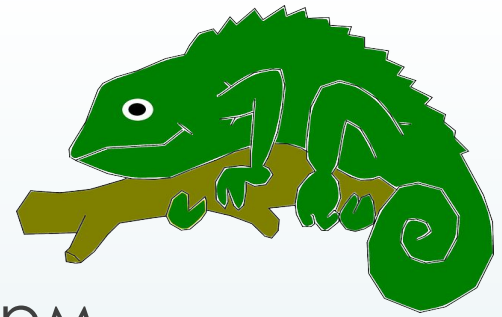


С++

Лекция 3

Полиморфизм

Полиморфизм



- Полиморфизм - много форм.
 - Способность одного “кусочка” кода вести себя по-разному.
- Формы полиморфизма.
 - Ad-hoc полиморфизм (**Перегрузка функций**)
 - Параметрический полиморфизм (**Шаблоны**)
 - **Полиморфизм подтипов**

Чаще всего когда говорят “полиморфизм” имеют в виду полиморфизм подтипов

Перегрузка функций

- **Ad-hoc Полиморфизм** перегрузка функций с одинаковым именем в одной области видимости.

```
int main() {  
    Base b;  
    b.foo(42);  
    b.foo("test");  
}
```

какая из
функций foo
вызывается
определяется
типом
параметров

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

А эта функция foo
сама по себе, так
как живет в
отдельной области
видимости

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

Перегрузка функций

4

- Несколько функций в одной области видимости с одинаковым именем, но разными *сигнатурами*.
 - Сигнатура включает в себя имя функции и входные параметры.

Что видим мы

```
class Person {  
public:  
    void greet() {...}  
    void greet(int x) {...}  
    void greet(string x) {...}  
    void greet(int x,  
                string x) {...}  
    bool greet() {...}  
};
```

Что видит компилятор

```
class Person {  
public:  
    void greet() {...}  
    void greet_int(int x) {...}  
    void greet_string(string x)  
    {...}  
    void greet_int_string(int x,  
                          string x)  
    {...}  
    bool greet() {...}  
};
```

Error, duplicate definition.

Полиморфизм подтипов

- Полиморфизм подтипов позволяет использовать базовый тип данных для хранения наследуемых типов
- Попробуем...

```
int main() {  
    Chicken c("Myrtle");  
    Duck d("Scrooge");  
    Bird b = c;  
}
```

Что-то пошло
не так?

Chicken 12 байт.
Bird 8 байт.



Полиморфизм подтипов

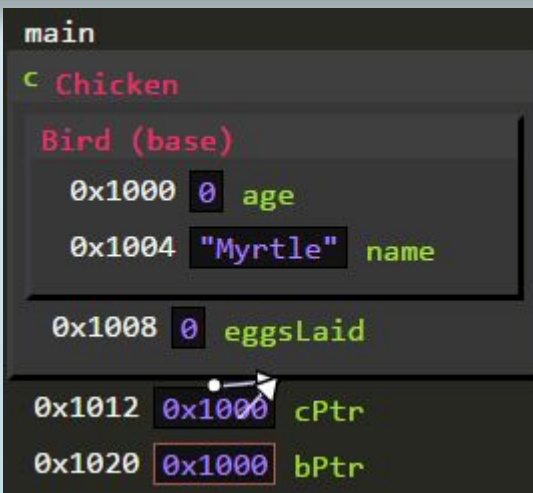
- Проблема: Нельзя использовать Bird для хранения Chicken, так как Chicken занимает большую область памяти.
- Решение: Использовать указатели или ссылки для не прямой работы с объектами.
 - Bird* и Chicken* равны по размеру.

