Effective memory management techniques in modern C++

Δ-course #3

2015, March 11
Dmitry Budaragin

Index

— copy elision (pass and return by value)
— value categories
— move
— move in STL
— raw pointers
— smart pointers

Have you seen this code?

```
Document* parse(std::istream& input);

std::vector<Document*>* parse_collection(std::istream& input);
```

Return by value whenever possible

Copy elision

§12.8.31.1

```
Document parse(std::istream& input) {
    Document doc;
    // ...
    return doc;
}
```

```
Document doc = parse(stream);
```

```
void parse(std::istream& input, Document* p) {
    // construct directly at *p
}
```

```
Document doc;
parse(stream, &doc);
```

Copy elision (cont.)

§12.8.31.3

```
Document parse(std::istream& input);
Document canonicalize(Document document);

auto valid_document = canonicalize(parse(input));
```

Passing arguments

Function does not modify the argument — pass by const ref.

```
template <typename T>
bool has_duplicates(std::vector<T> const& v) {
    std::vector<T> tmp(v);
    std::sort(std::begin(tmp), std::end(tmp));
    return std::unique(std::begin(tmp), std::end(tmp)) != std::end(tmp);
}
```

Passing by value is never slower than passing by const reference and copying

Passing arguments (cont.)

Function does not modify the argument — consider passing by const ref.

```
template <typename T>
bool has_duplicates(std::vector<T> v) {
    std::sort(std::begin(v), std::end(v));
    return std::unique(std::begin(v), std::end(v)) != std::end(v);
}
```

```
std::vector<Thing> make_things();
bool result = has_duplicates(make_things());
```

Value categories

§3.10

— lvalue *(left)*

— xvalue *(expiring)*

— prvalue *(pure right)*

Non-trivial resources (spot the bug)

(you might want to stick to standard containers in most cases)

```
class Buffer {
public:
    Buffer(size_t size): size(size) {
        data = new char[size];
    }
    ~Buffer() {
        delete[] data;
    }
    Buffer(Buffer const& b): size(b.size), data(b.data) {}
private:
    size_t size;
    char* data;
};
```

Non-trivial resources

(you might want to stick to standard containers in most cases)

```
class Buffer {
public:
    Buffer(size_t size): size(size) {
        data = new char[size];
    }
    ~Buffer() {
        delete[] data;
    }
    Buffer(Buffer const& b): size(b.size) {
        data = new char[size];
        memcpy(data, b.data, size);
    }
private:
    size_t size;
    char* data;
};
```

What if we could 'steal' the data?

Enter move ctor/assignment

§12.8.3

```
Buffer(Buffer&& b): data(b.data), size(b.size) {
    b.data = nullptr;
    b.size = 0;
}

Buffer& operator= (Buffer&& b) {
    data = b.data;  b.data = nullptr;
    size = b.size;  b.size = 0;
    return *this;
}
```

Enter move ctor/assignment

§12.8.3

```
Buffer(Buffer&& b): data(b.data), size(b.size) {
    b.data = nullptr;
    b.size = 0;
}

Buffer& operator= (Buffer&& b) {
    std::swap(data, b.data);
    std::swap(size, b.size);
    return *this;
}
```

std::move

```
Buffer f() {
    // ...
    Buffer buf(size);
    // ...
    return buf;
}
```

```
Buffer::Buffer(AnotherBuffer&& ab)

Buffer g() {
    AnotherBuffer buf;
    // ...
    return std::move(buf);
}
```

std::move (cont.)

(Informally)

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

std::move (cont.)

in constructors

```
struct User {
    std::string first_name;
    std::string last_name;

    User(std::string const& first, std::string const& last):
        first_name(first), last_name(last) {}

    User(User&& u):
        first_name(std::move(u.first_name)),
        last_name(std::move(u.last_name)) {}
};
```

STL

was revised for move semantics

```
std::vector<User> users;
users.push_back(User("John", “Smith"));
```

```
users.emplace_back("Mary", “Jane");
```

```
template <typename... Args>
void emplace_back(Args&&... args);
```

STL (cont.)

not everything is so smooth

Item 1 → Item 1’

Item 2 → Item 2’

Item 3 → exception

Item 4

```
std::move_if_noexcept();

T::T(T&&) noexcept;
```

noexcept

use carefully

```
template <typename T1, typename T2>
struct pair {
   void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                               noexcept(swap(second, p.second)));
   …
};
```

raw pointers

are evil?

1. A single object or an array?
2. Should you destroy it?
3. How to destroy?
4. delete or delete[]?
5. How to ensure that deletion is done exactly once?
6. Does it dangle?

raw pointers (cont.)

are hard to manage

```
void process() {
    auto thing = new Thing();
    // do stuff (1)
    auto another = new Another();
    // do stuff (2)
    delete thing;
    delete another;
}
```

Use std::unique_ptr for exclusive-ownership resource management

std::unique_ptr

```
void process_better() {
    auto thing = std::make_unique<Thing>();
    // do stuff (1)
    auto another = std::make_unique<Another>();
    // do stuff (2)
}
```

```
std::unique_ptr<Document> factory() {
    if (/* some condition */ true) {
        return std::make_unique<XMLDocument>();
    }
    else {
        return std::make_unique<JSONDocument>();
    }
}
```

std::unique_ptr (cont.)

```
auto delete_document = [](Document* doc) {
    std::cerr << "log" << std::endl;
    delete doc;
};

std::unique_ptr<Document, decltype(delete_document)> doc;
```

Use std::shared_ptr for shared-ownership resource management

std::shared_ptr

```
auto delete_document = [](Document* doc) {
    std::cerr << "log" << std::endl;
    delete doc;
};

std::shared_ptr<Document> doc(new XMLDocument(), delete_document);
```

Intel