Объектно-ориентированное программирование лекция №2

2019

&ссылки

```
// СТРУКТУРА ОБЪЯВЛЕНИЯ ССЫЛОК
/*ТИП*/ &/*ИМЯ ССЫЛКИ*/ = /*ИМЯ ПЕРЕМЕННОЙ*/;
```

```cpp
#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; ?

Итого: 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*).

С каждой функцией компилятор также связывает две вещи: ее адрес и ее тело («значение»).

Пример

```
char a [10];
i      —   lvalue
++i    —   lvalue
*&i    —   lvalue
a[5]   —   lvalue
a[i]   —   lvalue

однако:
10     —   rvalue
i + 1  —   rvalue
i++    —   rvalue
```

Пример Lvalue Reference

LVALUE_REFERENCE.CPP

Пример Rvalue Reference

REFERENCE.CPP

Еще один пример

RVALUE_REFERENCE.CPP

Константы

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;
}
```

Пример

CONST.CPP

constexpr

Ключевое слово constexpr, добавленное в C++11, перед функцией означает, что если значения параметров возможно посчитать на этапе компиляции, то возвращаемое значение также должно посчитаться на этапе компиляции.

```
constexpr int sum (int a, int b){
 return a + b;
}

void func() {
  constexpr int c = sum (5, 12);
}
```

Пример

CONSTEXPR.CPP

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

- 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**.

Пример

OPERATOR_PLUS.CPP

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
```

Пример

OPERATOR_INCREMENT.CPP

Пример

OPERATOR_COPY.CPP

Пример

OPERATOR_FUNCTOR.CPP

Литералы

Литерал — это некоторое выражение, создающее объект. В языке C++ существуют литералы для разных встроенных типов (2.14 Literals):

123 // int

1.2 // double

1.2F // float

'a' // char

1ULL // unsigned long long

0xD0 // unsigned int в шестнадцатеричном формате

"as" // string

Пользовательские литералы

Должны начинаться с подчеркивания:

```
OutputType operator "" _suffix(unsigned long long);
```

Конструктор типа должен так же иметь спецификатор **constexpr**

Могут иметь следующие параметры:

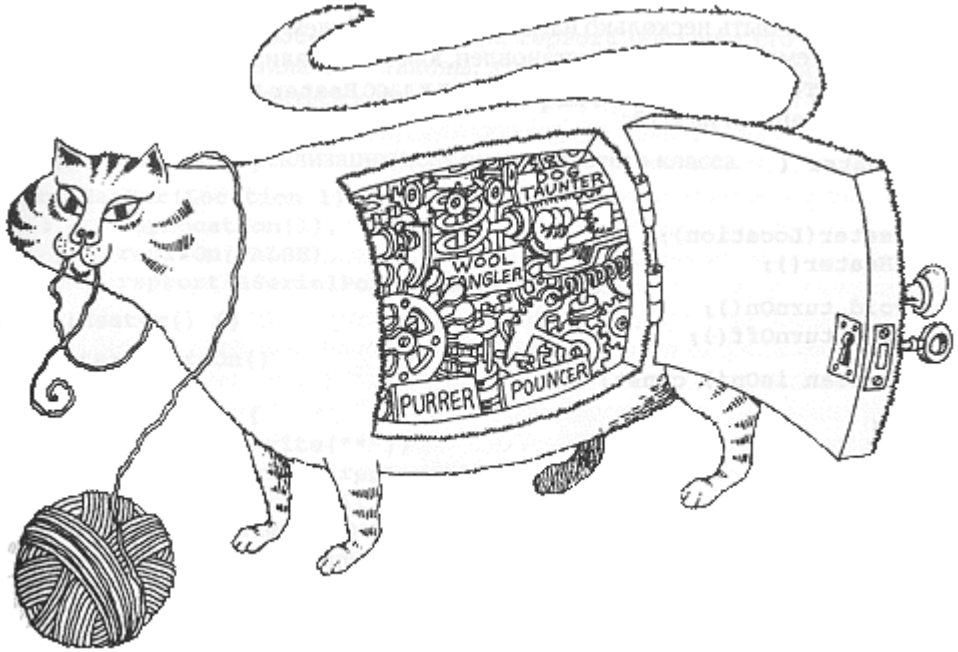
```
const char*
unsigned long long int  long double

char
wchar_t  char16_t  char32_t
const char*, std::size_t  const wchar_t*,
std::size_t
const char16_t*, std::size_t  const char32_t*,
std::size_t
```

Пример

LITERAL.CPP

Инкапсуляция



Инкапсуляция - это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Инкапсуляция, пример: контроль доступа в C++

Член класса может быть **частным** (**private**), **защищенным** (**protected**) или **общим** (**public**):

1. Частный член класса X могут использовать только функции-члены и друзья класса X.
2. Защищенный член класса X могут использовать только функции-члены и друзья класса X, а так же функции-члены и друзья всех производных от X классов (рассмотрим далее).
3. Общий член класса можно использовать в любой функции.

Контроль доступа применяется единообразно ко всем именам. На контроль доступа не влияет, какую именно сущность обозначает имя.

Друзья класса объявляются с помощью ключевого слова **friend**. Объявление указывается в описании того класса, к частным свойствам и методам которого нужно подучать доступ.

Пример

INCAPSULATION.CPP

Эквивалентность типов

Два структурных типа считаются различными даже тогда, когда они имеют одни и те же члены.

Например, ниже определены различные типы:

```
class s1 { int a; };
class s2 { int a; };
```

В результате имеем:

```
s1 x;
s2 y = x; // ошибка: несоответствие типов
```

Кроме того, структурные типы отличаются от основных типов, поэтому получим:

```
s1 x;
int i = x; // ошибка: несоответствие типов
```

Спасибо!

НА СЕГОДНЯ ВСЕ