Resource Ownership in C++

Jan Faigl

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Lecture 13

B3B36PRG – Programming in C

Overview of the Lecture

- Part 1 – RAII Principle (in C++)
  - Acquisition-Release Pattern in C/C++
  - RAII – Resource Acquisition is Initialization
  - RAII Threading
  - Smart Pointers
- Part 2 – Move and Copy Semantics (in C++)
  - Assignment of Objects Holding Resources
  - lvalues & rvalues
  - Move and Copy Semantics

Part I

Part 1 – RAII Principle (in C++)

Acquisition-Release Pattern in C

```c
int main(void)
{
    int *array = malloc(SIZE * sizeof(int));  /* ACQUISITION */

    /* do work */

    free(array);  /* RELEASE */
    return 0;
}
```


Acquisition-Release Pattern in C

```c
int main(void)
{
    FILE *in_file = fopen(FILE_NAME, "r");  /* ACQUISITION */

    /* do work */

    fclose(in_file);  /* RELEASE */
    return 0;
}
```


Acquisition-Release Pattern in C

```c
int main(void)
{
    pthread_mutex_init(&mtx, NULL);
    pthread_mutex_lock(&mtx);  /* ACQUISITION */

    /* do work in critical section */

    pthread_mutex_unlock(&mtx);  /* RELEASE */
    return 0;
}
```


Acquisition-Release Pattern in C

```c
int main(void)
{
    pthread_create(&thread, NULL, foo, NULL);  /* ACQUISITION */

    /* do work */

    pthread_join(&thread, NULL);  /* RELEASE */
    return 0;
}
```


Acquisition-Release Pattern in C++

```c
int main(void)
{
    MyClass* c = new MyClass();  /* ACQUISITION */
    int* array = new int[SIZE];

    /* do work */

    delete[] array;
    delete c;  /* RELEASE */
    return 0;
}
```


But what if something goes wrong?

```c
int main(void)
{
   int *array = malloc(SIZE * sizeof(int));  /* ACQUISITION */

   if(!everithing_ok) {
      return 100;  /* !!! Resource is not released */
   }

   free(array);  /* RELEASE */
   return 0;
}
```

Automatic Destructor Call

- Destructor is called at the end of life-time!

```
int main(void)
{
    MyClass c;  /* Constructor MyClass() is called */

    /* do work */

    return 0;
    // ~MyClass() /* Desctructor is called at the end of scope.  */
}
```


Automatic Destructor Call

- Destructor is called at the end of life-time!

```
int main(void)
{
    MyClass c;  /* Constructor MyClass() is called */

    if(not everithing_ok) {
        return 100;
        // ~MyClass() /* EVEN HERE! */
    }

    return 0;
    // ~MyClass() /* Desctructor is called at the end of scope.  */
}
```


Resources Acquisition is Initialization

- Implement *resource acquisition* in a constructor(*initialization*).
- Failure to release resource is handled by throwing an exception.
- Resource release is handled by the destructor.

- Resource is bound to lifetime object instance.

Example Array Implementation

```
struct MallocException : std::exception {
    const char* what() const noexcept { return "Malloc error"; }
};

class MyArray {
    ulong size_p;
    int* data_p;
public:
    MyArray(ulong size);
    ~MyArray();

    int& operator[](ulong index);
    uint size() const;
};
```

lec13/myarray.cpp

Example Array Implementation

```
MyArray::MyArray(ulong size) : size_p(size) {
    data_p = (int*)calloc(size, sizeof(int));
    if(data_p == nullptr) {
        throw MallocException();
    }
}

MyArray::~MyArray() {
    free(data_p);
}
```


Implementation of RAII in Standard Library

- Dynamic array – `std::vector`
- File – `std::ifstream` / `std::ofstream`
- Mutex – `std::lock_guard`
- Thread – `std::jthread`
- Pointer to heap – `std::unique_pointer` / `std::shared_pointer`

`std::vector`

- Generic wrapper for dynamic array.
- More general version of `MyArray`.
- Other useful features:
  such as `push_back()` with dynamic reallocation of the underlying array.

```
int main()
{
    std::vector<int> v = { 7, 5, 16, 8 };

    v.push_back(25);
    v.push_back(13);

    std::cout << "v = { ";
    for (int n : v) {
        std::cout << n << ", ";
    }
    std::cout << "}; \n";
}
```


File streams

```
int main(void)
{
    std::ofstream outFile("out.txt");
    outFile << "Hello World\n";

    std::ifstream inFile("in.txt");
    int a;
    inFile >> a;

    /* Destructor of outFile/inFile automatically closes the files. */
    return 0;
}
```


