Modern C++

Marcin Serwach

Łódź, 8.12.2016

C++11

- Move semantic
- Lambda
- Automatic type deduction
- Variadic templates
- Enum class
- Suffix
- Const expression
- initializer_list
- Attributes : *final*, *override*, *default*, *delete*
- Foreach
- Attribute noexcept
- Brackets : {}
- Calling constructor from constructor
- nullptr

C++11 STL

- Thread
- Smart pointers
- Chrono
- Random engines/distrubution
- Type traits
- Regex
- std : :array
- Tuple
- System errors
- Ratio
- *std : :bind, std : :function*

Move semantic

L-value reference

Alias name of object.

```
Class obj;
Class& lref = obj;
```

R-value reference

Reference to a temporary object, that will be destroyed in near future.

```
Class obj1, obj2;
Class&& rref1 = Class{}; // r-value as returned valued
Class&& rref2 = obj1 + obj2; // as above
Class&& rref3 = std::move(obj1); // calling std::move
void fun(Class&& aArg); // r-value as an argument of a
     method
/* ... */
fun(Class{}); // creating temporary object
```

R/L-value reference and function overloading

```
void fun(Class& aArg){
        std::cout<<"L-value\n";
}
void fun(Class&& aArg){
        std::cout<<"R-value\n";
}
int main() {
        Class a;
        fun(a); // Output: L-value
        fun(Class{}); // Output: R-value
        return 0;
}
```

Universal reference

Universal reference

Only for templates. Notation the same as R-value reference.

```
template<typename T>
void fun(T&& aArg){
}
```

The *fun* allows passing L-value and R-value reference.

Universal reference

```
template<typename T>
void fun(T&& aArg){
        std::cout<<"fun\n";
}

int main() {
        int a = 2, b = 1;
        int& lvalue = a;
        int&& rvalue = std::move(b);
        fun(lvalue);
        fun(rvalue);
        return 0;
}
```

```
int a = 2;
int&& rvalue1 = std::move(a);
```

The *std : :move* **doesn't** move anything. It is a **unconditional** cast to r-value reference. Moved object cannot be used.

```
template<typename T>
void fun(T&& aArg){
        anotherFun(std::forward<T>(aArg));
}
```

The *std : :forward* **doesn't** forward/move anything. It is a **conditional** cast to r-value reference. If a given argument of *forward* is r-value, it casts to r-value. Otherwise *forward* casts to l-value reference.

```
void anotherFun(Class& aArg){
        std::cout<<"L-value\n";
}
void anotherFun(Class&& aArg){
        std::cout<<"R-value\n";
}
template<typename T>
void fun(T&& aArg){
        anotherFun(aArg);
}
// ...
Class obj1;
fun(obj1); // Output: L-value
fun(Class{}); // Output: L-value
```

```
void anotherFun(Class& aArg){
        std::cout<<"L-value\n";
}
void anotherFun(Class&& aArg){
        std::cout<<"R-value\n";
}
template<typename T>
void fun(T&& aArg){
        anotherFun(std::forward<T>(aArg));
}
// ...
Class obj1;
fun(obj1); // Output: L-value
fun(Class{}); // Output: R-value
```

std : :forward - example explanation

```
void anotherFun(Class& aArg){
        std::cout<<"L-value\n";
}
void anotherFun(Class&& aArg){
        std::cout<<"R-value\n";
}
// ..
Class&& a = Class{};
anotherFun(a); // Output: L-value
```

R-value reference is passing to a method as L-value reference. To pass it as R-value reference you have to use *std : :move*.

Why it is so important ?

```cpp
void fun(Class&& aArg){
        std::cout<<"fun\n";
}
```

In method *fun* the argument is passing by r-value reference, what means that it doesn't have to be valid outside of it, because it is a temporary object and it will be destroyed while leaving the method. Therefore you can take everything from the argument, i. e. : replace pointers :

```cpp
void fun(Class&& aArg){
        this->iPointer = aArg.iPointer;
        aArg.iPointer = nullptr;
}
```

The argument **can't** be *const*.

