



Объектно-ориентированное программирование лекция №3

2020

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);  
}
```

CONSTEXPR.CPP

```
#include <cstdlib>
#include <iostream>

constexpr int One(){
    return 1;
}

constexpr int all(int a,int b) {
    return (a| b) + One();
};

int main() {
    int value;
    std::cin >> value;
    switch(value){
    case 0: std::cout << "Zero" << std::endl;break;
    case 1: std::cout << "One" << std::endl;break;
    case all(0,1): std::cout << "Two" << std::endl;break;
    case all(1,2) : std::cout << "All" << std::endl;break;
    }
    return 0;
}
```

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
```

LITERAL.CPP

```
class Massa
{
public:
    std::string number;
    Massa(const char *str) : number(str) { number += "kg";
    };
    };
    Massa operator"" _kg(const char *str)
    {
        return Massa(str);
    }

    std::string operator"" _zzz(const char *str, size_t)
    {
        return std::string(str);
    }
}
```

Объектно-ориентированное программирование

Объектно-ориентированное программирование - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.



Как придумать класс?

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.



Абстракция сущности

Объект представляет собой полезную модель некой сущности в предметной области



Абстракция поведения

Объект состоит из обобщенного множества операций

Абстракция виртуальной машины

Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня



Произвольная абстракция

Объект включает в себя набор операций, не имеющих друг с другом ничего общего

Наследование

В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общем суперклассе. По этой причине говорят о наследовании, как об иерархии *обобщение-специализация*. Суперклассы при этом отражают наиболее общие, а подклассы - более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы и даже скрыты.

Классификация животного мира



INHERITANCE. CPP

```
class Animal
{
protected:
std::string name;
public:
Animal(const char *val) : name(val){};
const char *get_name() { return name.c_str(); }
};

class Moose : public Animal
{
public:
Moose() : Animal("Moose"){};
void run()
{
std::cout << name << " running" << std::endl;
}
};
```

Модификаторы функций



Ключевое слово **virtual** опционально и поэтому немного затрудняло чтение кода, заставляя вечно возвращаться в вершину иерархии наследования, чтобы посмотреть объявлен ли виртуальным тот или иной метод.



Типовые ошибки: Изменение сигнатуры метода в наследнике.

