Объектно-ориентированное программирование в C++

Сокрытие элементов

Ковариантность возврата для переопределений виртуальных функций

Преобразование к производному типу: cast-операции

Динамическая идентификация типов

Множественное наследование

Сокрытие

```
class Animal {
public:
   bool angry = false;
   string isAngry() const;
   ..
}
```

```
class Dog : public Animal {
public:
   bool angry = false;
};
```

```
class Bullterrier : public Dog {
public:
   bool angry = true;
};
```

```
cout << boolalpha;

Bullterrier bullie("Bullie");
// Обращаемся к полю Bullterrier::angry
//bullie.angry = false;
cout << "Is bullie angry? " << bullie.angry;

// Обращаемся к полю angry (наследство от Dog)
cout << "Is Dog angry? " << bullie.Dog::angry;


bullie.Dog::angry = true;

// Обращаемся к методу класса Animal
cout << bullie.isAngry();
```

Переопределение или сокрытие

- Переопределение требует *полного совпадения сигнатуры** метода в производного классе по отношению к методу базового класса; в противном случае происходит «сокрытие».

```
cout << student.toString(true) << endl;
cout << student.toString() << endl;
```

```
struct Person
{
   virtual string toString() const;
};
struct Student : Person
{
   virtual string toString(bool bFullInfo) const;
};
```

- Если в базовом классе несколько перегруженных функций, которые необходимо переопределить, то в производном классе нужно реализовать переопределение каждой версии

Ковариантность типа возврата

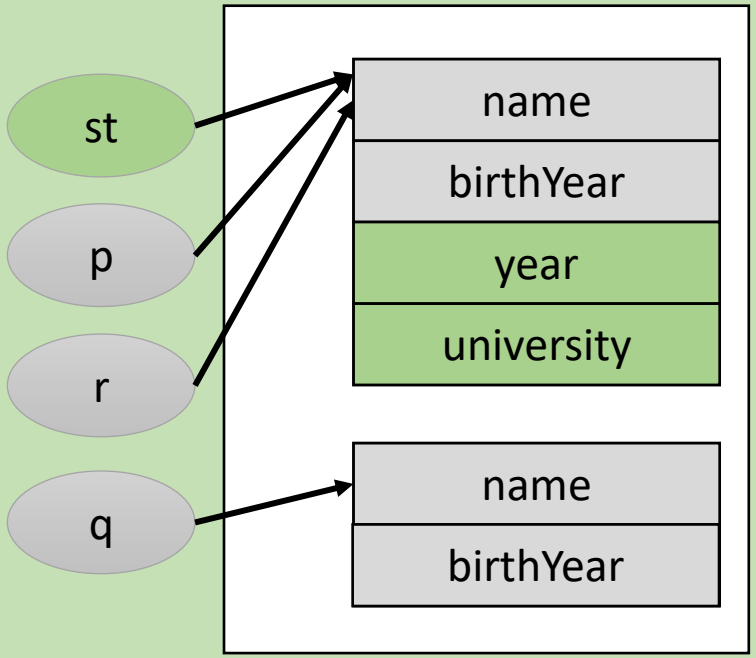
- Если виртуальная функция возвращает ссылку или указатель на какой-либо класс (н-р, **House**), то переопределения могут возвращать ссылки или указатели на этот же класс или на производный от него класс (**Dormitory: public House**)
- Это называется *ковариантным типом возврата.*
- Одноименный метод в производном классе (liveAt) с ковариантным типов возврата рассматривается как *переопределение*, а не как сокрытие;

```
struct Person
{
    virtual House& liveAt() {};
};
struct ForeignStudent : Student
{
    Dormitory& liveAt() {};
};
```

```
ForeignStudent juan { "Juan", 1999, 2, "Stanford", "Colombia" };
// ForeignStudent -> Person
Person& someone = juan;
// Dormitory -> House
House& house = someone.liveAt();
```

Приведение типов при наследовании

- Переход от производного типа к базовому выполняется неявно (*только для *открытого наследования*):




```
Student* st = new Student("John", 2000, 1, "Stanford");
// полиморфизм «адресов»:
Person* p =  st;
Person& r = *st;
// копирование объекта («обрезка»):
Person q = *st;
```

- Переход от ссылки или указателя на объект базового типа к производному типу (*понижающее преобразование*) требует явного приведения :

```
// В стиле Си:
Student* s = (Student*) p;
Student& ref_s = (Student&)(*p);
```

Приведение типов в C++: семейство cast-ов

- Для «безопасного» приведения типов в C++ вводятся операции: **static_cast, dynamic_cast, const_cast, reinterpret_cast**
- Все операции требуют указания целевого типа (во что преобразуем) в угловых скобках

```
Student* s = static_cast<Student*>(p);
```