Move constructor and move assignment operator

A move constructor is similar to a copy constructor with a small difference - an argument is passing by nonconst r-value reference.

```
Class(Class&& aArg) : iPointer{aArg.iPointer} {
  aArg.iPointer = nullptr;
}
```

A move assignment operator it is a R-value version of an assignment operator.

```
Class& operator=(Class&& aArg) {
  iPointer = aArg.iPointer;
  aArg.iPointer = nullptr;
  return *this;
}
```

Rule of 3 ; now it is Rule of 5

The Rule of 3 in C++11 is extended to the Rule of 5, because of R-value related methods. Therefore the following method should be implemented :

- Copy constructor
- Move constructor
- Assignment operator
- Move assignment operator
- Virtual destructor

What about setters?

```
void Class::setString(std::string aArg) {
  iString = std::move(aArg);
}
```

vs

```
void Class::setString(const std::string& aArg){
  iString = aArg;
}
void Class::setString(std::string&& aArg){
  iString = std::move(aArg);
}
```

What about setters?

By value

```
cl.setString("abc");
```

- constructor
- move assignment operator

```
std::string myString{"abc"};
cl.setString(myString);
```

- copy constructor
- move assignment operator

By reference

- constructor
- move assignment operator

- assignment operator

What about setters?

```
std::string myString{"abc"};
cl.setString(std::move(myString));
```

By value

- move constructor
- move assignment operator

By reference

- move assignment operator

Lambda

Lambda

```
// method
int fun(double aArg1, float aArg2){
  return static_cast<int>(aArg1 + aArg2);
}
 // pointer to method
 int (*ptrFun)(double, float) = fun;
 fun(1.2, 3.4f);

 // lambda
 int (*ptrFunLambda)(double, float) =
  [](double aArg1, float aArg2)->int { return
    static_cast<int>(aArg1 + aArg2);};
  ptrFunLambda(1.2, 3.4f);
```

Lambda structure - passing arguments

```
[passingArguments](ArgType1 arg1, ArgType2 arg1)
    mutable -> ReturnType {
        ReturnType value;
        return value;
}
```

[passingArguments] - Passing variables to lambda while creating lambda. Possible options :
& - all variables passing by reference,
= - all variables passing by value (copy constructor is called),
&, var1, var2 - *var1*, *var2* passing by value, other by reference,
=, &var1, &var2 - *var1*, *var2* passing by reference, other by value.

```
auto ptrRef = [=, &str2](int aArg1){ return str1.size
    () + str2.size() + str3.size() + aArg1;};
}
```

Lambda structure - mutable

```cpp
int value = 0;
auto ptrRef = [=]() mutable { ++value;};
ptrRef();
```

mutable - it is required if *value* is passed by value.

Lambda structure - return type

```
double value = 12.345;
auto lambda1 = [&]() -> int { return value;};
std::cout<<lambda1();

double value1 = 12.345;
double value2 = 54.321;
auto lambda2 = [&]() -> decltype(value1 + value2) {
    return value1 + value2;};
std::cout<<lambda2();
```

Lambda with STL

```
std::vector<int> v{1, 2, 3, 4};
const auto it = std::find_if(
            v.begin(),
            v.end(),
            [](const int& aArg){return aArg % 2 == 0;});
std::cout<<*it;
```

- STL has many methods in algorithm that end with _*if* to easily support lambdas.
- It is very likely that lambda will be inlined.

auto

```cpp
std::string fun(){
        return std::string{"as"};
}
std::string& funRef(){
        static std::tring str{"abc"};
        return str;
}
/* ... */
auto var1 = 1;          // <-- var1
auto var2 = fun();      // <-- var2
auto var3 = funRef();   // <-- var3

int intValue = 0;
int& intRef = intValue;
auto var4 = intRef;      // <-- var4
```

auto

```
std::string fun(){
        return std::string{"as"};
}
std::string& funRef(){
        static std::tring str{"abc"};
        return str;
}
/* ... */
auto var1 = 1;          // <-- int
auto var2 = fun();      // <-- std::string
auto var3 = funRef();   // <-- std::string

int intValue = 0;
int& intRef = intValue;
auto var4 = intRef;       // <-- int
```

```cpp
int intValue = 10;
auto var5 = std::move(intValue); // <-- var5
auto& var6 = std::move(intValue); // <-- var6
auto&& var7 = std::move(intValue); // <-- var7

const int intValueC = 0;
auto var8 = intValueC;      // <-- var8
auto& var9 = intValueC; // <-- var9
const auto& var10 = intValueC; // <-- var10

auto var11 = {1, 2, 3}; // <-- var11
auto var12 = {1}; // <-- var12
auto var13{1}; // <-- var13
auto var14{1, 2};  // <-- var14
```

