RESOURCE MANAGEMENT

RAII & Rule of Five

Florian Warg, Max Staff

May 18, 2017

WHAT ARE RESOURCES?

- a resource can be anything that exists in limited supply
- allocated heap memory
- thread, locked mutex, …
- open socket, open file, …
- disk space, database connection, …

- system resources are usually obtained via low level C APIs
- management involves 2 functions
  1. `acquire();`
  2. `release();`

```
char msg[] = "100: [error] bugs are everywhere!\n";
/* acquire() */
int fd = open("/tmp/log.txt", O_WRONLY | O_APPEND);
/* use resource */
write(fd, msg, sizeof(msg));
/* release() */
close(fd);
```

LEAKING RESOURCES

- it is easy to forget about releasing resources
- handle to acquired resource may be lost
- -> resource leak

```
{
    int arrSize;
    cin >> arrSize;
    int* arr = new int[arrSize];
    for (int i = 0; i < arrSize; ++i) {
        arr[i] = i;
    }
}
/* resource leaked here */
```

LEAKING RESOURCES II: DEADLOCK

```
int sum = 0;
mutex m;
void f() { m.lock(); sum += 2; m.unlock(); }
void g() { m.lock(); sum += 10; }
thread t1(f); thread t2(g);
t1.join(); t2.join();
```

- if t2 starts execution before t1, it will cause a deadlock
- even worse: t2 might only **sometimes** start before t1
- very hard to debug because it is not always reproducable

- Resource Acquisition Is Initialization
- bind the life cycle of resources to the life time of an object
- resources are acquired during object construction
- resources are released during object destruction
- -> resource becomes an invariant of the object

RAII WITH FILE EXAMPLE

```
class File {
private:
    int fd;
public:
    File(string path) {
        fd = open(path.c_str(), O_RDONLY);
    }
    ~File() { close(fd); }
    ssize_t write(void* buf, size_t size) { ... }
    ssize_t read(void* buf, size_t size) { ... }
};
/* ... */
{
    File f("/tmp/log.txt"); // resource acquired
    char msg[] = "101: [success] Way better!\n";
    f.write(msg, sizeof(msg));
} // resource released
```

RAII WITH MUTEX EXAMPLE

```
class LockGuard {
private:
    mutex& mtx;
public:
    LockGuard(mutex& mtx) : mtx(mtx) { mtx.lock(); }
    ~LockGuard() { mtx.unlock(); }
};
mutex m;
int sum = 0;
void f() { LockGuard guard(m); sum += 2; }
void g() { LockGuard guard(m); sum += 10; }
int main() {
    thread t1(f); thread t2(g);
    t1.join(); t2.join();
}
```

RAII WITH HEAP STORAGE EXAMPLE

```
class HeapArray {
private:
    int* arr;
    size_t size;
public:
    HeapArray(size_t size)
        : arr(new int[size]), size(size) {}
    ~HeapArray() { delete[] arr; }
    int& operator[](size_t i) { return arr[i]; }
};
/* ... */
{
    HeapArray a(1000); // resource acquired
    a[512] = 42;
} // resource released
```

- resource management introduces a new problem
- the user does not know that resources are managed
- the user expects every class to behave like a fundamental data type
- reference semantics should only apply to pointers and references

```
HeapArray a(1000);
a[512] = 42;
HeapArray b = a; // user wants to make a copy
b[512] = 0; // user wants to edit 2nd array
cout << "a[512] = " << a[512] << "\n"; // ???
```

EVEN MORE PROBLEMATIC BEHAVIOR

- when making copies, resources are not an invariant anymore
- this **will** introduce bugs

```
void f(HeapArray a) { cout << a[0] << "\n"; }
int main() {
    HeapArray x(10);
    f(x);
    x[3] = 42; // ???
}
```

A class that manages its own resources, must provide 5 special functions in order to implement the semantics of fundamental types

1. destructor
2. copy constructor
3. copy assignment operator
4. move constructor
5. move assignment operator
6. (functions to make type swappable)

DESTRUCTOR

- release resources in reverse order of construction

```
/* ctor */
HeapArray(size_t size)
    : arr(new int[size]), size(size) {}
/* dtor */
~HeapArray() { delete[] arr; }
```

- construct a new object with the same value

```
HeapArray(const HeapArray& rhs)
    : arr(new int[rhs.size]), size(rhs.size) {
    copy(rhs.arr, rhs.arr + size, arr);
}
```

COPY ASSIGNMENT OPERATOR

- assign value of other object to this object
- usually implemented in terms of copy ctor

```
HeapArray& operator=(HeapArray rhs) {
    swap(*this, rhs); // no throw
    return *this;
}
```

- any lvalue or rvalue of `T` can be swapped with an lvalue or rvalue of some other type
- swapping is done by calling unqualified `swap()` in context where `std::swap` and user-defined `swap` are visible

```
struct MyT {
    int x;
    friend void swap(MyT& lhs, MyT& rhs);
};
void swap(MyT& lhs, MyT& rhs) {
    using std::swap;
    swap(lhs.x, rhs.x);
}
```

- construct new object by moving state from other object
- ”hand over pointer instead of copying big objects”
- other object is left empty (but in valid state)

```
HeapArray(HeapArray&& rhs)
    : arr(move(rhs.arr)), size(move(rhs.size)) {
    rhs.arr = nullptr;
}
```

- move internal state of object to existing object
- ”hand over pointer instead of copying big objects”
- other object is left empty (but in valid state)

```
HeapArray& HeapArray::operator=(HeapArray&& rhs) {
    arr = move(rhs.arr);
    size = move(rhs.size);
    rhs.arr = nullptr;
}
```

RULE OF ZERO

- assume you compose a new data type
- if all of the member types conform to RAII and no further resources are required, the new type conforms to RAII as well
- -> in high level C++ none of the RO5 functions have to be implemented
- the compiler-generated implementation is sufficient
- this is called the **Rule of Zero**

```
struct MyType {
    string str;
    vector<int> vec;
}; // string and vector conform to RAII
MyType x;
MyType y = x; // implicit copy ctor
MyType z = std::move(z); // implicit move assign
```