- У каждой операции есть своя специфика и ограничения
- Использование cast-операций позволяет обнаружить некоторые ошибки на стадии компиляции (до запуска программы)
- Тем не менее успешная компиляция cast-операций не гарантирует «безаварийное» выполнение программы; некоторые ошибки могут возникнуть при выполнении программы

Приведения типов в C++: const_cast

- Операция **const_cast** позволяет «снять» ограничение константности с переменной:

```
int* x, *y;
x = new int;
const int * cx;
// переход int* -> const int*
cx = x;
y = (int*)cx; // синтаксис Си
y = const_cast<int*>(cx);
*y = -1;
// проверка типов на стадии компиляции
const short* sh;
y = const_cast<int*>(sh);  // ошибка компиляции! sh не const int*
```

Приведение типов в C++: static_cast

- Операция **static_cast** позволяет выполнить проверку возможности преобразования на стадии компиляции
- В случае классов (структур) преобразование с static_cast возможно, если типы связаны родством (*открытое наследование*):

```
Person* p = new Student("John", 2000, 1, "Stanford");
Student* s = static_cast<Student*>(p);
Rectangle* rec = static_cast<Rectangle*>(p);
```

- При выполнении программы уже не выполняется проверки, действительно ли преобразование возможно:

```
Person* p = new Person("John", 2000);
Student* s = static_cast<Student*>(p);
```

- Использовать static_cast можно, если по ссылке/указателю *действительно* содержится объект указанного типа (или производного от него)

Приведение типов в C++: dynamic_cast

- Проверка возможности преобразования осуществляется *во время выполнения программы*
- Как и в случае static_cast возможно преобразование между классами, связанными открытым наследованием

```
Person* p = new Student("John", 2000, 1, "Stanford");
Student* s = dynamic_cast<Student*>(p);
```

- Во время компиляции программы проверка корректности преобразования в плане родственности типов не выполняется
- В случае невозможности преобразования возвращается нулевой указатель

```
Rectangle* rec = dynamic_cast<Rectangle*>(p);
if(!rec)
{
    cout << "p is not rectangle" << endl;
    return -1;
}
```

Приведение типов в C++: dynamic_cast

- Проверка типов выполняется на основе указателя на таблицу виртуальных методов (vmtp). Таблица (и её адрес) уникальна для каждого типа.
- Поэтому *динамическое преобразование* с dynamic_cast возможно для классов с виртуальными методами (для *полиморфных классов*)
- Если в базовом классе устранить виртуальность методов, то преобразование станет невозможным (в отличие от **static_cast**):

```
Person* p = new Student("John", 2000, 1, "Stanford");
Student* s = dynamic_cast<Student*>(p);
```

- Операция поддерживается и для ссылок; в случае невозможности преобразования генерируется исключение bad_cast.

Система RTTI

- **dynamic_cast** является частью механизма RTTI (**run time type identification**) – динамической идентификации типов
- Другие средства RTTI: typeid – для идентификации типа, type_info – для получения информации о типах
- RTTI работает только с иерархией классов, содержащих виртуальные методы
- Операция typeid используется для проверки совпадения типов; в качестве аргумента – либо название класса, либо конкретный объект.

```
Person* p = new Student("John", 2000, 1, "Stanford");
if(typeid(*p) == typeid(Person))
        cout << "Type is Person" << endl;
if(typeid(*p) == typeid(Student))
         cout << "Type is Student" << endl;
cout << "name of type: " << typeid(*p).name() << endl;
```

Приведение типов в C++: reinterpret_cast

- Самая «низкоуровневая» операция приведения
- Обычно используется для «интерпретации» адресов (преобразования указателей разных типов, не связанных «родством»)
- Переход от void* к указателю на фактический объект:

```
void* v = new Student("John", 2000, 1, "Stanford");
Student* zero = reinterpret_cast<Student*>(v);
```

- Переход между разными типами, но с совместимой внутренней структурой:

```
Point* point1 = new Point { 1.5, 2.4};
Point2D* point2 = reinterpret_cast<Point2D*>(point1);
cout << "x = " << point2->x  << ", y = " << point2->y << endl;
```

