MODERN C++

MOVE SEMANTICS

CODERS SCHOOL

ŁUKASZ ZIOBROŃ

AGENDA

- intro
- r-values and l-values
- move constructor and move assignment operator
- implementation of move semantics
- rule of 0, 3, 5
- `std::move()`
- forwarding reference
- reference collapsing
- `std::forward()` and perfect forwarding
- copy elision, RVO (return value optimisation)
- recap

SOMETHING ABOUT YOU

- What you don't like in C++?
- What other programming languages do you know?

ŁUKASZ ZIOBROŃ

NOT ONLY A PROGRAMMING XP

- Front-end dev, DevOps & Owner @ Coders School
- C++ and Python developer @ Nokia & Credit Suisse
- Team leader & Trainer @ Nokia
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Web developer (HTML, PHP, CSS) @ StarCraft Area

EXPERIENCE AS A TRAINER

- C++ online course @ Coders School
- Company trainings @ Coders School
- Practical Aspects Of Software Engineering @ PWr & UWr
- Nokia Academy @ Nokia

PUBLIC SPEAKING EXPERIENCE

- code::dive conference
- code::dive community
- Academic Championships in Team Programming
- Coders School YouTube channel

HOBBIES

- StarCraft Brood War & StarCraft II
- Motorcycles
- Photography
- Archery
- Andragogy

CONTRACT

- 🎰 Vegas rule
- 🗣 Discussion, not a lecture
- ☕ Additional breaks on demand
- ⌚ Be on time after breaks

PRE-TEST 📝

QUESTION 1/2

We have only the below template function defined. What will happen in each case? Which example will compile and display "OK"?

```
template <typename T>
void foo(T && a) {std::cout << "OK\n"; }

int a = 5;
```

1. `foo(4);`
2. `foo(a);`
3. `foo(std::move(a));`

QUESTION 2/2

What will be printed on the screen?

```
class Gadget {};
void f(const Gadget&) { std::cout << "const Gadget&\n"; }
void f(Gadget&)       { std::cout << "Gadget&\n"; }
void f(Gadget&&)      { std::cout << "Gadget&&\n"; }

template <typename Gadget>
void use(Gadget&& g) { f(g); }

int main() {
    const Gadget cg;
    Gadget g;
    use(cg);
    use(g);
    use(Gadget());
}
```

MOVE SEMANTICS

RATIONALE

- Better optimization by avoiding redundant copies
- improved safety by keeping only one instance

NEW SYNTAX ELEMENTS

- `auto && value` - r-value reference
- `Class(Class &&)` - move constructor
- `Class& operator=(Class&&)` - move assignment operator
- `std::move()` auxilary function
- `std::forward()` auxilary function

R-VALUE AND L-VALUE

```
struct A { int a, b; };

A foo() { return {1, 2}; }

A a;                         // l-value
A{5, 3};                     // r-value
foo();                       // r-value
```

R-VALUE AND L-VALUE

- l-value object has a name and address
- l-value object is persistent, in the next line it can be accessed by name
- r-value object does not have a name (usually) or address
- r-value object is temporary, in the next line it will not be accessible

R-VALUE AND L-VALUE REFERENCES

```
struct A { int a, b; };
A foo() { return {1, 2}; }

A a;                        // l-value
A{5, 3};                    // r-value
foo();                      // r-value

A & ra = a;                 // l-value reference to l-value, OK
A & rb = foo();             // l-value reference to r-value, ERROR
A const& rc = foo();        // const l-value reference to r-value, OK (exception)

A && rra = a;               // r-value reference to l-value, ERROR
A && rrb = foo();           // r-value reference to r-value, OK

A const ca{20, 40};
A const&& rrc = ca;         // const r-value reference to const l-value, ERROR
```

R-VALUE OR L-VALUE?

```
str1 += str2                        // l-value
str1 + str2                         // r-value
[](int x){ return x * x; };         // r-value
std::move(a);                       // r-value
int && a = 4;                       // 4 is an r-value
```

R-VALUE REFERENCE IS... L-VALUE?

```
int && a = 4;
```

- `4` is r-value
- `a` is r-value reference
- name `a` itself is an l-value (has an address, can be referenced lated)
- but let's not think about it now 😉

VALUE CATEGORIES IN C++

- lvalue
- prvalue
- xvalue
- glvalue = lvalue | xvalue
- rvalue = prvalue | xvalue

Full list at cppreference.com

USAGE OF MOVE SEMANTICS

```cpp
template <typename T>
class Container {
public:
    void insert(const T& item);  // inserts a copy of an item
    void insert(T&& item);       // moves item into the container
};

Container<std::string> c;
std::string str = "text";

c.insert(str);                 // lvalue -> insert(const std::string&)
                               // inserts a copy of str, str is used later
c.insert(str + str);           // rvalue -> insert(string&&)
                               // moves temporary into the container
c.insert("text");              // rvalue -> insert(string&&)
                               // moves temporary into the container
c.insert(std::move(str));      // rvalue -> insert(string&&)
                               // moves str into the container, str is no longer used
```

