(D&& d)` since `D(N - i)` is a rvalue
  - Hence, no resource `R` object is destructed or constructed - just owenership of resource is transferred
- `10 9 8 7 6 5 4 3 2 1`
- `Array Sort: R obj Created = 10 R obj Copy Created = 135 R obj Destroyed = 135`
- `1 2 3 4 5 6 7 8 9 10`
- `Program End: R obj Created = 10 R obj Copy Created = 135 R obj Destroyed = 145`

swap Function with Move Support

- To minimize copies further, we provide move support in `swap()` function using `std::move`

```
template<typename T>
void swap(T& a, T&b) { // Swap a and b using move
    T t = std::move(a); // t move-created from a: a's ownership transferred to t
    a = std::move(b);   // a move-assigned from b: b's ownership transferred to a
    b = std::move(t);   // b move-assigned from t: t's ownership transferred to b
} // t destroyed, but no resource destruction as t had no ownership
```

Analysis of Statistics: Move Support in Class `D` and Function `swap`

- Here is the statistics with move support. We note the changes:
- `Program Start: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0`
- `Array Defa: R obj Created = 0 R obj Copy Created = 0 R obj Destroyed = 0`
- `Array Init: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 0`
- `10 9 8 7 6 5 4 3 2 1`
- `Array Sort: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 0`
  - `sort(arr, N);`: Being the worst case of bubble sort, $\frac{N*(N-1)}{2} = \frac{10*(10-1)}{2} = 45$ `swap`s are performed. But no `swap` *copies* any `R` object *only moves*
  - Hence no *unnecessary* construction and destruction of `R` objects
- `1 2 3 4 5 6 7 8 9 10`
- `Program End: R obj Created = 10 R obj Copy Created = 0 R obj Destroyed = 10`

Sorting Objects: Summary

Analysis of Statistics: Summary

`struct D`		`void swap(T&, T&)`		`R(int)`	`R(const R&)`	`~R()`
Only Copy	**Copy+ Move**	**Copy**	**Move**			
Yes		Yes		N	$\frac{3N*(N-1)}{2} + N$	$\frac{3N*(N-1)}{2} + 2N = \frac{N*(3N+1)}{2}$
Yes			Yes	N	$\frac{3N*(N-1)}{2} + N$	$\frac{3N*(N-1)}{2} + 2N = \frac{N*(3N+1)}{2}$
	Yes	Yes		N	$\frac{3N*(N-1)}{2}$	$\frac{3N*(N-1)}{2} + N = \frac{N*(3N-1)}{2}$
	Yes		**Yes**	N	0	N

- With move support in the class and in swap function, we can elide $O(N^2)$ copies (and destructions)

Sorting Objects: Project Codes

Resource Class R


```
struct R { // Resource class
    int i; // Wrapped resource
    R(int i) : i(i) { ++nCtor; }          // Parametric constructor
    R(const R& r) : i(r.i) { ++nC_Ctor; } // Copy constructor
    ~R() { ++nDtor; }                     // Destructor

    static unsigned int nCtor;   // Count of direct construction of R objects
    static unsigned int nC_Ctor; // Count of copy construction of R objects
    static unsigned int nDtor;   // Count of destruction of R objects

    static void stat(std::string s) { // Print R object statistics
        std::cout << s /* Banner message */ << "R obj Created = " << R::nCtor <<
            " R obj Copy Created = " << R::nC_Ctor << " R obj Destroyed = " << R::nDtor <<
            std::endl;
    }
};
// Instantiations of static R objects in global namespace
unsigned int R::nCtor = 0;   // Count of direct construction of R objects
unsigned int R::nC_Ctor = 0; // Count of copy construction of R objects
unsigned int R::nDtor = 0;   // Count of destruction of R objects

struct Stat { // Helper class to print R objects statistics
    Stat() { R::stat("Program Start: "); } // Construct before main(), initial stat
    ~Stat() { R::stat("Program End: "); }  // Destruct after main(), final stat
} extremeStat;
```