- Интерпретация числа как адреса на область динамической памяти

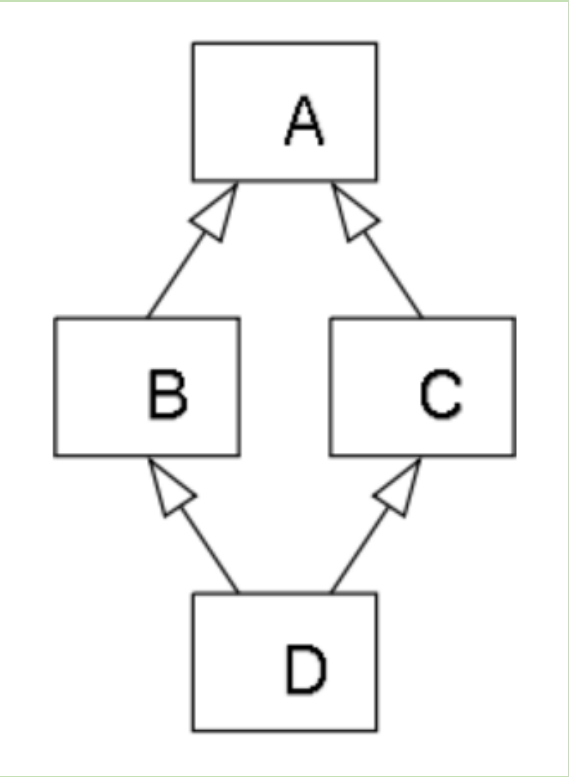
```
long x = 0xd76e50;
Point* point3 = reinterpret_cast<Point*>(x);
```

Множественное наследование

- C++ поддерживает множественное наследование без ограничений на классы-потомки

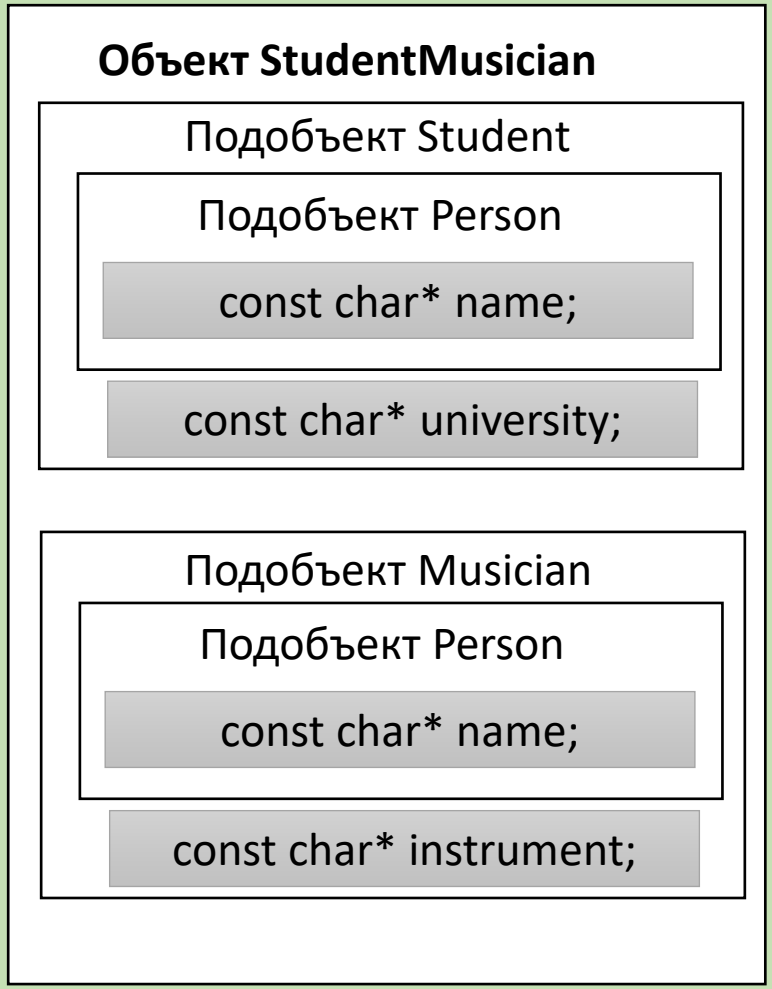
```
class Printer { .. };
class Scanner { .. };
class MFU: public Printer, public Scanner { .. };
```

- Множественное наследование может приводить к проблемам:
  - конфликт имен;
  - проблема «алмаз смерти» (diamond problem);



Проблемы множественного наследования

```
struct Student : Person { .. };
class Musician :  public Person { .. };
class StudentMusician: public Student,
                       public Musician { .. };
```



- «Алмаз смерти» образуется, когда базовые классы, участвующие в множественном наследовании, имеют общего предка (в примере – Person)
- В этом случае поля такого предка дублируются
- «Прямая» работа с элементами предка невозможна из-за неоднозначности вызовов

```
carlos.name = "Carlos";
```

- Полиморфизм по отношению к предку не возможен.

```
Person& p = carlos;
```

«Виртуальное наследование»

- Виртуальное наследование (virtual inheritance) предотвращает появление множественных объектов базового класса в иерархии наследования.

```
struct Student :  virtual public Person { .. };
class Musician :  virtual public Person { .. };
class StudentMusician: public Student,
                       public Musician { .. };
```

- Для объектов класса StudentMusician информация виртуального предка Person хранится в одном экземпляре
- Снимается проблема конфликта имен
- Разрешается вызов конструктора Person *напрямую*