PROPERTIES OF MOVE SEMANTICS

- Transfer all data from the source to the target
- Leave the source object in an unknown, but safe to delete state
- The source object should never be used
- The source object can only be safely destroyed or, if possible, a new resource can be assigned to it (eg. `reset()`)

```
std::unique_ptr<int> pointer1{new int{5}};
std::unique_ptr<int> pointer2 = std::move(pointer1);
*pointer1 = 4;  // Undefined behaviour, pointer1 is in the moved-from state
pointer1.reset(new int{20});    // OK
```

IMPLEMENTATION OF MOVE SEMANTIC

```
class X : public Base {
    Member m_;

    X(X&& x) : Base(std::move(x)), m_(std::move(x.m_)) {
        x.set_to_resourceless_state();
    }

    X& operator=(X&& x) {
        Base::operator=(std::move(x));
        m_ = std::move(x.m_);
        x.set_to_resourceless_state();
        return *this;
    }

    void set_to_resourceless_state() { /* reset pointers, handlers, etc. */ }
};
```

IMPLEMENTATION OF MOVE SEMANTIC

USUAL IMPLEMENTATION

```
class X : public Base {
    Member m_;

    X(X&& x) = default;
    X& operator=(X&& x) = default;
};
```

TASK

Write your implementation of `unique_ptr`

Aim: learn how to implement move semantics with manual resource management

HINTS

- Template class
- RAII
- Copy operations not allowed
- Move operations allowed
- Interface functions - at least:
  - `T* get() const noexcept`
  - `T& operator*() const`
  - `T* operator->() const noexcept`
  - `void reset(T* = nullptr) noexcept`

RULE OF 3

If you define at least one of:

- destructor
- copy constructor
- copy assignment operator

it means that you are manually managing resources and you should implement them all.

It will ensure correctness in every context.

RULE OF 5

Rule of 5 = Rule of 3 + optimizations

- destructor
- copy constructor
- copy assignment operator
- move constructor
- move assignment operator

From C++11 use Rule of 5.

RULE OF 0

Do not implement any of Rule of 5 functions 😎

If you use RAII handlers (like smart pointers), all the copy and move operations will be generated (or deleted) implicitly.

For example, when you have a `unique_ptr` as your class member, copy operations of your class will be automatically blocked, but move operations will be supported.

TASK

Aim: learn how to refactor code to use RAII and Rule of 0

Write a template class that holds a pointer

- use a raw pointer to manage the resource of a template type
- implement constructor to acquire a resource
- implement the Rule of 3
- implement the Rule of 5
- implement the Rule of 0
  - use a roper smart pointer instead of the raw pointer

IMPLEMENTATION OF `std::move()`

"UNIVERSAL REFERENCE"

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

- `T&&` as a template function parameter is not only r-value reference
- `T&&` is a "forwarding reference" or "universal reference" (name proposed by Scott Meyers)
- `T&&` in templates can bind to l-values and r-values
- `std::move()` takes any kind of reference and cast it to r-value reference
- `std::move()` convert any object into a temporary, so that it can be later matched by the compiler to be passed by an r-value reference

REFERENCE COLLAPSING RULES

- `T& & -> T&`
- `T& && -> T&`
- `T&& & -> T&`
- `T&& && -> T&&`

REFERENCE COLLAPSING

When a template is being instantiated reference collapsing may occur

```
template <typename T>
void f(T & item) {}     // takes item always as an l-value reference

void f(int& & item);    // passing int& as a param, like f(a) -> f(int&)
void f(int&& & item);   // passing int&& as a param, like f(5) -> f(int&)
```

```
template <typename T>
void g(T && item) {}    // takes item as a forwarding reference

void g(int& && item);   // passing int& as a param, like g(a) -> f(int&)
void g(int&& && item); // passing int&& as a param, like g(5) -> f(int&&)
```

INTERFACE BLOAT

Trying to optimize for every possible use case may lead to an interface bloat.

```cpp
class Gadget;
void f(const Gadget&)     { std::cout << "const Gadget&\n"; }
void f(Gadget&)           { std::cout << "Gadget&\n"; }
void f(Gadget&&)          { std::cout << "Gadget&&\n"; }
void use(const Gadget& g) { f(g); }              // calls f(const Gadget&)
void use(Gadget& g)       { f(g); }              // calls f(Gadget&)
void use(Gadget&& g)      { f(std::move(g)); } // calls f(Gadget&&)

int main() {
    const Gadget cg;
    Gadget g;
    use(cg);       // calls use(const Gadget&) then calls f(const Gadget&)
    use(g);        // calls use(Gadget&) then calls f(Gadget&)
    use(Gadget()); // calls use(Gadget&&) then calls f(Gadget&&)
}
```

TASK

Improve the `use()` function to catch more types of references to have fewer overloads.

