

KIV/CPP – Programming in C++

06. Copy and move semantics, templates

Martin Úbl

KIV ZČU

2025/2026

- *l-value*
- “locator value”, sometimes “left-hand side value”
- all identifiers that denote a specific place in memory
- “from the code’s point of view”; physically, other objects also have some place in memory, we just cannot identify it by name
- we can assign a value to an *l-value*
 - the exception is `const` variables, but even those are *l-values*

```
int a = 5;           // a is an l-value
std::string b;       // b is an l-value
MyClass c;           // c is an l-value
```

l-value, r-value

- *r-value*
- “right-hand side value”
- temporary and transient objects
- (syntactically) they do not have addressable memory – they have no identifier
- they cannot be assigned to

```
int a = 5;
    // a is an l-value, 5 is an r-value
std::string b = "hi";
    // b is an l-value, "hi" is an r-value
float c = get_c();
    // get_c() is (returns) an r-value

5 = c; // ouch!
```

- *l-value* reference
- a reference to an *l-value*
- the classic reference we have worked with
- a non-const one cannot be bound to an *r-value*

```
int a = 5;  
int& ra = a; // ok  
  
int& rb = 5; // ouch!
```

l-value, r-value

- const *l-value* reference
- also a reference to an *l-value*
- when binding to an *r-value*, thanks to const, the compiler converts it to an *l-value*

```
const int& rb = 5; // ok
```

```
void my_func(const int& a) {  
    ...  
}
```

```
...  
my_func(a); // ok  
my_func(10); // ok
```

- *r-value* reference
- a reference to an *r-value*
- cannot be bound to an *l-value*
- `std::move`
- it has the same lifetime as a regular reference, but does not have to be `const` for an *r-value*
- introduced by `&&`
- enables move semantics to work

```
int&& rb = 5; // ok
```

```
void my_func(int&& a) {  
    ...  
}
```

```
my_func(a)      // ouch!  
my_func(10);    // ok
```

- *r-value* reference and overloading
- allows temporary and “permanent” instances to be distinguished

```
void my_func(int&& a) {  
    // for temporaries  
}  
  
void my_func(const int& a) {  
    // for permanent ones  
}  
  
my_func(a)      // ok  
my_func(10);    // ok
```

l-value and r-value context

- the context of a method call on an object can be distinguished – when calling a method on an *l-value* than on an *r-value*
- introduced by `&` for an *l-value* and `&&` for an *r-value* after the method signature

```
class Test {  
    std::string GetContext() & {  
        return "l-value";  
    }  
    std::string GetContext() && {  
        return "r-value";  
    }  
};
```

```
a.GetContext();           // l-value  
Test().GetContext();      // r-value
```

“C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.”

- Bjarne Stroustrup

“There are only two kinds of languages: the ones people complain about and the ones nobody uses.”

- Bjarne Stroustrup

Copy and move semantics

- an object's lifetime should also be delimited semantically
- copy semantics
 - “clones” the source
 - during construction or by assignment
- move semantics
 - “moves” the source
 - during construction or by assignment
- examples:
 - `std::unique_ptr` – move only
 - `std::thread` – move only
 - `std::ostream` and `std::istream` – move only
 - STL containers – both move and copy (if the elements support them!)
- rather than a set of exact rules, this is a semantic distinction of context
- it can express more clearly who owns which object

Copy and move semantics

- copy constructor and copy assignment
- an object can be copied while creating another one
- or by assigning to an already existing object
- does not modify the source object

```
// copy constructor
```

```
MyClass(const MyClass& o) : mValue(o.mValue) {  
}
```

```
// copy assignment operator
```

```
MyClass& operator=(const MyClass& other) {  
    mValue = other.mValue;  
    return *this;  
}
```

Copy and move semantics

- move constructor and move assignment
- an object can be moved while creating another one
- or by assigning to an already existing object
- modifies the source object (“empties” it)

```
// move constructor
```

```
MyClass(MyClass&& o) : mValue(o.mValue) {  
    o.mValue = 0;  
}
```

```
// move assignment operator
```

```
MyClass& operator=(MyClass&& other) {  
    mValue = other.mValue;  
    other.mValue = 0;  
    return *this;  
}
```

Copy and move semantics

- copy construction

```
MyClass a(other);
```

- move construction

```
MyClass b(std::move(other));
```

- copy assignment

```
a = other;
```

- move assignment

```
b = std::move(other);
```

- if we intend to express ownership, there are rules
- *Rule of zero*
 - we do not define move/copy constructors and operators, or a destructor, ourselves
 - we use existing classes that do this for us
 - e.g. `std::unique_ptr`
- *Rule of five*
 - we define everything:
 - copy constructor
 - copy assignment operator
 - move constructor
 - move assignment operator
 - destructor

“If you think it’s simple, then you have misunderstood the problem.”
- Bjarne Stroustrup

Templates

- templates
- an element of generic programming
- allows an implementation to be generalized for multiple parameters
- code generation at compile time
- parameterization:
 - by type
 - by value
 - by a variable list (so-called *variadic templates*)
- the `template` and `typename` keywords and angle brackets
- template functions, methods, classes, `constexpr` expressions, ...

```
template<typename T>
void DoSomethingGeneric(const T& a) {
    ...
}
```