NO

```
int main() {  
    Chicken c("Myrtle");  
    Bird b = c;  
}
```

YES

```
int main() {  
    Chicken c("Myrtle");  
    Chicken *cPtr = &c;  
    Bird *bPtr = cPtr;  
}
```



Upcast vs. Downcast

- Безопасно конвертировать один тип указателей¹ в другой? Компилятор руководствуется следующими правилами:
 - Приведение вверх по иерархии (Upcasts) безопасно(скомпилируется)
 - Приведение вниз по иерархии (Downcasts) *не безопасно* (не скомпилируется)

```
int main() {  
    Chicken c("Myrtle");  
    Chicken *cPtr = &c;  
    Bird *bPtr = cPtr;  
}
```

Норм.
Приведение вверх

```
int main() {  
    Chicken c("Myrtle");  
    Bird *bPtr = &c;  
    Chicken *cPtr = bPtr;  
}
```

Не норм.
приведение вниз

¹ Аналогично для ссылок

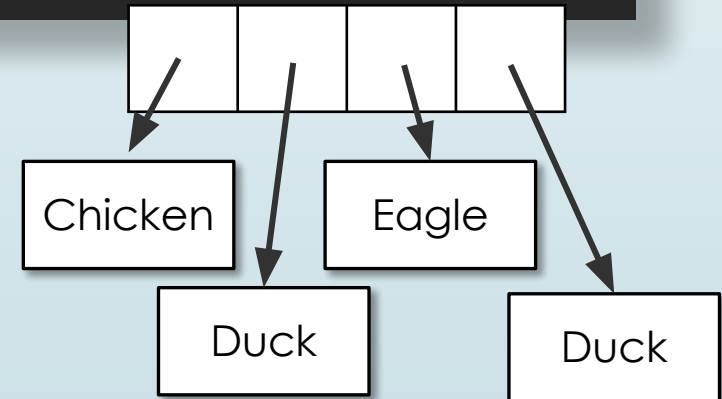
ИСПОЛЬЗОВАНИЕ

- Например можно создать массив базового класса содержащий указатели на любые наследуемые типы .

```
void allTalk(Bird * birds[], int length) {  
    for(int i = 0; i < length; ++i) {  
        birds[i]->talk();  
    }  
}
```

tweet
tweet
tweet
tweet

- Проблема:
Мы можем создать массив разных птиц, но так как они все типа Bird* будут вызываться функции базового класса, а не наследуемых.



Static vs. Dynamic Type

- **Static type** указателя/ссылки тип известный на этапе компиляции.
- **Dynamic type** указателя/ссылки тип объекта на который указывает во **время исполнения**.

```
1 Chicken c("Myrtle");  
2 Duck d("Scrooge");  
3 Bird *ptr;  
4 if (random() < 0.5) {  
5     ptr = &c;  
6 }  
7 else {  
8     ptr = &d;  
9 }
```

Какой **static** type
ptr?
Bird*

Какой **static** ptr?
Bird*

Какой **dynamic** тип
ptr?
undefined

Какой **dynamic** ptr?
Chicken*

Раннее связывание

```
int main() {  
    Chicken c("Myrtle");  
    Duck d("Scrooge");  
    Bird *ptr;  
    if (random() < 0.5) {  
        ptr = &c;  
    }  
    else {  
        ptr = &d;  
    }  
    ptr->talk();  
}
```

static type
Bird.

Что произойдет при
вызове talk()?

tweet

- Компилятор определяет во время компиляции какую из функций вызвать. Это называется раннее связывание (**static binding**).
 - Основывается на **static type** объекта вызывающего функцию (**receiver**).

Explicit Downcast

- `dynamic_cast` безопасное приведение типов по иерархии наследования.
 - В случае неправильного приведения типов для ссылок вызывается исключительная ситуация `std::bad_cast`, а для указателей будет возвращен `0`.

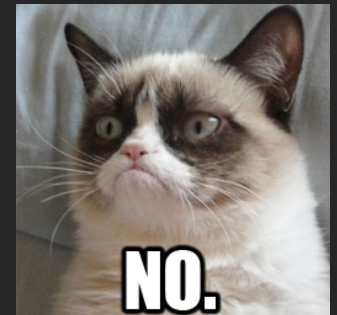
```
int main() {  
    Chicken c("Myrtle");  
    Bird *bPtr = &c;  
    Chicken *cPtr = dynamic_cast<Chicken>  
    if (cPtr != nullptr) {  
        // something chicken-specific  
    }  
}
```