Data Class D


```
struct D { // Data class with resource
    R* r;                                       // Resource to be dynamically constru. / destru.
    D() : r(nullptr) { }                        // Default constructor - null resource
    D(int i) : r(new R(i)) { }                  // Parametric constructor - create resource
    D(const D& d) : r(new R(*(d.r))) { } // Copy constructor - copy resource
    D& operator=(const D& d) {                  // Copy assignment - copy resource
        if (this != &d) { // Self copy guard
            delete r;     /* Free resource */ r = new R(*(d.r)); // Copy resource
        } return *this;
    }
    ~D() { delete r; }                          // Destructor - free resource
#ifdef _MOVE_ // If _MOVE_ is defined (set -D=_MOVE_ flag in GCC to define _MOVE_), use move operations
    D(D&& d) : r(d.r) { d.r = nullptr; } // Move constructor - move resource, ownership
    D& operator=(D&& d) {                       // Move assignment - move resource, ownership
        if (this != &d) { // Self move guard
            r = d.r;      /* Move resource */ d.r = nullptr; // Take ownership
        } return *this;
    }
#endif // _MOVE_ // End of conditional compilation by _MOVE_
    friend bool operator>(const D& c1, const D& c2) { // Compare D objects for sorting
        return c1.r->i > c2.r->i;
    }
    friend std::ostream& operator<<(std::ostream& os, const D& d) { // Stream D objects
        os << d.r->i << ' '; return os;
    }
};
```

swap & sort Functions

```
#ifndef _MOVE_ // If _MOVE_ is not defined, use copy version
template<typename T>
void swap(T& a, T&b) { // Swap a and b using copy
    T t = a; // t copy-created from a: two a's
    a = b;   // a copy-assigned from b: two b's, one a destroyed
    b = t;   // b copy-assigned from t: two t's, one b destroyed
} // t destroyed
#else // If _MOVE_ is defined (set -D=_MOVE_ flag in GCC to define _MOVE_), use move version
template<typename T>
void swap(T& a, T&b) { // Swap a and b using move
    T t = std::move(a); // t move-created from a: a's ownership transferred to t
    a = std::move(b);   // a move-assigned from b: b's ownership transferred to a
    b = std::move(t);   // b move-assigned from t: t's ownership transferred to b
} // t destroyed, but no resource destruction as t had no ownership
#endif // _MOVE_ // End of conditional compilation by _MOVE_

template<typename T>
void sort(T arr[], int n) { // Bubble Sort for easy analysis
    for (int i = 0; i < n - 1; ++i)
        for (int j = 0; j < n - i - 1; ++j)
            if (arr[j] > arr[j + 1]) {    // Compare by D::operator>()
                swap(arr[j], arr[j + 1]); // 3 constr.s and destr.s of R objs with copy
                                          // 0 constr. and destr. of R objs with move
    }
}
```

main Function


```
int main() { // To populate and sort an array of D objs having R obj resources
    const int N = 10;          // Size of array and number of elements
    D arr[N];                  // Defa. initialization of array - use D::D() calls N times
    R::stat("Array Defa: ");   // Statistics after Defa. initialization of array

    // Assignments of array elements with D objs having R obj resources
    // Fill with a strictly decreasing sequence for worst case of Bubble Sort
    for (int i = N - 1; i >= 0; --i)
        arr[i] = D(N - i); // Construct by D::D(int), assign by
                           // D::operator=(const D&) for copy, constr. / destru. R objs
                           // D::operator=(D&&) for move, no constr. / destru. R objs

    R::stat("Array Init: ");   // Statistics after assignment of array
    for (int i = 0; i < N; ++i) // Print array before sorting
        std::cout << arr[i];
    std::cout << std::endl;

    sort(arr, N);              // Sort array in ascending order

    R::stat("Array Sort: ");   // Statistics after sorting of array
    for (int i = 0; i < N; ++i) // Print array after sorting
        std::cout << arr[i];
    std::cout << std::endl;
} // Statistics after destruction of array elements by s.~Stat()
```

Problems

- Provide construction / destruction counting and statistics generation support for class `D`
- Consider that the resource in `D` is held as a data member (`R r;`) and not as a pointer (`R *r;`). Provide appropriate support in classes `R` and `D` to avoid unnecessary copies during sorting
- Explore the move support in standard library containers, especially `vector` and `map`

Tutorial Summary

- Understood optimization by copy elision
- Understood copy / move optimization by Rvalues and Move Semantics
- Developed a complete sorting project with copy optimization by move