Junior C++ developer

Лекция 5

- Пространство имен
- Наследование

Пространство имен

Используются для предотвращения конфликта имен

Для доступа к классам из ***namespace*** используется оператор **::**

Например:
std::string

```
#include "person.h"
#include <iostream>


Person::Person(std::string f_name, std::string l_name,
int id) :
    first_name(f_name),
    last_name(l_name),
    unic_id(id)
{
    std::cout << "Constructing " <<
    first_name << " " << last_name << std::endl;
}

Person::Person() :
    unic_id(0)
{
    std::cout << "Constructing " <<
    first_name << " " << last_name << std::endl;
}

Person::~Person()
{
    std::cout << "Destructing " <<
    first_name << " " << last_name << std::endl;
}
```

```
#include "person.h"
#include <iostream>
using std::cout;
using std::endl;

Person::Person(std::string f_name, std::string l_name, int id)
:
    first_name(f_name),
    last_name(l_name),
    unic_id(id)
{
    cout << "Constructing " <<
    first_name << " " << last_name << endl;
}

Person::Person() :
    unic_id(0)
{
    cout << "Constructing " <<
    first_name << " " << last_name << endl;
}

Person::~Person()
{
    cout << "Destructing " <<
    first_name << " " << last_name << endl;
}
```

```cpp
#include "person.h"
#include <iostream>

using namespace std;

Person::Person(string f_name, string l_name, int id) :
    first_name(f_name),
    last_name(l_name),
    unic_id(id)
{
    cout << "Constructing " <<
    first_name << " " << last_name << endl;
}

Person::Person() :
    unic_id(0)
{
    cout << "Constructing " <<
    first_name << " " << last_name << endl;
}

Person::~Person()
{
    cout << "Destructing " <<
    first_name << " " << last_name << endl;
}
```

Подключает всё из пространства имен ***std***.
Так можно делать в файле ***.cpp***
Так делать **нельзя** в файле ***.h***

```cpp
#include "person.h"
#include <iostream>

using std::cout;
using std::endl;
using std::string;

Person::Person(string f_name, string l_name, int id) :
    first_name(f_name),
    last_name(l_name),
    unic_id(id)
{
    cout << "Constructing " <<
    first_name << " " << last_name << endl;
}

Person::Person() :
    unic_id(0)
{
    cout << "Constructing " <<
    first_name << " " << last_name << endl;
}

Person::~Person()
{
    cout << "Destructing " <<
    first_name << " " << last_name << endl;
}
```

Хороший пример использования ***using***

```cpp
#include <string>

class Person
{
public:
    Person(std::string f_name, std::string l_name, int id);
    Person();
    ~Person();

    std::string getName();

private:

    std::string first_name;
    std::string last_name;
    int unic_id;
};
```

“Наследование - это процесс добавления полей и методов” Голуб А.И.

Принцип подстановки Барбары Лисков

Наследующий класс должен дополнять, а не замещать поведение базового класса

Если у нас есть класс A (не виртуальный, а вполне реально используемый в коде) и отнаследованный от него класс B, то если мы заменим все использования класса A на B, ничего не должно измениться в работе программы. Ведь класс B всего лишь расширяет функционал класса A. Если эта проверка работает, то поздравляю: ваша программа соответствует принципу подстановки Лисков! Если нет, стоит уволить ведущего программиста задуматься: «а правильно ли спроектированы классы?». habr

```
class Base
{
public:
    Base()
    {
        std::cout << "Create Base class" << std::endl;
    }
    ~Base()
    {
        std::cout << "Destroy Base class" << std::endl;
    }
    void base_function()
    {
        std::cout << "call base function " << std::endl;
    }

private:
    int base_value;
};
```

```
class Derived : public Base
{
public:

    Derived()
    {
        std::cout << "Create Derived class" << std::endl;
    }

    ~Derived()
    {
        std::cout << "Destroy Derived class" << std::endl;
    }

    void derived_function()
    {
        base_function();
        std::cout << "into derived function" << std::endl;
    }

};
```

Типы наследования

- В C++ существует 3 типа наследования
  - `public`
  - `protected`
  - `private`

```
class Duck : public Bird {
   ...
   ...
   ...
};
```

Если здесь не указан тип, то по умолчанию считается private

Типы наследования

- При объявлении члена класса открытым (***public***) к нему можно получить доступ из любой другой части программы.

- Если член класса объявляется закрытым (***private***), к нему могут получаться доступ только члены того же класса. К закрытым членам базового класса не имеют доступа даже производные классы.

- Если член класса объявляется защищенным (***protected***), к нему могут получать доступ только члены тоже же или производных классов. Спецификатор ***protected*** позволяет наследовать члены, но оставляет их закрытыми в рамках иерархии классов.