Check for
null

- Если в коде часто используется `dynamic_cast` это говорит о том что программа плохо спроектирована.

```
void allTalk(Bird * birds[], int length) {  
    for(int i = 0; i < length; ++i) {  
        Chicken *cPtr = dynamic_cast<Chicken*>(cPtr);  
        if(cPtr) {  
            cPtr->talk();  
        }  
        Duck *dPtr = dynamic_cast<Duck*>(dPtr);  
        if(dPtr) {  
            dPtr->talk();  
        }  
        Eagle *ePtr = dynamic_cast<Duck*>(ePtr);  
        if(ePtr) {  
            cPtr->talk();  
        }  
    }  
}
```

Как тебе
такой код?



Для каждого наследника
Bird, придется добавлять
новый if.

Позднее связывание

```
int main() {  
    Chicken c("Myrtle");  
    Duck d("Scrooge");  
    Bird *ptr;  
    if (random() < 0.5)  
        ptr = &c;  
    }  
    else {  
        ptr = &d;  
    }  
    ptr->talk();  
}
```

Динамический тип объекта Duck*.

Допустим random() вернул 0.7.

What is the desired output of this call to talk()? Which talk() function should be used?

кря

- Что если мы хотим определить тип во время исполнения? (Динамический тип, реальный тип.)
- Для этого используется позднее связывание.
 - Основываясь **dynamic type** объекта.

Non-Virtual vs. Virtual Functions

- В C++ можно выбрать какой тип связывания использовать.
- **Раннее связывание используется по умолчанию (non-virtual).**
- Ключевое слово `virtual` для использования позднего связывания.

Non-Virtual vs. Virtual Functions

```
class Bird {  
    ...  
    void talk() const {  
        cout << "tweet" << endl;  
    }  
    ...  
};
```

Non-virtual по
умолчанию

```
class Duck : public Bird {  
    ...  
    void talk() const {  
        cout << "quack" << endl;  
    }  
    ...  
};
```

Non-virtual по
умолчанию

```
int main() {  
    Chicken c("Myrtle");  
    Duck d("Scrooge");  
    Bird *ptr;  
    if (random() < 0.5) {  
        ptr = &c;  
    }  
    else {  
        ptr = &d;  
    }  
    ptr->talk();  
}
```

Всегда будет
выводиться
"tweet".

Non-Virtual vs. Virtual Functions

```
class Bird {  
    ...  
    virtual void talk() const {  
        cout << "tweet" << endl;  
    }  
    ...  
};
```

Обязательно
чтобы функция
была объявлена
virtual только в
базовом классе

```
class Duck : public Bird {  
    ...  
    virtual void talk() const {  
        cout << "quack" << endl;  
    }  
    ...  
};
```

Но намного лучше если в
наследуемом она тоже будет virtual!

```
int main() {  
    Chicken c("Myrtle");  
    Duck d("Scrooge");  
    Bird *ptr;  
    if (random() < 0.5) {  
        ptr = &c;  
    }  
    else {  
        ptr = &d;  
    }  
    ptr->talk();  
}
```

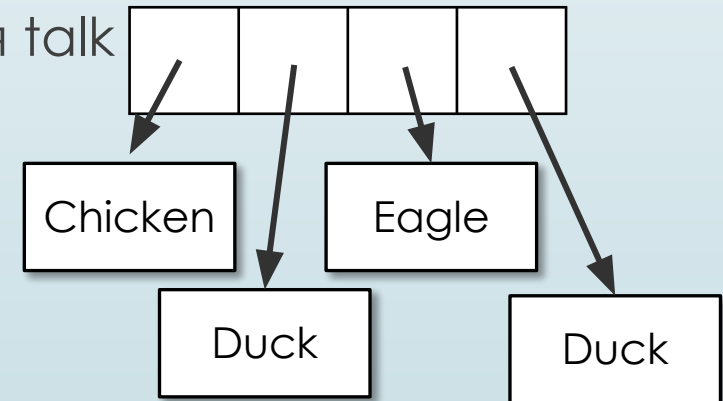
Теперь
поведение
зависит от типа
объекта во
время
исполнения.