RAII Thread and Mutex

```
/* jthread not implemented in g++ 9.4.0 */
class my_jthread {
    std::thread thread;

public:
    template<class Function, class... Args>
    my_jthread(Function&& f, Args&&... args) : thread(f, args...) {};

    ~my_jthread() {
        if(thread.joinable()) {
            thread.join();
        }
    }
};
```

lec13/thread.cpp

RAII Thread and Mutex

```
class my_lock_guard {
    std::mutex* mtx;

public:
    my_lock_guard(std::mutex& mtx) : mtx(&mtx) {
        mtx.lock();
    };

    ~my_lock_guard() {
        mtx->unlock();
    };
};
```


RAII Thread and Mutex

```
void coutnWorker(int n, int* a, std::mutex* mtx) {
    for(int i = 0; i < n; ++i) {
        my_lock_guard guard(*mtx);
        int tmp = *a;
        std::this_thread::sleep_for(std::chrono::microseconds(1));
        *a = tmp + 1;
    }
}

void countTwice2(int* counter, int val) {
    std::mutex counterMutex;

    my_jthread thrd1(coutnWorker, val, counter, &counterMutex);
    my_jthread thrd2(coutnWorker, val, counter, &counterMutex);
}
```


RAII Thread and Mutex

```
int main(void)
{
    int counter = 0;
    countTwice2(&counter, 10);

    std::cout << "final counter value: " << counter << '\n';

    return 0;
}
```


Smart Pointers

- Wrappers around heap pointer.
- `std::unique_ptr`
  - Frees the memory on deletion.
  - Only one `unique_ptr` pointing to a specific address may exist.
  - May not be *copied* only *moved*.
- `std::shared_ptr`
  - Keeps reference counter.
  - Last shared pointer frees the memory.
  - Multiple `shared_ptrs` pointing to the same address may exist.

Shared Pointer

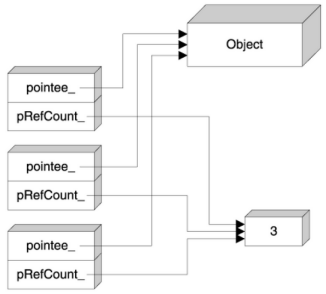


image source: https://stackoverflow.com/questions/9200664/how-is-the-stdtr1shared-ptr-implemented

Shared Pointer

```
template<class T>
class my_shared_ptr {
    T* ptr;
    int* ref_counter;

public:
    my_shared_ptr(T* ptr);
    my_shared_ptr(my_shared_ptr<T>& other);

    ~my_shared_ptr();

    T& operator*();
};
```

lec13/shared-ptr.cpp

Shared Pointer

```
template<class T> my_shared_ptr<T>::my_shared_ptr(T* ptr)
    : ptr(ptr), ref_counter(new int(1)) {}
template<class T> my_shared_ptr<T>::my_shared_ptr(my_shared_ptr<T>&
    other)
    : ptr(other.ptr), ref_counter(other.ref_counter) {
    *ref_counter += 1;
}
template<class T> my_shared_ptr<T>::~my_shared_ptr() {
    if (*ref_counter > 1) {
        *ref_counter -= 1;
    } else {
        delete ref_counter;
        delete ptr;
    }
}
```


Part II

Part 2 – Move and Copy Semantics (in C++)

Assignment of Objects Holding Resources

- Recall `MyArray`
- What should the following code do?

```
MyArray array1(10);
MyArray array2 = array1;
```

- Remember `MyArray` structure

```
class MyArray {
    ulong size_p;
    int* data_p;
};
```

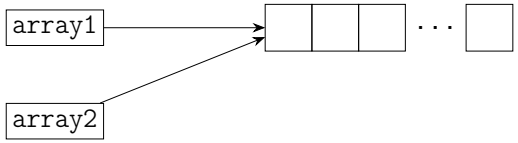
Assignment of Objects Holding Resources

```
class MyArray {
    ulong size_p;
    int* data_p;
};
```

- More specifically:
  What should happen to `data_p`?
- Multiple options:
  - Copy the pointer.
  - Allocate new array and copy data.
  - Copy the pointer, but invalidate original data.

Assignment of Objects Holding Resources

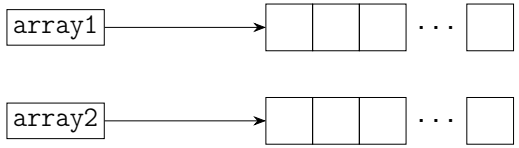
- Copy the pointer.



- PROBLEM: Which object handles deletion of the array.
- This is simmilar to the behavior of shared_ptr.

Assignment of Objects Holding Resources

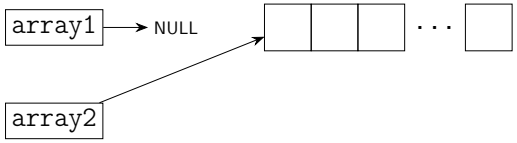
- Allocate new array and copy data.