Типы наследования

- Если базовый класс наследуется как ***public***:
  - ***public*** -> ***public***
  - ***protected*** -> ***protected***

- Если базовый класс наследуется как ***protected***:
  - ***public*** -> ***protected***
  - ***protected*** -> ***protected***

- Если базовый класс наследуется как ***private***:
  - ***public*** -> ***private***
  - ***protected -> private***

Время жизни объектов

- Конструктор вызывается в момент создания объекта, один раз.

```
Triangle()
  : a(1), b(1), c(1) {
  cout << "Triangle ctor" << endl;
}
```

- Деструктор вызывается в момент окончания жизни объекта

```
~Triangle() {
  cout << "Triangle dtor" << endl;
}
```

- Для локальных переменных время жизни заканчивается когда они покидают область видимости.

Практика

1. Реализовать класс SuperPerson, который
   a. Публично наследуется от класса Person
   b. Имеет одно приватное поле *std::string super_name;*

1. Реализовать класс SuperPerson, который:
   a. Публично наследуется от класса Person
   b. Имеет одно приватное поле *std::string super_name;*

```
class SuperPerson : public Person
{
public:
   SuperPerson(std::string first_name,
         std::string last_name,
         int unic_id,
         std::string super_name);
   ~SuperPerson();

private:
   std::string super_name;
};
```

```
#include "super_person.h"
#include <iostream>

using std::cout;
using std::endl;
using std::string;

SuperPerson::SuperPerson(string first_name,
      string last_name,
      int unic_id,
      string super_name)
      :
      Person(first_name, last_name, unic_id),
      super_name(super_name)
{
   cout << "Constructing SuperPerson" << endl;
}

SuperPerson::~SuperPerson()
{
   cout << "Destructing SuperPerson";
}
```

Что будет выведено на экран?

```
#include <iostream>
#include "person.h"
#include "super_person.h"

using std::cout;
using std::endl;
using std::string;

int main() {
    Person person1("Fiona", "Smith", 7);
    {
        SuperPerson s_person("Alice", "Alice", 8, "@alice");
        cout << s_person.getName() << endl;
    }
    cout << person1.getName() << endl;
    return 0;
}
```

```
Constructing Fiona Smith
Constructing Alice Alice
Constructing SuperPerson
Alice Alice
Destructing SuperPerson
Destructing Alice Alice
Fiona Smith
Destructing Fiona Smith
```

Конструкторы и деструкторы в классах-наследниках

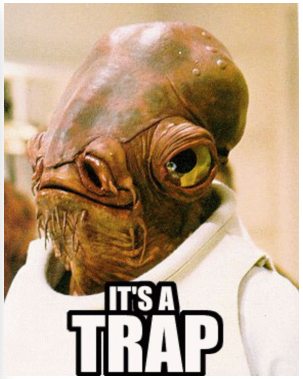
- Деструкторы вызываются в момент уничтожения объекта
- Конструкторы вызываются в порядке происхождения классов.
- Деструкторы в обратном порядке.

Порядок поиска полей и методов

- Сначала компилятор ищет среди полей и методов класса-наследника
- Если не находит, то ищет в базовом классе
- **Останавливается если не находит совпадений**

- Что случится в каждом из случаев?
  - Какой член класса будет найдет?
  - Скомпилируется?
  - Произойдет то чего ожидали?

```
class Base {
public:
  int x;
  void foo(int a);
  int foo(string b);
  void bar(Duck *c);
};
```



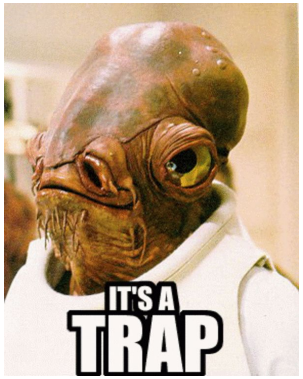

```
int main() {
  Derived der;
  Duck q;
1  int y = der.x;
2  der.foo("test");
3  der.bar(&q);
}
```

```
class Derived : public Base
{
public:
  void x(int a);
  int foo(int b);
  void bar(bool c);
};
```

Ответы:

Все поля и методы будут найдены в классе Derived

1. Не скомпилируется.
2. Не скомпилируется.
3. Скомпилируется! Указатель преобразуется к bool.




```
class Base {
public:
  int x;
  void foo(int a);
  int foo(string b);
  void bar(Duck *c);
};
```

```
int main() {
  Derived der;
  Duck q;

1  int y = der.x;
2  der.foo("test");
3  der.bar(&q);
}
```

```
class Derived : public Base
{
public:
  void x(int a);
  int foo(int b);
  void bar(bool c);
};
```