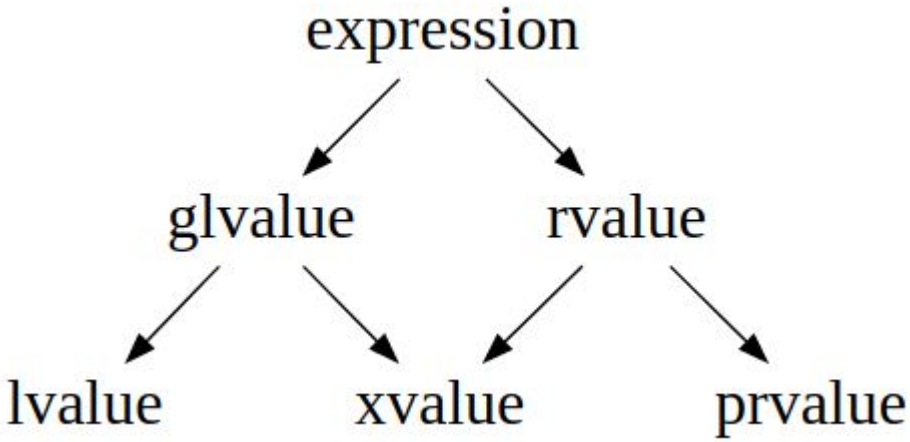
Язык C++

Value categories & Move Semantics

Value categories



Каждое выражение
имеет тип и категорию

Lvalue. Язык Си

A 5. Объекты и Lvalues

Объект — это некоторая именованная область памяти; *lvalue* — это выражение, обозначающее объект. Очевидным примером lvalue является идентификатор с соответствующим типом и классом памяти. Существуют операции, порождающие lvalue. Например, если `E` — выражение типа указатель, то `*E` есть выражение для lvalue, обозначающего объект, на который указывает `E`. Термин "lvalue" произошел от записи присваивания `E1 = E2`, в которой левый (left — левый (англ.), отсюда буква l, value — значение) операнд `E1` должен быть выражением lvalue. Описывая каждый оператор, мы сообщаем, ожидает ли он lvalue в качестве операндов и выдает ли lvalue в качестве результата.

Язык Си. lvalue & rvalue

```c
int main() {
    int i;
    i = 2024; // i - lvalue, 2024 - rvalue
    2024 = i; // Compile-time error

    int arr[10];
    arr[1] = i;    // arr[1] - lvalue, i - lvalue

    return 0;
}
```

Не справедливо для современного C++

- Выражение относящееся к объекту, **который занимает место в памяти**
- rvalue - все что не lvalue. **Не обязан иметь выделенное место**
- При присваивании левый операнд всегда lvalue, правый lvalue или rvalue

lvalue & rvalue

```
'a';          // rvalue
128;          // rvalue
3.14f;        // rvalue

int i = 1;    // lvalue
int j = 2;    // lvalue
i + j;        // rvalue
i + j = 2;    // Compile-time error
&i;           // rvalue

int* pi = &i;
*pi;          // lvalue

const int k = 1; // lvalue
k = 3;           // Compile-time error
```

- rvalue
  - нельзя поменять
  - нельзя получить адрес
- lvalue
  - можно получить адрес
  - менять можно но не всегда

References and lvalue & rvalue

```
int func(int i) {
    return i;
}

int main() {

    int x = 2;
    func(x);
    func(2);

    return 0;
}
```

```
int func(int& i) {
    return i;
}

int main() {

    int x = 2;
    func(x);
    func(2);  // Error

    return 0;
}
```

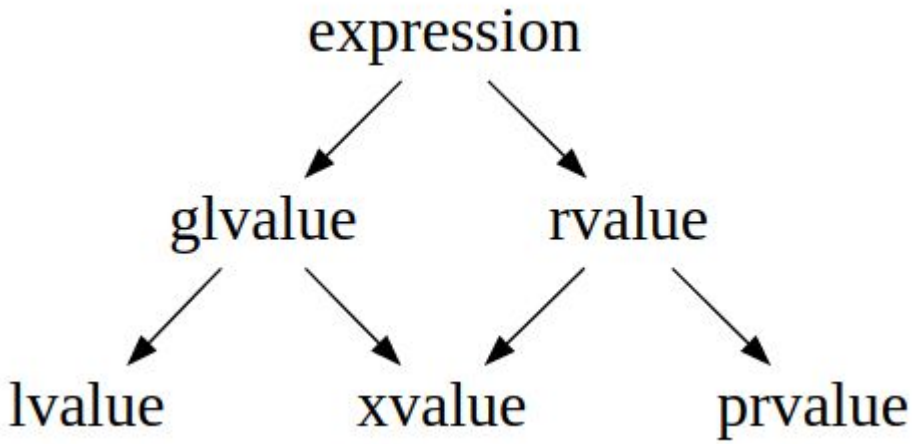
```
int func(const int& i) {
    return i;
}

int main() {

    int x = 2;
    func(x);
    func(2);

    return 0;
}
```

Value categories



Value Categories

- **glvalue (generalized lvalue)** - Выражение определяющие **идентичность** объекта или функции
- **rvalue** - prvalue или xvalue

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

Rvalue

```
struct Foo {
    Foo() = default;
    Foo(int i) : value(i){}
    int value = 0;
};

void func(const Foo& v) {
}

int main() {
    func(Foo{}); // creating temprorary object
    func(2);     // creating temprorary object
    Foo{}.value; // creating temprorary object
    return 0;
}
```

Rvalue

- **prvalue** - pure rvalue
- **xrvalue** - expiring rvalue

Rvalue reference

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

Rvalue reference

```
int&& func(int&& i) {
    return i;
}

int main() {
    int&& i = 1;
    const int&& j = 2;

    std::cout << func(1);

    int x = 2;
    int&& rx = x; // error
    const int&& crx = x; // error
    return 0;
}
```

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);  // !!!
}
```

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;
}
```

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
- Правило 0

Special Members

compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Проблема избыточного копирования

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

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

Copy-And-Swap Idiom

```
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)...));
}
```

Lvalue & rvalue reference

```
void boo(Boo&) {
}

void boo(const Boo&) {
}

void boo(Boo&&) {
}

void boo(const Boo&&) {
}

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

Copy elision

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

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

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

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

Foo nvro() {
    Foo result;

    return result;
}

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