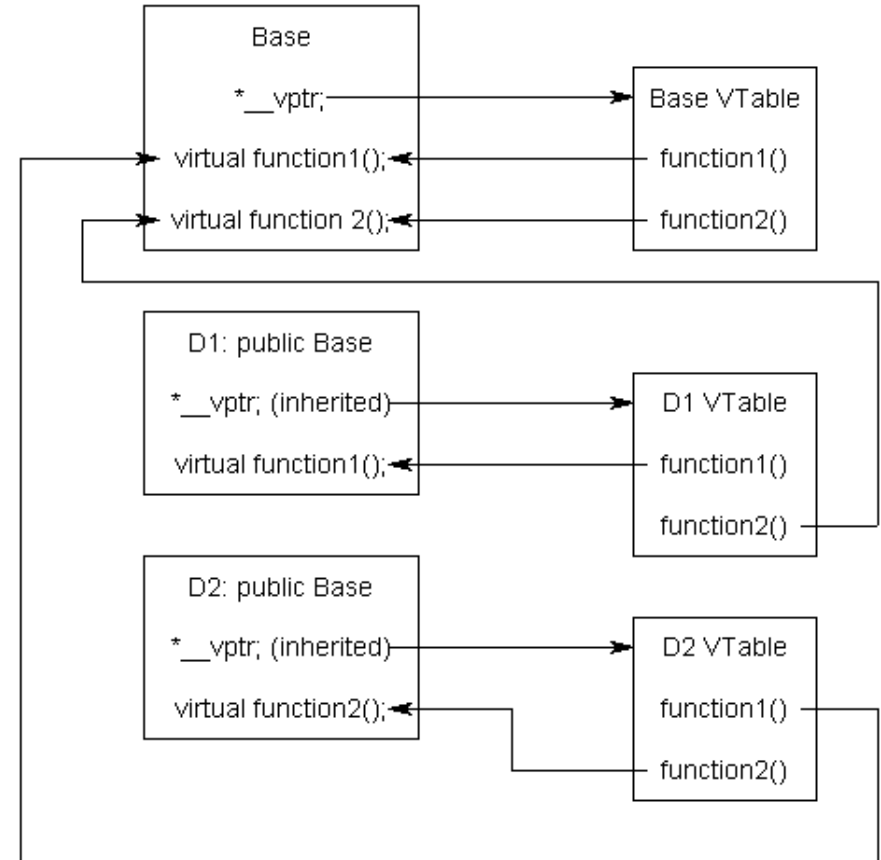
Модификаторы функций

Модификатор **override** позволяет указать компилятору, что мы хотим переопределить виртуальный метод. Если мы ошиблись в описании сигнатуры метода – то компилятор выдаст нам ошибку. Этот модификатор влияет только на проверки в момент компиляции.



Виртуальные функции

```
class Base{  
    virtual void function1();  
    virtual void function2();  
};  
  
class D1{  
    void function1();  
};  
  
class D2{  
    void function2();  
}
```



VIRTUAL.CPP

```
class Animal{
public:
    virtual size_t get_size() =0;
    virtual ~Animal(){
        std::cout << " Animal destructor" << std::endl;
    }
};

class Moose : public Animal{
public:
    size_t get_size() override{
        return 1000;
    }
    ~Moose(){
        std::cout << " Moose destructor" << std::endl;
    }
};
```

О виртуальных функциях

1. Виртуальную функцию можно использовать, даже если нет производных классов от ее класса.
2. В производном же классе не обязательно переопределять виртуальную функцию, если она там не нужна.
3. При построении производного класса надо определять только те функции, которые в нем действительно нужны.

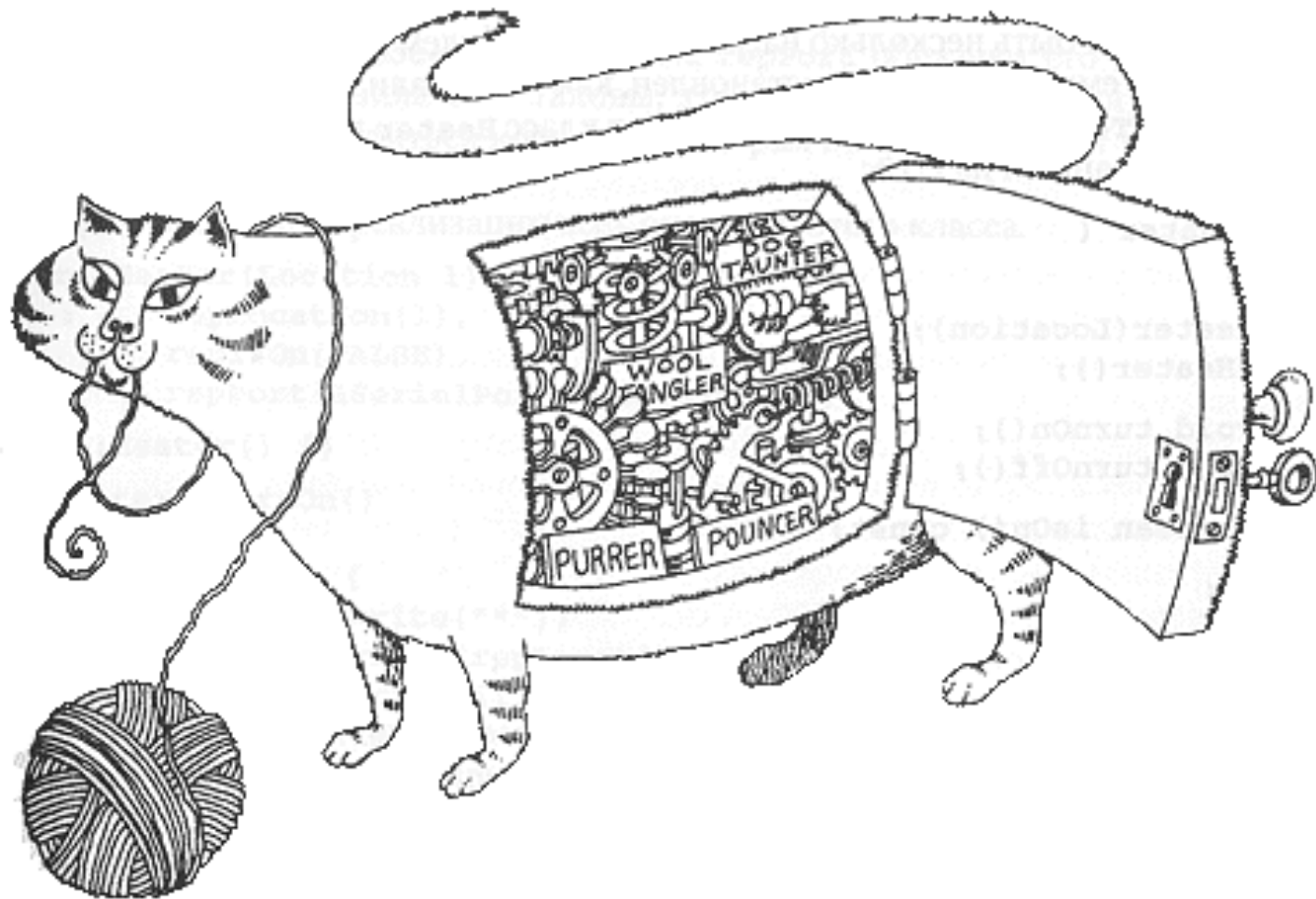
Абстрактные классы

```
class Item
{
public:
    virtual const char * GetMyName( ) = 0;
};
```

