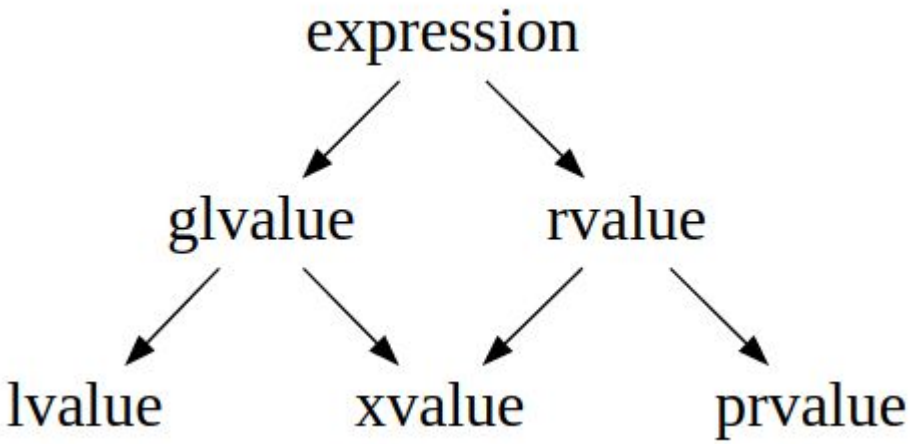
Язык C++

Move Semantics

Value categories



Value Categories

- glvalue (generalized lvalue)
  - Выражение определяющие **идентичность** объекта или функции
- prvalue (pure rvalue)
  - Выражение вычисляющие временный объект
- xvalue (expiring value)
  - Объект, значение которого может быть переиспользовано
- lvalue
  - glvalue, но не xvalue
- rvalue
  - prvalue или xvalue

Value Categories

```
int main(int, char**) {
    int i = 1; // lvalue
    ++i; // lvalue
    f(); // lvalue

    42;        // prvalue
    nullptr;   // prvalue
    i++;       // prvalue
    i + i;     // prvalue
    g();       // prvalue


    boo().x;   // xvalue
}
```

```
struct Boo {
    int x;
};
Boo boo() {
    return Boo();
}

int x = 0;
int& f() {
    return x;
}

int g() {
    return 48;
}
```

Return Value Optimization (RVO/NRVO)

```
struct Foo {
    Foo() {
        std::cout << "Foo()\n";
    }
    ~Foo() {
        std::cout << "~Foo()\n";
    }

    Foo(const Foo&) {
        std::cout << "Foo(const Foo&)\n";

    }
    Foo& operator=(const Foo&) {
        std::cout << "operator=\n" ; return *this;
    }
};
```

```
Foo foo1() {
    return Foo();
}

Foo foo2() {
    Foo result;

    return result;
}

Foo foo3(int i) {
    Foo odd;
    Foo even;
    return i % 2 == 0 ? odd : even;
}
```

CArray

```
class CArray {
public:
   CArray() {...}
   explicit CArray(size_t size) {...}
   ~CArray() {...}
   CArray(const CArray& array) {....}
   CArray& operator=(const CArray& array) {. . .  }
protected:
   void swap(CArray& array) {};
private:
   int8_t* data_ = nullptr;
   size_t size_ = 0;
};
```

Проблема избыточного копирования

```
int main() {
    CArray arr1{5};
    CArray arr2{};

    arr2 = arr1;
    arr2 = createArray();
    return 0;
}
```

Rvalue reference

- && - rvalue reference (& - lvalue reference)
- Позволяет передавать в функцию rvalue
- Продлевает жизнь временным объектам
- move constructor
- move assignment operator
- reference collapsing

RValue reference

```
void foo(Foo& ) {
   std::cout << "void foo(Foo& ) \n";
}

void foo(const Foo& ) {
   std::cout << "void foo(const Foo& ) \n";
}

void foo(Foo&& ) {
   std::cout << "void foo(Foo&& ) \n";
}
```

```
int main(int, char**) {
   Foo f;
   const Foo cf;
   Foo&&  rvf = Foo{};

   foo(f);
   foo(cf);
   foo(Foo{});
   foo(rvf);  // !!!
}
```

Move constructor & move assignment

```
CArray(CArray&& array) noexcept
    : size_(std::exchange( array.size_, 0))
    , data_(std::exchange( array.data_, nullptr))
{
}

CArray& operator=(CArray&& array) noexcept {
    delete[] data_;
    size_ = std::exchange( array.size_, 0);
    data_ = std::exchange( array.data_, nullptr);

    return* this;
}
```

Move constructor & move assignment

- Передают все значения полей в текущий объект
- Оставляют копируемый объект в инвариантном но неопределенном состоянии
- Очищают ресурсы текущего объекта
- default\delete
- Правило 5

Проблема избыточного копирования

```
int main() {
   CArray arr1{5};
   CArray arr2{};

   arr2 = arr1;           // lvalue
   arr2 = createArray(); // prvalue
   arr2 = std::move(arr1); // xvalue
   return 0;
}
```

Copy\Move assignment operator

```
CArray& operator=(CArray array) {
    swap(array);
    return *this;
}
```

std::move

- Кастит в rvalue

```
template <class _Tp>
typename remove_reference<_Tp>::type&&
move(_Tp&& __t) _NOEXCEPT {
 typedef typename remove_reference<_Tp>::type _Up;
 return static_cast<_Up&&>(__t);
}
```

std::swap

```
template<class T>
void std::swap(T& x, T& y) {
    T tmp = move(x);
    x = move(y);
    y = move(tmp);
}
```

Forwarding reference

```
template<typename T>
void function(T&& value) {
}

int main(int, char**) {
   Foo&& foo = Foo{};
   auto&& value = foo;
}
```

- Универсальная ссылка
- lvalue если инициализируется lvalue
- rvalue если инициализируется rvalue

Reference collapsing

```
int main(int, char**) {
   Foo f;

   function(f);
   function(Foo{});
}
```

- Foo& & -> Foo&
- Foo&& & -> Foo&
- Foo& && -> Foo&
- Foo&& && -> Foo&&

Perfect forwarding

```
template<typename T, typename Arg>
std::unique_ptr<T> my_make_unique(Arg arg) {
   return std::unique_ptr<T>(new T(arg));
}

template<typename T, typename Arg>
std::unique_ptr<T> my_make_unique(Arg arg) {
   return std::unique_ptr<T>(new T(arg));
}


int main(int, char**) {
   Foo f;
   my_make_unique<Foo>(f);
   my_make_unique<Foo>(Foo{});
}
```

Perfect forwarding

```
template<typename T, typename Arg>
std::unique_ptr<T> my_make_unique(Arg&& arg) {
    return std::unique_ptr<T>(new T(arg)); // !! lvalue
}

int main () {
    my_make_unique<Foo>(Foo{});
}
```

std::forward

```
template< typename T >
T&& forward(std::remove_reference_t<T>& t ) noexcept {
   return static_cast<T&&>( t );
}
```

- lvalue скастит к lvalue
- rvalue скастит к rvalue
- в отличии от std::move который делает это безусловно

Perfect forwarding

```
template<typename T, typename Arg>
std::unique_ptr<T> my_make_unique(Arg&& arg) {
   return std::unique_ptr<T>(new T(std::forward<Arg>(arg)));
}

int main () {
   my_make_unique<Foo>(Foo{});
}
```

Perfect forwarding

```
template<typename T, typename... Arg>
std::unique_ptr<T> my_make_unique(Arg&&... arg) {
   return std::unique_ptr<T>(new T(std::forward<Arg>(arg)...));
}
```