- PROBLEM: Possible redundancy if array1 is about do be deleted (e.g. returning from function).

Assignment of Objects Holding Resources

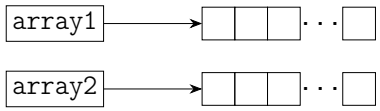
- Copy the pointer, but invalidate original data.



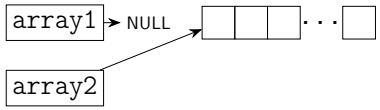
- PROBLEM: Original array becomes invalid.
- Similar to the behavior of `unique_ptr`.

Move and Copy Semantics

- **Copy:**



- **Move:**



Value Categories

- Each expression in C++ has a type and value category.
- **lvalue** – 'left value' (`L = r`)
  - *An expression whose evaluation determines the identity of an object or function* [1] – glvalue
  - Is not xvalue.
- **rvalue** – 'right value' (`l = R`)
  - *An expression whose evaluation computes the value of an operand of a built-in operator (such prvalue has no result object), or initializes an object.*[1] – prvalue
  - *Object whose resources can be reused.*[1] – xvalue

[1] en.cppreference.com/w/cpp/language/value_category

lvalue

- **lvalue** – 'left value' (`L = r`)
- Can be assigned to.
  - Variable name
  - Function/operator call whose value is a (lvalue) reference, such as the asignment operator `a = b`.
  - Pre-increment/decrement `++i`, `--i`.
  - Indirection `*p`.
  - Subscript `a[i]`.
  - and more[1]

[1] en.cppreference.com/w/cpp/language/value_category

rvalue

- **rvalue** – 'right value' (`l = R`)
- Cannot be assigned to.
  - Function/operator call whose value is non-reference.
  - Post-increment/decrement `i++`, `i--`.
  - All built in arithmetic operators `a + b`, `a % b`, ...
  - Addres-of expression `&a`;
  - `std::move(T)`
  - And more[1]

[1] en.cppreference.com/w/cpp/language/value_category

rvalue / lvalue reference

- `lvalue` reference `T&`
  - Alias to an existing object.
  - Can be initialized by an `lvalue`.
- `rvalue` reference `T&&`
  - Extend lifetime of temporary object.[1] e.g. result of an operator

```
std::string s = "hello";
std::string&& r = s + s;
```

  - Can be initialized by an `rvalue`.

[1] en.cppreference.com/w/cpp/language/value_category

Copy Semantics

- Copy constructor: `T(const T&)`
  - Constructs object as a copy of another object.
- Copy assignment: `T& operator=(const T&)`
  - Copies an object in another object
  - Frees resources previously owned by the modified object.
- Any resources required by an object fo a given instance must be acquired.

Move Semantics

- Move constructor: `T(const T&&)`
  - Constructs an object using resources of another object.
- Move assignment: `T& operator=(T&&)`
  - Moves an object into another.
  - Ownership of resources is transferred.
  - Frees resources previously owned by the modified object.
- No new resources are allocated.
- It is assumed the source object will be destroyed after the move.

Copy Semantics of MyArray

```
MyArray::MyArray(const MyArray& other)
    : size_p(other.size_p), data_p(new int[size_p])
{
    std::cout << "MyArray(&)" << '\n';
    for(int i = 0; i < size_p; ++i) {
        data_p[i] = other.data_p[i];
    }
}
MyArray& MyArray::operator=(const MyArray& other) {
    std::cout << "MyArray operator=(&)" << '\n';
    delete[] data_p;
    size_p = other.size_p;
    data_p = new int[size_p];
    for(int i = 0; i < size_p; ++i) {
        data_p[i] = other.data_p[i];
    }
    return *this;
}
```


Move Semantics of MyArray

```
MyArray::MyArray(MyArray&& other)
    : size_p(other.size_p), data_p(other.data_p)
{
    std::cout << "MyArray(&&)" << '\n';
    other.size_p = 0;
    other.data_p = nullptr;
}
MyArray& MyArray::operator=(MyArray&& other) {
    std::cout << "MyArray operator=(&&)" << '\n';
    delete[] data_p;
    size_p = other.size_p;
    data_p = other.data_p;
    other.size_p = 0;
    other.data_p = nullptr;
    return *this;
}
```


Summary of the Lecture

Topics Discussed

- Resouce Acquision-Release pattern.
- RAII using automatic destructor call
- Example RAII array wrapper
- RAII handlig of other resources
  - Files
  - Mutexes
  - Threads
  - Smart pointers
- Assignment of object with resources.
- `lvalue` and `rvalue`
- `lvalue` reference and `rvalue` reference
- Move and copy semantics