Использование полиморфизма

- Один массив для разных “птиц”.

```
void allTalk(Bird * birds[], int length) {  
    for(int i = 0; i < length; ++i) {  
        birds[i]->talk();  
    }  
}
```

- За счет того что функция talk объявлена как virtual, каждый класс может вызывать свой вариант функции.



bawwk
quack
eeeagle
quack



18

Exercise: Виртуальные функции

```
class Grandma {
public:
    int f1() { return 1; }
    virtual int f2() { return 2; }
};

class Mom : public Grandma {
public:
    int f1() { return 3; }
    virtual int f2() { return 4; }
};

class Child : public Mom {
public:
    int f1() { return 5; }
    virtual int f2() { return 6; }
};
```

```
int main() {
    Grandma g;
    Mom m;
    Child c;
    Grandma *gPtr = &c;
    Mom *mPtr = &m;

    A cout << g.f2() << endl;
    B cout << m.f1() << endl;
    C cout << gPtr->f1() << endl;
    D cout << gPtr->f2() << endl;
    E cout << mPtr->f2() << endl;
      mPtr = &c;
    F cout << mPtr->f1() << endl;
    G cout << mPtr->f2() << endl;
}
```

Solution: Виртуальные функции

```
class Grandma {
public:
    int f1() { return 1; }
    virtual int f2() { return 2; }
};

class Mom : public Grandma {
public:
    int f1() { return 3; }
    virtual int f2() { return 4; }
};

class Child : public Mom {
public:
    int f1() { return 5; }
    virtual int f2() { return 6; }
};
```

```
int main() {
    Grandma g;
    Mom m;
    Child c;
    Grandma *gPtr = &c;
    Mom *mPtr = &m;
```

```
A  cout << g.f2() << endl;      2
B  cout << m.f1() << endl;      3
C  cout << gPtr->f1() << endl;   1
D  cout << gPtr->f2() << endl;   6
E  cout << mPtr->f2() << endl;   4
    mPtr = &c;
F  cout << mPtr->f1() << endl;   3
G  cout << mPtr->f2() << endl;   6
}
```

Переопределение виртуальных функций

- **Виртуальные функции переопределяются** в наследуемом классе, сигнатура должна совпадать
- Это не то же самое что и **перегрузка**.
 - Перегрузка функций работает с функциями в одной области видимости.
 - Переопределение работает с функциями в одной иерархии наследования.

Переопределение vs. Перегрузка

- Переопределение
 - Позволяет наследникам переопределить поведение одного из наследуемых методов.
- Перегрузка
 - Позволяет существовать нескольким функциям с одинаковым именем, но с различными параметрами.



Неудобный класс Bird

- Действительно ли нужен класс описывающий “просто птицу”?
- В реальном мире птицы представлены намного разнообразнее.
- Функция `talk()` неудобная, зачем нужен вариант по умолчанию если все птицы должны говорить по своему:

```
class Bird {  
private:  
    int age;  
    string name;  
  
public:  
    Bird(string name_in)  
        : age(0), name(name_in) {  
        cout << "Bird ctor" << endl;  
    }  
    ...  
    virtual void talk() const {  
        cout << "tweet" << endl;  
    }  
  
    virtual int getWingspan() const {  
        return -1; // ???  
    }  
};
```

Чисто виртуальные функции

- Можно сделать функции **pure virtual**.
- Это значит что в базовом классе будет только объявление, но не реализация.
- Класс Bird превращается в абстрактный.
- Нельзя создать экземпляр базового класса, но это не нужно!
- Наследники должны **переопределить** виртуальные функции базового класса.

1 В противном случае класс наследник тоже будет являться абстрактным классом.

```
class Bird {  
private:  
    int age;  
    string name;  
  
public:  
    Bird(string name_in)  
        : age(0), name(name_in) {  
        cout << "Bird ctor" << endl;  
    }  
  
    ...  
  
    virtual void talk() const = 0;  
  
    virtual int getWingspan()  
        const = 0;  
};
```

Зачем нужны чисто виртуальные функции?