auto

```cpp
int intValue = 10;
auto var5 = std::move(intValue); // <-- int
auto& var6 = std::move(intValue); // <-- int&
auto&& var7 = std::move(intValue); // <-- int&&

const int intValueC = 0;
auto var8 = intValueC;      // <-- int
auto& var9 = intValueC; // <-- const int&
const auto& var10 = intValue; // <-- const int&

auto var11 = {1, 2, 3}; // <-- std::initializer_list
auto var12 = {1}; // <-- std::initializer_list, C++17: int
auto var13{1}; // <-- int
auto var14{1, 2};  // <-- Compilation Error
```

auto is **never** deduced as reference

Variadic templates

Variadic templates

```cpp
template<typename T>
void write(const T& aArg){
        std::cout<<aArg<<std::endl;
}
template<typename T1, typename ...T2>
void write(const T1& aArg, const T2& ...aArgs){
        std::cout<<aArg<<std::endl;
        write(aArgs...);
}
template<typename ...T>
void fun(const T&...aArg){
        std::cout<<"Arguments: "<<sizeof...(T)<<std::endl;
        write(aArg...);
}
fun(1, 2, 3, 4);
```

Enums

Enums

```
enum class MyEnum : char { VALUE1, VALUE2};

MyEnum var1 = MyEnum::VALUE2;
MyEnum var2 = static_cast<MyEnum>(1);

std::cout<<static_cast<int>(var1)<<std::endl;
std::cout<<static_cast<int>(var2)<<std::endl;

std::cout<<sizeof(var1);
```

Suffix

Suffix

```
constexpr long double operator"" _deg (long double deg
    ) {
    return deg * 3.141592 / 180;
}

long double angle = 90_deg;

constexpr std::chrono::minutes operator"" _min(
    unsigned long long m) {
    return std::chrono::minutes(m);
}

std::chrono::minutes var = 20_min;
```

C++14 has several predefined suffixes : min, hours, seconds, etc.

Const expressions

Const expressions

```
struct Class {
        constexpr Class(int aArg1, int aArg2)  : iVar1
    {aArg1}, iVar2{aArg2}{}
        int iVar1;
        int iVar2;
};
constexpr int fun(const int aArg){
        return 5 + aArg;
}
/* ... */
        constexpr int var1 = 2;
        constexpr int var2 = fun(4);
        constexpr Class cl{1, 2};
```

```
constexpr int fun1(int aArg){
        return aArg < 0 ? -1 : 1;
}
constexpr int ret = fun1(5);
```

C++14 :

```
constexpr int fun2(int aArg){
        if (aArg < 0)
                return -1;
        else
                return 1;
}
constexpr int ret = fun2(5);
```

Const expressions

All methods in header flie.

```
struct Class {
        constexpr Class() {}
        constexpr int fun(int aArg) const{
                return iValue + aArg;
        }
        int iValue = 0;
};
```

In source file :

```
constexpr Class obj;
obj.fun(10);
```

Initializer list

Initializer list

```
struct Class {
        Class(std::initializer_list<int> aArg) : iVar(aArg.begin(), aArg.end()){}

        std::vector<int> iVar;
};
        Class var{1, 2, 3, 4};

        std::vector<int> var2{1, 2, 3, 4, 5, 6, 7};
```

Attributes

Attributes : *final*, *override*, *default*, *delete*,

```
struct FinalClass final {
        FinalClass(const FinalClass&) = delete;
        virtual ~FinalClass() = default;
        virtual void overrideFun() override;
        virtual void finalFun() final;
};

template<typename T>
void fun(T& aArg){}

template<>
void fun(int&) = delete;
```

Foreach

Foreach

```
std::vector<int> var{1, 2, 3};
for (const auto& i : var) {
}
```

Foreach requires methods *begin* and *end*.

noexcept

```cpp
void fun() noexcept {}
```

throw() - possible that stack unwinding

noexcept - now stack unwiding - faster code

Brackets { }

Brackets { }

```
struct Class {
   Class(int aArg){}
};
        Class cl{1.23}; // It will not compile
        Class cl{static_cast<int>(1.23)}; // this is ok
```

There is no automatic conversion.