SOLUTION: PERFECT FORWARDING

Forwarding reference `T&&` + `std::forward()` is a solution to interface bloat.

```cpp
class Gadget;

void f(const Gadget&) { std::cout << "const Gadget&\n"; }
void f(Gadget&)       { std::cout << "Gadget&\n"; }
void f(Gadget&&)      { std::cout << "Gadget&&\n"; }

template <typename Gadget>
void use(Gadget&& g) {
    f(std::forward<Gadget>(g)); // forwards original type to f()
}

int main() {
    const Gadget cg;
    Gadget g;
    use(cg);         // calls use(const Gadget&) then calls f(const Gadget&)
    use(g);          // calls use(Gadget&) then calls f(Gadget&)
    use(Gadget()); // calls use(Gadget&&) then calls f(Gadget&&)
}
```

std::forward

Forwarding reference (even bind to r-value) is treated as l-value inside a template function.

```
template <typename T>
void use(T&& t) {
    f(t);                        // t treated as l-value unconditionally
}
```

```
template <typename T>
void use(T&& t) {
    f(std::move(t));             // t treated as r-value unconditionally
}
```

```
template <typename T>
void use(T&& t) {                // forwards t as r-value if r-value was passed,
    f(std::forward(t));          // forwards as l-value otherwise
}
```

In other words, `std::forward()` restores the original reference type.

COPY ELISION

- omits copy and move constructors
- results in zero-copy pass-by-value semantics

MANDATORY COPY ELISION FROM C++17

```
T f() {
    return T();
}
f();                // only one call to default c-tor of T
T x = T(T(f())); // only one call to default c-tor of T, to initialize x
```

- in the `return` statement, when the object is temporary (RVO - Return Value Optimisation)
- in the initialization, when the initializer is of the same class and is temporary

Do not try to "optimize" code by writing `return std::move(sth);`. It may prevent optimizations.

Copy elision on cppreference.com

RVO AND NRVO

```
T f() {
    T t;
    return t;   // NRVO
}
```

- NRVO = Named RVO
- RVO is mandatory from C++17, NRVO not

```
T bar()
{
    T t1{1};
    T t2{2};
    return (std::time(nullptr) % 2) ? t1 : t2;
}  // don't know which object will be elided
```

RVO and NRVO on cpp-polska.pl

KNOWLEDGE CHECK 🙂

TEMPLATE TYPE DEDUCTION

```cpp
template <typename T>
void copy(T arg) {}

template <typename T>
void reference(T& arg) {}

template <typename T>
void universal_reference(T&& arg) {}

int main() {
    int number = 4;
    copy(number);          // int
    copy(5);               // int
    reference(number);     // int&
    reference(5);          // candidate function [with T = int] not viable: expects an l-v
    universal_reference(number);                // int&
    universal_reference(std::move(number));     // int&&
    universal_reference(5);                     // int&&
}
```

KNOWLEDGE CHECK 🤯

```cpp
void foo(int && a);          // r
void foo(int & a);           // l

int a = 5;
```

Which of above functions will be called by below snippets?

- `foo(4);`
  - r
- `foo(a);`
  - l
- `foo(std::move(a));`
  - r
- `foo(std::move(4));`
  - r (move is redundant)

KNOWLEDGE CHECK 🤯

```
template <typename T>
void foo(T && a);          // r

template <typename T>
void foo(T & a);           // l

int a = 5;
```

Which of above functions will be called by below snippets?

- `foo(4);`
  - r
- `foo(a);`
  - l
- `foo(std::move(a));`
  - r

KNOWLEDGE CHECK 🤯

```
template <typename T>
void foo(T && a);           // r

int a = 5;
```

What will happen now?

- `foo(4);`
  - r
- `foo(a);`
  - r
- `foo(std::move(a));`
  - r

PRE-TEST ANSWERS 📝

QUESTION 1/2

We have only the below template function defined. What will happen in each case? Which example will compile and display "OK"?

```cpp
template <typename T>
void foo(T && a) {std::cout << "OK\n"; }

int a = 5;
```

- `foo(4);`
  - "OK"
- `foo(a);`
  - "OK"
- `foo(std::move(a));`
  - "OK"

QUESTION 2/2

What will be printed on the screen?

```
class Gadget {};
void f(const Gadget&) { std::cout << "const Gadget&\n"; }
void f(Gadget&)       { std::cout << "Gadget&\n"; }
void f(Gadget&&)      { std::cout << "Gadget&&\n"; }

template <typename Gadget>
void use(Gadget&& g) { f(g); }

int main() {
    const Gadget cg;
    Gadget g;
    use(cg);
    use(g);
    use(Gadget());
}
```

- const Gadget&
- Gadget&
- Gadget&

RECAP

Mention as many keywords/topics from this session as you can

- r-value and l-value referencesss
- Move constructor and move assignment operator
- RAII
- Rule of 0, 3, 5
- `std::move()` and `std::forward()`
- Forwarding reference
- Reference collapsing
- Perfect forwarding
- Copy elision, RVO

POST-WORK

If you wish to practice more on move semantics and resource management, try to implement `shared_ptr`. You can even try to make it thread-safe 😊 Send me a link to your repo to lukasz@coders.school if you wish to have a code review.

POST-TEST

Please take this quiz (10-15 min) about 2-5 days after the training. It will help you recall this session and make it last a little bit longer in your memory.

EVALUATION

Please fill in the survey about this training (5-10 min) now. It will help me understand how can I improve this session in future.

CODERS SCHOOL