Объект абстрактного
класса нельзя создать!

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

```
class A {  
    friend class B;  
private:  
    int value;  
public:  
    A(int v) : value(v) {};  
};
```


Конструктор

Если у класса есть конструктор, он вызывается всякий раз при создании объекта этого класса. Если у класса есть деструктор, он вызывается всякий раз, когда уничтожается объект этого класса.

Объект может создаваться как:

1. автоматический, который создается каждый раз, когда его описание встречается при выполнении программы, и уничтожается по выходе из блока, в котором он описан;
2. статический, который создается один раз при запуске программы и уничтожается при ее завершении;
3. объект в свободной памяти, который создается операцией `new` и уничтожается операцией `delete`;
4. объект-член, который создается в процессе создания другого класса или при создании массива, элементом которого он является.

Конструкторы при наследовании

```
class employee {  
    // ...  
public:  
    // ...  
    employee(char* n, int d);  
};  
  
class manager : public employee {  
    // ...  
public:  
    // ...  
    manager(char* n, int i, int d);  
};
```

```
manager::manager(char* n, int l, int d)  
: employee(n,d), // вызов родителя  
  level(l), // аналогично level=1  
  group(0)  
{  
}
```

Виртуальные деструкторы

C++ вызывает деструктор для текущего типа и для всех его родителей.

Однако, если мы работаем с указателями то можем попасть в неприятную ситуацию, когда вызываем оператор `delete` у указателя, предварительно приведя его к типу родителя. В этом случае, вообще говоря, у наследников деструктор вызван не будет (например, если у родителя нет деструктора).

Однако если объявить деструктор базового класса как `virtual` то будут вызваны деструкторы всех классов.

Всегда объявляй деструктор как `virtual`!
... но это не бесплатно



Пример

VIRTUAL_DESTRUCTOR.CPP

Последовательность вызова конструкторов и деструкторов

Конструкторы вызываются начиная от родителя к наследнику.