Constructor delegate

Calling constructor from constructor

```
struct Class {
        Class() : Class(0){}

        Class(int aArg){}
};
```

It is similar to Java.

nullptr

nullptr

```
void fun(char* aArg){
        std::cout<<"char*\n";
}

void fun(int aArg){
        std::cout<<"int\n";
}
        fun(0); // int
        fun(NULL); // it will not compile
        fun(nullptr); // char*
```

STL in C++11

Thread

```cpp
struct Counter {
    std::mutex mutex; // <-- mutex
    int value = 0;

    void increment(){
        std::lock_guard<std::mutex> guard(mutex); // <-- lock
        ++value;
    }
};
```

Thread - mutex II

```cpp
for(int i = 0; i < 5; ++i){
            threads.push_back(std::thread([&counter](){
                for(int i = 0; i < 100; ++i){
                    counter.increment();
                }
            }));
        }

        for(auto& thread : threads){
            thread.join(); // <-- wait
        }
```

Thread - async

```cpp
#include <thread>
int fun(){
	std::cout<<"Thread id: "<<std::this_thread::get_id()<<std::endl;
	int i = 0;
	for (; i < 100; ++i){
		std::this_thread::sleep_for(std::chrono::nanoseconds{100000000});
	}
	return i;
}
```

```
#include <future>
int main() {
        std::cout<<"Main Thread id: "<<std::this_thread::get_id()<<std::endl;
        std::vector<std::future<int>> v;
        for (int i = 0; i < 2; ++i) {
                std::future<int> ret = std::async(std::launch::async, fun);
                v.push_back(std::move(ret));
        }
        std::cout<<"wait"<<std::endl;
        for (std::future<int>& fut : v) {
                int value = fut.get();
                std::cout<<value<<std::endl;
        }
        return 0;
}
```

Thread - async II

```cpp
std::future<int> ret = std::async(std::launch::async,
    fun);
```

- std : :launch : :async - new thread
- std : :launch : :deferred - caller thread

Destructor of *std : :future<T>* waits for execution of async call.
C++14 : waits if all are true

- created from std : :async
- shared object is not ready
- the last reference of shared object

Smart pointers

DON'T USE RAW POINTERS

```
#include <memory>

std::shared_ptr<int> ptr1 = std::make_shared<int>(2);

std::shared_ptr<int> ptr2(new int[size], [](int* aArg)
    { delete[] aArg;});

std::unique_ptr<int> ptr3(new int());

std::unique_ptr<int> ptr4 = std::make_unique<int>();
    // C++14
```

DON'T USE RAW POINTERS

```
#include <memory>

struct Base {
        virtual ~Base() = default;
};
struct Derivative : public Base {};

std::shared_ptr<Base> ptrBase =
        std::make_shared<Derivative>();
std::shared_ptr<Derivative> ptrDerivative =
        std::dynamic_pointer_cast<Derivative>(ptrBase)
    ;
```

DON'T USE RAW POINTERS

```cpp
#include <memory>

struct Class {};

std::shared_ptr<Class> ptr = std::make_shared<Class>()
    ;
std::weak_ptr<Class> weakPtr = ptr;

std::shared_ptr<Class> sharedPtrFrom = weakPtr.lock();
/* using sharedPtrFrom */
```

tuple

std : :tuple

```
#include <tuple>
std::tuple<int, double> fun(){
        return std::make_tuple(1, 2.3);
}
// ...
        auto ret = fun();
        int intValue = std::get<0>(ret);
        double doubleValue = std::get<1>(ret);
```

std : :tie

```
#include <tuple>
std::tuple<int, double> fun(){
        return std::make_tuple(1, 2.3);
}
// ...
        int intValue;
        double doubleValue;
        std::tie(intValue, doubleValue) = fun();
        std::cout<<intValue<<" "<<doubleValue<<std::endl;

        // C++ 17 (now only Clang):
        auto [intValue, doubleValue] = fun();
```

array

DON'T USE RAW ARRAYS

```
#include <array>
std::array<int, 5> arr1 = {1, 2, 3};

// C++14 only one {}
std::array<int, 5> arr2{{1, 2, 3}};

std::cout<<"size: "<<arr1.size()<<std::endl;
for (auto& it : arr1){
        std::cout<<it<<std::endl;
}
```