- we have already encountered them
 - `std::array<int, 5>`
 - `std::vector<int>`
 - `std::function<void()>`
 - ...
- these classes are generic
- they do not know in advance what type they should wrap or represent internally
- the decision is made only at the moment of so-called *specialization*
 - explicit or implicit specification of template parameters
 - “a copy of the code is generated” for the given set of parameters

```
std::array<int, 5> arr;
```

- it is specialized for `int` and size 5

Templates

- parameterization by type
- the typename or class keyword (equivalent)

```
template<typename T>
void PrintLine(const T& something) {
    std::cout << "Something: " << something
               << std::endl;
}
```

```
PrintLine("Tree fiddy");
PrintLine(3.50);
PrintLine<int>(999);
PrintLine<Monster>(lochNess);
```

Templates

- parameterization by value

```
template<size_t Dim>
class Vector {
    private:
        std::array<double, Dim> mCoords;

        ...
};
```

- typically for some analytically defined classes that may differ by parameters
- changing the value essentially creates a new type

Templates

- parameterization by both type and value
- it can be parameterized by multiple types and values, and they can be combined

```
template<typename T, size_t Size>
std::array<T, Size> FillNatural() {

    std::array<T, Size> ret;
    std::generate(ret.begin(), ret.end(),
                  [n = 1]() mutable { return n++; });

    return ret;
}

auto arr = FillNatural<long, 1000>();
```

Templates

- generic types
- e.g. a stack (simplified) without the need for a common ancestor (as e.g., Java has `java.lang.Object`)

```
template<typename T>
class Stack {
    private:
        std::vector<T> mData;

    public:
        void push(const T& value);
        T& top() const;
        void pop();

        bool empty() const;
};
```

Templates

- explicit specialization of a function/method
- e.g. one exception to otherwise generic behavior

```
template<typename T>
void do_something(const T& value)
{
    //
}
```

```
template<>
void do_something<double>(const double& value)
{
    //
}
```

Templates

- explicit specialization of a method
- e.g. one exception to otherwise generic behavior of the class

```
template<typename T>
class Test {
public:
    void Do_This();
    void Do_That();
};
```

```
template<>
void Test<int>::Do_This() {
    //
}
```

Templates

- every type for which it is specialized must support everything performed in the method body
 - e.g. incrementing with the ++ operator – all types must have the ++ operator defined

```
template<typename T>
T& increment(T& val) {
    return val++;
}
```

```
int a = 5;
increment(a); // ok
```

```
std::string s{ "one_does_not_simply_increment"
               "a_string" };
increment(s); // ouch!
```

- conditional specialization
- `std::enable_if`
- allows the compiler to specify the conditions under which the template should be specialized
- conditions, e.g.:
 - `std::is_integral`
 - `std::is_floating_point`
 - `std::is_signed`
 - `std::is_pod`
 - `std::is_class`
 - and others...

Templates

- it can be cumbersome

```
template<typename T,  
        typename std::enable_if<  
            std::is_integral<T>::value  
            >::type* = nullptr>  
T& increment(T& val) {  
    val++;  
    return val;  
}
```

- it can be shortened a little
 - instead of `std::enable_if<...>::type`, write `std::enable_if_t<...>`
 - instead of `std::is_integral<T>::value`, write `std::is_integral_v<T>`

- still cumbersome

```
template<typename T,  
        typename std::enable_if_t<  
            std::is_integral_v<T>  
            >* = nullptr>  
T& increment(T& val) {  
    val++;  
    return val;  
}
```

- it essentially forces the specialization to fail

Templates

- it can be made a little clearer using *concepts*
- see later lectures

```
template<typename T>
concept TIntegral = std::is_integral_v<T>;

TIntegral auto& increment(TIntegral auto& val)
{
    val++;
    return val;
}
```

- constexpr templates

```
template<typename T>  
constexpr T PI = T(3.141592653589793238L);
```

- template lambda functions (since C++20)
- a template definition can be inserted into a lambda function after the *capture* block
- the rules are then essentially the same as for ordinary functions

```
auto multiply_2 = []<typename T>(T a) -> T {  
    return a + a;  
};
```

- note: beware – the more template instances, the more lambda functors, and therefore much more “local” code bloat

- template lambda functions (since C++20)
- what cannot be done with a generic lambda?

```
auto vf = []<typename T>(std::vector<T>& a) {  
    // something  
};
```

- this way, for example, a template parameter of a specific type can be defined and `std::vector` can be enforced at the input

Templates

- curiosities
- e.g. factorial by recursive template specialization

```
template <int N>
struct Factorial {
    static const int value =
        N * Factorial<N - 1>::value;
};
```

```
template <>
struct Factorial<0> {
    static const int value = 1;
};
```

- not a good idea – why?

- factorial by recursive template specialization
- not a good idea:
 - because factorial by recursion will never be a good idea (see PPA and others)
 - if e.g., $N = 10$, then 10 structures with one element are created
 - it must be evaluated at compile time – slowing down compilation
 - neither is propagated to the final binary – optimized out by compilation process
- it only demonstrates the genericity of templates
- technically speaking, the template system is Turing-complete

- what next?
- variadic templates
- perfect forwarding
- e.g. `emplace_back`
 - in-place insertion of an element, e.g. into a vector
 - then multiple copies are not needed
 - nor multiple `std::move` calls
- SFINAE
- concepts