- **Виртуальные функции** позволяют наследнику переопределить родительские функции, если нужно.
- **Чисто виртуальные функции** являются только объявлением и заставляют классы наследники написать реализацию функций.
- Например: У всех птиц есть параметр `wingspan`, но не существует значения по умолчанию, в каждом классе наследнике оно свое.

```
class Bird {  
public:  
...  
    virtual int getWingspan() const {  
        return -1; // ???  
    }  
};
```

Не внушает
доверия

```
class Bird {  
public:  
    virtual int getWingspane()  
        const = 0;  
};
```

Лучш

```
int main() {  
    Bird p;  
}
```

Ошибка, Bird
является
абстрактным.

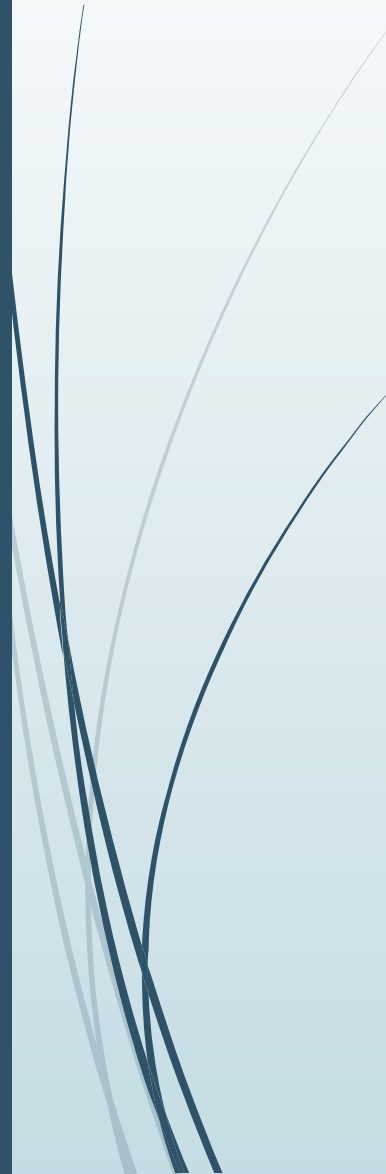
Интерфейс

- **Интерфейс** - класс в котором все функции чисто виртуальные.

```
class Shape {  
public:  
    virtual double area() const = 0;  
    virtual double perimeter() const = 0;  
    virtual void scale(double s) = 0;  
};
```

- Классы наследуемые от класса *Shape* (например *Triangle*, *Rectangle*) называются реализацией интерфейса *Shape*.

Interfaces





27

Exercise: Interfaces

Реализовать
классы Circle
Square

```
const double PI = 3.14159265;

class Shape {
public:
    virtual double area()
        const = 0;

    virtual double perimeter()
        const = 0;

    virtual void scale(double s) = 0;
};
```

```
int main() {
    Circle c(1.5);
    Square s(2.4);
    Shape *ptr = &c;

    ptr->scale(2);
    // result: 18.8496 28.2743
    cout << ptr->perimeter() << " "
          << ptr->area() << endl;

    ptr = &s;
    ptr->scale(0.5);
    // result: 4.8 1.44
    cout << ptr->perimeter() << " "
          << ptr->area() << endl;
}
```

```

class Square : public Shape {
private:
    double side;
public:
    Square(double s_in) : side(s_in) {
    }
    virtual double area() const {
        return side * side;
    }
    virtual double perimeter() const {
        return 4 * side;
    }
    virtual void scale(double s) {
        side *= s;
    }
};

```

```

int main() {
    Circle c(1.5);
    Square s(2.4);
    Shape *ptr = &c;

    ptr->scale(2);
    // result: 18.8496 28.2743
    cout << ptr->perimeter() << " "
         << ptr->area() << endl;

    ptr = &s;
    ptr->scale(0.5);
    // result: 4.8 1.44
    cout << ptr->perimeter() << " "
         << ptr->area() << endl;
}

```