Деструкторы вызываются начиная от наследника к родителю.

```
class A {}  
class B : A {}  
class C: B{}
```

Конструкторы:

A, B, C

Деструкторы:

~C, ~B, ~A

Inheritance / наследование

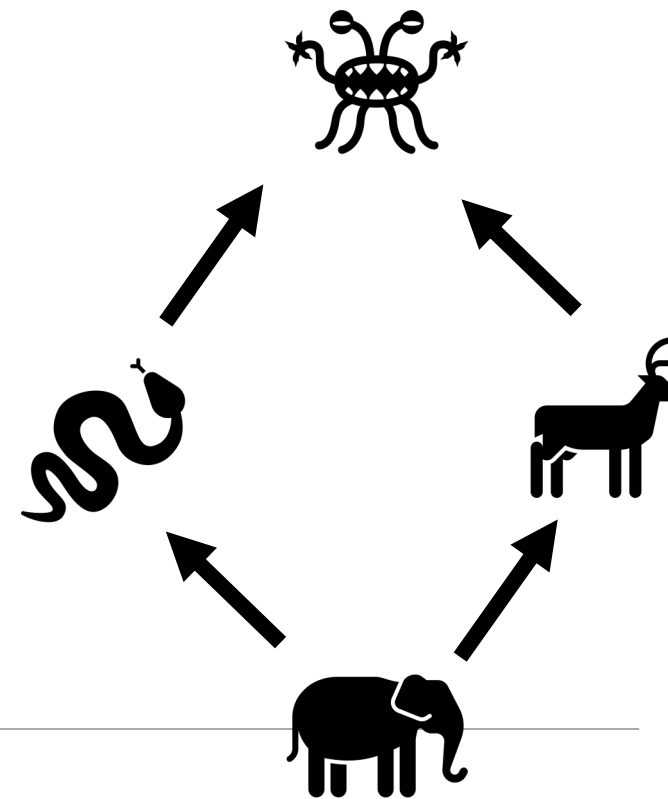
```
class temporary { /* ... */ };  
class secretary : public  
employee { /* ... */ };  
  
class tsec  
  
: public temporary, public  
secretary { /* ... */ };  
  
class consultant  
  
: public temporary, public  
manager { /* ... */ };
```

В C++ в отличие от большинства других объектно-ориентированных языков есть множественное наследование!

Очень плохо, если у двух родителей класса есть общий родитель!

Пример

MULTI_INHERITANCE.CPP



Модификатор final

Модификатор **final**, указывающий что производный класс не должен переопределять виртуальный метод.

Работает только с модификатором **virtual**. Т.е. Создавать «копию» функции в классе-наследнике с помощью этой техники запретить нельзя.

Применяется, в случае если нужно запретить дальнейшее переопределение метода в дальнейших наследниках наследника (очевидно, что в родительском классе такой модификатор ставить бессмысленно).

Спецификатор delete и default

Иногда очень важно сделать так, что бы у объекта был только один экземпляр. Т.е., что бы его нельзя было скопировать.

Сейчас стандартная идиома «запрещения копирования» может быть явно выражена следующим образом:

```
class X {  
    // ...  
  
    X& operator=(const X&) = delete;    // Запрет копирования  
    X(const X&) = delete; // запрет копирование в момент конструирования  
};
```

Такая конструкция запрещает компилятору «создавать» конструкторы и оператор копирования «по умолчанию».

Ключевое слово **=default**, наоборот, указывает что мы хотим что бы компилятор использовал операцию «по умолчанию». Вообще говоря, она является избыточной.

Пример

DELETE.CPP

Ссылка на родителя

Example25_ReferenceToParent

```
class Parent {  
public:  
    Parent(void);  
    ~Parent(void);  
    void Foo(void);  
};  
  
class Child : public Parent {  
public:  
    Child(void);  
    ~Child(void);  
    void Foo(void);  
};
```

```
void Parent::Foo(void)  
{  
    std::cout << "Parent\n";  
}  
  
void Child::Foo(void)  
{  
    Parent::Foo();  
    std::cout << "Child\n";  
}
```

Пустой указатель `nullptr`

Раньше, для обнуления указателей использовался макрос `NULL`, являющийся нулем — целым типом, что, естественно, вызывало проблемы (например, при перегрузке функций).

Ключевое слово **`nullptr`** имеет свой собственный тип **`std::nullptr_t`**, что избавляет нас от бывших проблем.

Существуют неявные преобразования `nullptr` к нулевому указателю любого типа и к `bool`.



Спасибо!

НА СЕГОДНЯ ВСЕ