bind

std : :bind

```
#include <functional>
void fun1(){
        std::cout<<"void fun1()\n";
}
void fun2(int aArg){
        std::cout<<"void fun2(int aArg)\n";
}
// ...
int intValue = 1;
std::function<void()> functor1a = fun1;
std::function<void()> functor1 = std::bind(fun1);
functor1();
// with one argument
std::function<void()> functor2 = std::bind(fun2,
    intValue);
functor2();
```

std : :bind

```cpp
#include <functional>
void fun3(int& aArg, double* aPtr){
        std::cout<<"void fun3(int& aArg, double* aPtr)\n";
}
// ...
std::function<void()> functor3 = std::bind(fun3, intValue, nullptr);
functor3();

using namespace std::placeholders;
std::function<void(int&)> functor4 = std::bind(fun3, _1, nullptr);
functor4(intValue);

std::function<void(double*, int&)> functor5 = std::bind(fun3, _2, _1);
functor5(nullptr, intValue);
```

std : :bind

```
#include <functional>
struct Class{
        void method(int aArg)  {
                std::cout<<"void Class::method(int aArg)\n";
        }
};
// ...
Class obj;
// class method
std::function<void( Class*, int)> functor6 = &Class::method;
functor6(&obj, intValue);
// class method with placeholder
std::function<void(int)> functor7 = std::bind(&Class::method, &obj, _1);
functor7(intValue);
```

Type traits

Type traits

```cpp
#include <type_traits>
template<typename T>
void traits(){
        cout<<is_array<T>::value<<"\n";
        cout<<is_class<T>::value<<"\n";
        cout<<is_function<T>::value<<"\n";
        cout<<is_pointer<T>::value<<"\n";
        cout<<is_polymorphic<T>::value<<"\n";
}
//...
traits<int>();
traits<std::string>();
```

Type traits - std::enable_if

```
#include <type_traits>
template<class T>
typename std::enable_if<std::is_integral<T>::value,
    bool>::type
  is_odd (T i) {
        return i % 2;
}
//...
int intValue = 0;
is_odd(intValue);
std::string strValue{"abc"};
is_odd(strValue); // Error
```

Type traits - std::enable_if

```cpp
#include <type_traits>
template<class T,
         class = typename std::enable_if<std::is_integral<T>::value>::type>
bool is_even (T i) {
        return !bool(i % 2);
}
//...
int intValue = 0;
is_even(intValue);
std::string strValue{"abc"};
is_even(strValue); // Error
```

ratio

std : :ratio

```cpp
#include <ratio>
std::ratio<1,3> one_third;
std::ratio<2,4> two_fourths;

std::cout << "one_third= " << std::ratio<1,3>::num
          << "/" << std::ratio<1,3>::den << std::endl;
std::cout << "two_fourths= " << std::ratio<2,4>::num
          << "/" << std::ratio<2,4>::den << std::endl;

std::ratio_add<std::ratio<1,3>, std::ratio<2,4>> sum;
std::cout << "sum= " <<
          std::ratio_add<std::ratio<1,3>, std::ratio<2,4>>::num
          << "/" <<
          std::ratio_add<std::ratio<1,3>, std::ratio<2,4>>::den;
```

chrono

std : :chrono

```
#include <chrono>
std::chrono::time_point<std::chrono::system_clock> start, end;
start = std::chrono::system_clock::now();
std::cout<<"write sth\n";
end = std::chrono::system_clock::now();
std::chrono::nanoseconds diff = end - start;
std::cout<<diff.count()<<"ns\n";

std::time_t end_time = std::chrono::system_clock::to_time_t(end);
std::cout << "finished computation at " << std::ctime(&end_time);
// write sth
// 25000ns
// finished computation at Sun Dec 18 17:06:17 2016
```

C++14

Binary literal & separator :

```
int binary = 0b101; // = 5
int bigNumber = 1'000'000; // ' - separator
```

Automatic return type decuction :

```
auto fun(bool aArg){
        if (aArg)
                return 1;
        else
                return 0;
}
```

C++14

Generic lambda :

```
auto lambda = [](auto x, auto y) {return x + y;};
std::cout<<lambda(1, 2.3);
```

Lambda capture expressions :

```
int var = 1;
auto lambda = [inside = std::move(var)](auto x, auto y
    ) {return x + y + inside;};
std::cout<<lambda(1, 2.3);
```