**Modern & Effective C++**

1. **R-value and L-value**

**L-value (left) –** об'єкт, який зберігається існує після обчислення виразу. Такий об'єкт має ім'я. Таким чином усі змінні - це l-value**.**

**R-value (right)** – тимчасовий об'єкт, який не зберігається після обчислення виразу в якому він використовується.

Приклад:

auto x = 1.5 + 0.2

auto result = std::sin(x);

Значення у змінних x та result - це l-values. До цих значень можна звернутися за іменем. Значення 1.5 + 0.2 і std::sin(x) - це r-values, які являються тимчасовими об’єктами. R-values не обов'язково знаходяться праворуч:

auto result(std::sin(x));

1. **Moving**

Розглянемо приклад:

store.push\_back(std::string("Some string"));

До С++ 11:

1. Буде викликано конструктор std::string з аргументом "Some string".
2. Буде викликано метод vector<string>::push\_back(const value\_type& \_Val).
3. Буде викликано конструктор копіювання string::string(const string&).

Щоб уникнути зайвого копіювання в С++11 ввели операцію переміщення. Насправді, для передавання у функцію тимчасового об'єкта рядка, немає потреби виділяти нову ділянку пам'яті та копіювати старий рядок у новий буфер: достатньо лише перемістити вказівник на початкову строку:

С++ 11:

1. Буде викликано конструктор std::string з аргументом "Some string".
2. Буде викликано метод **vector<string>::push\_back(value\_type&& \_Val)**.
3. Буде викликано **конструктор переміщення** string::(string&&).

Синтаксис **type&&** - позначає **r-value** посилання.

**Зверніть увагу**: Якщо r-value посилання має ім'я, то воно є l-value!

**Обратите внимание:** Если r-value ссылка имеет имя, то она является l-value!

1. **Universal-reference (forwarding reference)**
   1. Універсальне посилання - це посилання, яке залежно від умов може бути, як r-value посиланням, так і l-value посиланням.
   2. **Правило особливого виведення (special type deduction rules)**

Нехай є шаблонна функція

template <typename T>

void Func(T&& arg)

{

//some actions

}

Залежно від того, що передається у Func, параметр arg може бути або l-value посиланням, або r-value. Правила, що визначають це, називаються правилом особливого висновку:

*Якщо Func викликається для r-value, то arg буде r-value посиланням; у випадку ж, коли Func викликається для l-value, то arg буде l-value посиланням.*

Розглянемо наслідки з цього правила на кількох прикладах:

Func(int(0));

У цьому прикладі у функцію передається r-value, а T виводиться як **int**.

int x = 0;

Func(x);

У цьому випадку x - lvalue, тому arg - l-value-посилання, а T виводиться як **int&.**

Залежно від того, що передається l-value або r-value, має місце різне виведення типу T.

* 1. **Правила згортання посилань (reference collapsing rules)**

T& и & стає T&

T&& и & стає T&

T& и && стає T&

T&& и && стає T&&

З прикладу передачі у функцію Func змінної x (l-value) T виводиться як int&. Має місце згортання посилання:

void Func(int& &&arg) => void Func(int& arg).

Згортання посилань відбувається в чотирьох контекстах:

* виведення шаблонного типу
* виведення auto
* виведення typedef
* у decltype

**Поняття універсального посилання виникає лише в контекстах, у яких відбувається виведення типу.**

1. **Perfect forwarding**

Розглянемо фабричну функцію:

template<typename T, typename Arg>

shared\_ptr<T> Factory(Arg arg)

{

return shared\_ptr<T>(new T(arg));

}

Призначення наведеної фабричної функції - передати аргумент конструктору T. Недолік функції - аргумент передається за значенням, що призводить до зайвого копіювання

Розглянемо альтернативу:

template<typename T, typename Arg>

shared\_ptr<T> Factory(Arg& arg)

{

return shared\_ptr<T>(new T(arg));

}

Тепер стало краще, за одним винятком: функцію не можна викликати для r-value: Factory<Foo>(5) і Factory<Foo>("Some string") призведуть до помилки компіляції. Ця ситуація може бути виправлена, якщо додатково надати перевантаження для константного посилання:

template<typename T, typename Arg>

shared\_ptr<T> Factory(Arg const & arg)

{

return shared\_ptr<T>(new T(arg));

}

У такого підходу є дві проблеми:

1. необхідно надавати дві версії функції (для константного і не константного посилання);
2. аргумент у тілі функції Factory - lvalue, а це блокує move-семантику, що призводить до деградації продуктивності.

Вихід є - використання універсальних посилань і perfect forwarding:

template<typename T, typename Arg>

shared\_ptr<T> Factory(Arg**&&** arg)

{

return shared\_ptr<T>(new T(**std::forward**<Arg>(arg)));

}

**std::forward** - це шаблонна функція стандартної бібліотеки С++, яка реалізована наступним способом:

template <typename T>

T&& forward(typename std::remove\_reference<T>::type& arg)

{

return static\_cast<T&&>(arg);

}

Шаблонна функція remove\_reference прибирає декорування типу посиланням (тобто для int& і int&& виведений тип буде int).

Використовуючи правило особливого виводу і правила згортання посилань легко переконатися, що при передачі в Factory аргументу як l-value, в конструктор буде передано l-value посилання, а при передачі r-value - в конструктор буде передано r-value посилання, що дасть змогу використовувати переміщення.

Щоб отримати r-value посилання на l-value слід використовувати std::move:

template <typename T>

typename std::remove\_reference<T>::type&& move(T&& arg)

{

return static\_cast<typename std::remove\_reference<T>::type&&>(arg);

}

У наступному прикладі буде викликано переміщуюче перевантаження push\_back

std::vector<std::string> v;

std::string str = "Hello World!";

v.push\_back(std::move(str));

**Використовуйте std::forward для універсальних посилань (наприклад, у контексті шаблонної функції), а std::move для l-value посилань.**

1. **Special members**

|  |  |
| --- | --- |
| [Конструктор](http://www.cplusplus.com/doc/tutorial/classes2/#default_constructor) за замовчанням | Widget::Widget(); |
| [Деструктор](http://www.cplusplus.com/doc/tutorial/classes2/#destructor) | Widget::~Widget(); |
| [Конструктор](http://www.cplusplus.com/doc/tutorial/classes2/#copy_constructor) копіювання | Widget::Widget(const Widget&); |
| [Оператор присвоювання](http://www.cplusplus.com/doc/tutorial/classes2/#copy_assignment) | Widget& operator= (const Widget&); |
| [Переміщуючий](http://www.cplusplus.com/doc/tutorial/classes2/#move) конструктор | Widget::Widget(Widget&&); |
| [Присвоювання, що переміщує](http://www.cplusplus.com/doc/tutorial/classes2/#move) | Widget& operator=(Widget&&); |

Операції, що переміщують, генеруються (за необхідності) для класів, у яких виконуються три такі умови:

* у класі не оголошено жодних копіювальних операцій
* у класі не оголошено жодних операцій, що переміщують
* у класі не оголошено деструктор

**Рекомендація "великої трійки":** Якщо оголошено хоча б одну з трьох операцій - копіювальний конструктор, копіювальний оператор присвоювання або деструктор, то мають бути оголошені всі три операції.

**У базових класах зазвичай оголошується віртуальний деструктор**, а отже, це пригнічує переміщувальні операції. Щоб базовий клас мав переміщувальні операції їх слід оголосити явно:

­­class Base

{

public:

virtual ~Base() = default;

Base(Base &&) = default;

Base& operator=(Base&&) = default;

Base(const Base&) = default;

Base& operator = (const Base&) = default;

};