



Объектно-ориентированное программирование лекция №2

2020

Передача информации о переменных в функции и объекты

Из C мы помним, что передавать переменные можно:

- «по значению» – путем копирования;
- «с помощью указателя» – тогда копируется указатель, а переменная «остается на месте»;

&ССЫЛКИ

// СТРУКТУРА ОБЪЯВЛЕНИЯ ССЫЛОК

/*ТИП*/ &/*ИМЯ ССЫЛКИ*/ = /*ИМЯ ПЕРЕМЕННОЙ*/;

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    int value = 15;
    int &reference = value; // объявление и инициализация ссылки значением переменной value
    cout << "value      = " << value      << endl;
    cout << "reference = " << reference << endl;
    reference+=15; // изменяем значение переменной value посредством изменения значения в ссылке
    cout << "value      = " << value      << endl; // смотрим, что получилось, как будет видно даль
    ше значение поменялось как в ссылке,
    cout << "reference = " << reference << endl; // так и в ссылочной переменной
    system("pause");
    return 0;
}
```

& Ссылки

- ✓ Ссылка — это синоним имени переменной, т. е. другое имя для использования переменной.
- ✓ Отличие ссылки от указательной переменной в том, что ссылка не является объектом. Для названия ссылки может не отводиться место в памяти. Для указательной переменной место в памяти выделяется всегда.
- ✓ Ссылка не может ссылаться на несуществующий объект, указатель может.
- ✓ Ссылку нельзя переназначить, а указательную переменную можно.

Lvalue & Rvalue переменные

С каждой **обычной** переменной связаны две вещи – **адрес** и **значение**.

```
int I; // создать переменную по адресу,  
например 0x10000
```

```
I = 17; // изменить значение по адресу  
0x10000 на 17
```

А что будет если у меня есть только значение? Могу ли я сделать так: `20=10; ?`



с любым
выражением
связаны либо адрес
и значение, либо
только значение

Для того, чтобы отличать выражения, обозначающие объекты, от выражений, обозначающих только значения, ввели понятия **lvalue** и **rvalue**. Изначально слово lvalue использовалось для обозначения выражений, которые могли стоять слева от знака присваивания (*left-value*); им противопоставлялись выражения, которые могли находиться только справа от знака присваивания (*right-value*).

Пример

`i` — lvalue

`++i` — lvalue

`*&i` — lvalue

`a[5]` — lvalue

`a[i]` — lvalue

`10` — rvalue

`i + 1` — rvalue

`i++` — rvalue

LVALUE_REFERENC E.CPP

```
void foo(std::string &val){  
    val = "I am foo";  
}
```

```
void boo(std::string val){  
    val = "I am boo";  
}
```

```
int main() {  
    std::string value = "Hello world!";  
  
    std::cout << "value = " << value << std::endl;  
    foo(value);  
    std::cout << "value = " << value << std::endl;  
    boo(value);  
    std::cout << "value = " << value << std::endl;  
    return 0;  
}
```

Пример Rvalue Reference

```
struct A {  
    A(){  
        std::cout << "Created" << std::endl;  
    }  
  
    A(int a) : value(a){  
        std::cout << "Created with value" <<  
        std::endl;  
    }  
  
    A(const A &a) :A() {  
        value= a.value;std::cout << "Вот оно что!"  
        << std::endl;  
    }int value;  
};  
  
void over(A &a) { std::cout << "LValue:" <<  
a.value << std::endl;}  
void over(A &&a) { std::cout << "RValue:" <<  
a.value << std::endl;}  
void cross(A a) { std::cout << "Copy:" <<  
a.value << std::endl;}
```



Неизменяемость

// const

Константный указатель и указатель на константу

```
int main()
{
    int a = 200;
    const int *p1 = &a;
    int const *p2 = &a;
    const int *p3 = &a;
    int const * const p4 = &a;
    const int * const p5 = &a;
}
```

<https://habr.com/ru/post/59558/>

Константные функции

```
struct A {  
    int x;  
    void f(int a) const {  
        x = a; // <-- не работает  
    }  
};
```

const.cpp

```
struct MyClass {  
    /*mutable*/ int value{};  
    const int foo(const int a) const {  
        // ++value;  
        return value+a;  
    }  
};
```

static

КЛЮЧЕВОЕ СЛОВО,
КОТОРОЕ ЗАДАЕТ
ГЛОБАЛЬНОЕ ВРЕМЯ
ЖИЗНИ ОБЪЕКТА.

static.cpp

```
int counter(){  
    static int i=0;  
    return ++i;  
}
```


Перегрузка операций

OPERATOR

Перегрузка операций

Почему операция `std::cin >> file_text` имеет смысл?

В C++ существуют механизмы, которые позволяют сопоставлять арифметический и другие операции, такие как побитовый сдвиг обычным функциям!

Это позволяет лучше описывать типы. Мы можем описать не просто класс, но и операции с объектами этого класса.

Прежде чем начать

Предупреждение

- Это механизм, при неумелом использовании которого можно полностью запутать код.
- Непонятный код – причина сложных ошибок!
- Перегруженные операции помогают определить «свойства» созданного вами класса, но не алгоритма работы с классами!

Перегрузка операций

Можно описать функции, для описания следующих операций:

+ - * / % ^ & | ~ !

= < > += -= *= /= %= ^= &=

|= << >> >>= <<= == != <= >= &&

|| ++ -- ->* , -> [] () new delete

Нельзя изменить приоритеты этих операций, равно как и синтаксические правила для выражений. Так, нельзя определить унарную операцию %, также как и бинарную операцию !.

Синтаксис

type **operator** operator-symbol (parameter-list)

Ключевое слово `operator` позволяет перегружать операции. Например:

- Перегрузка унарных операторов:
 - **ret-type operator** op (arg)
 - где **ret-type** и op соответствуют описанию для функций-членов операторов, а arg — аргумент типа класса, с которым необходимо выполнить операцию.
- Перегрузка бинарных операторов
 - **ret-type operator** op(arg1, arg2)
 - где *ret-type* и op — элементы, описанные для функций операторов членов, а arg1 и arg2 — аргументы. Хотя бы один из аргументов **должен принадлежать типу класса**.

OPERATOR_PLUS.CPP

Пример сложения

Префиксные и постфиксные операторы

++ и --

Операторы инкремента и декремента относятся к особой категории, поскольку имеется два варианта каждого из них:

- преинкрементный и постинкрементный операторы;
- предекрементный и постдекрементный операторы.

При написании функций перегруженных операторов полезно реализовать отдельные версии для префиксной и постфиксной форм этих операторов. Для различения двух вариантов используется следующее правило: **префиксная** форма оператора объявляется точно так же, как и любой другой унарный оператор; в **постфиксной** форме принимается дополнительный аргумент типа **int**.

Пример:

```
friend Point& operator++( Point& ) // Prefix increment
friend Point& operator++( Point&, int ) // Postfix increment
friend Point& operator--( Point& ) // Prefix decrement
friend Point& operator--( Point&, int ) // Postfix decrement
```


OPERATOR_IN
CREMENT.CPP

Пример унарного оператора

OPERATOR_COPY.CPP

Пример оператора присваивания

OPERATOR_FUNCTOR.CPP

Пример функтора

START_WITH.CPP

Пример бинарной перегрузки

Еще один пример

RVALUE_REFERENCE.CPP



Спасибо!

НА СЕГОДНЯ ВСЕ