Объектно-ориентированное программирование в C++

Наследование

Полиморфизм адресов

Доступ к элементам при наследовании

Конструкторы и деструкторы при наследовании

Наследование

- Наследование – один важнейших принципов (возможностей) ООП
- Наследование позволяет удобным образом вводить новые типы на базе существующих, расширяя их описание и функциональность

```
struct Animal { .. };
```

```
struct Dog : Animal {};
```

```
struct Cat : Animal {};
```

- Наследование – это переход от *общего* к *частному*, уточнение описания объекта
- Наследование неразрывно связано с *полиморфизмом* и возможностью унифицированной работы со всей иерархией объектов

```
void Show(Animal& a) { }
```

Наследование

- С точки зрения расположения данных в памяти наследование это добавление новых полей данных к ранее описанной структуре

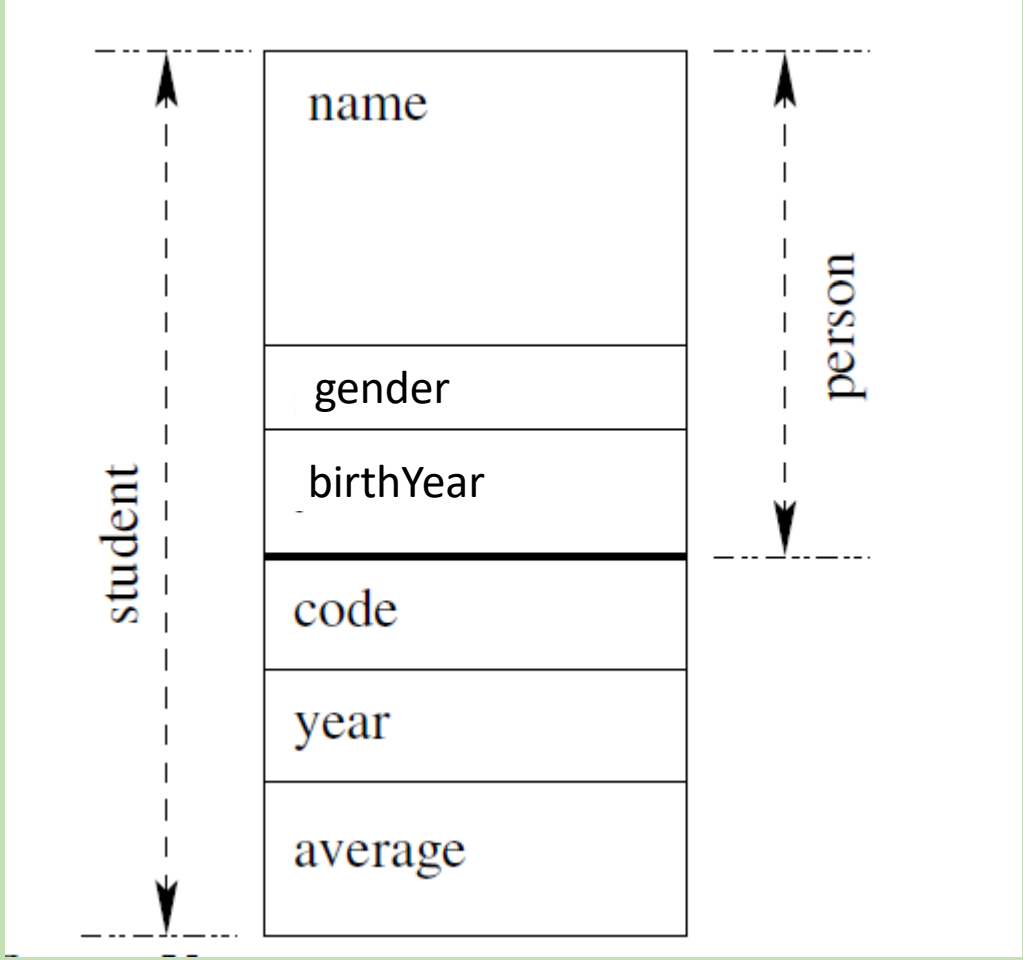
- Структура Person – базовая (родительская)
- Структура Student – производная (потомок)

```
struct Person
{
    std::string name;
    char gender; // f or m
    int birthYear;
};

struct Student : Person {
    int code;
    int year;
    float average;
};
```

Размещение в памяти

- Поля структуры Student, унаследованные от Person, в памяти располагаются на тех же местах (по тем же смещениям), как и в структуре Person
- Объект производного класса содержит в себе объект базового класса как составную часть
- Поэтому возможен переход от Student к Person



Приведение типов

- Разрешено неявное преобразование адреса переменной производного типа в адрес объекта родительского типа:

```
Student s;
Person* p = &s;
Person& p_ref = s;
```

- Возможность неявного преобразования можно использовать при вызове функции

```
void Show(Person& p)
{
    cout << "Name: " << p.name << endl;
    cout << "Birth year: " << p.birthYear << endl;
    cout << "Gender: " << p.gender << endl;
}
```

```
Student s1;
Show(s1);
```

- Такое преобразование называют *полиморфизмом адресов*

Модели защиты при наследовании

- В C++ различаются типы наследования: открытое наследование (public) и закрытое наследование (private)
- Модель защиты указывается в заголовке класса рядом с базовым классом:

```
class B : public A {};
class C : private A {};
```

- *Открытое наследование*: все открытые элементы, объявленные в классе A, будут доступны и для объектов класса B
- *Закрытое наследование*: все открытые элементы класса A в классе C становятся закрытыми, т.е. доступны только внутри класса C; информация о наследовании класса C от класса A не доступна (неявное преобразование от C к A не выполняется)
- По умолчанию: для структур реализуется открытое наследование, для классов - закрытое

Доступ к элементам при (открытом) наследовании

- Производный класс является недружественным типом по отношению к базовому классу и не имеет доступа к закрытым элементам базового класса
- Модификатор доступа **protected** позволяет ввести элементы, которые доступны в рамках класса и производных типах.

```
class Base
{
private:
    int x, y;
    void setX(int xx) { x = xx; }
protected:
    void setY(int yy) { y = yy; }
public:
    int getX() { return x; }
    int getY() { return y; }
};
```

```
class Derived : public Base
{
    public:
        void Update(int xx, int yy)
        {
            setX(xx);
            setY(yy);
        }
        void Get(int& xx, int& yy)
        {
            xx = getX(); yy = getY();
        }
};
```

Конструирование и деструкция «наследника»

- Объект производного типа содержит в себе элементы базового типа (некоторые из них могут быть непосредственно недоступны «наследнику»)
- При создании объекта производного типа должен отработать и конструктор базового типа (*раньше*)
- При уничтожении объекта производного типа должен отработать и деструктор базового типа (*позже*)
- В случае уничтожения: компилятор автоматически добавляет в код деструктора производного класса (в самый его конец) вызов деструктора базового класса
- В случае создания: компилятор может автоматически вызвать конструктор без параметров (по умолчанию) базового класса перед кодом конструктора производного класса

Конструкторы при наследовании

- В случае создания может возникнуть необходимость в вызове определенного конструктора; можно явно обращаться к конструктору базового класса перед кодом конструктора производного класса

```
class Base
{
public:
    Base(int xx, int yy) : x(xx), y(yy) {}
private:
    int x, y;
    ..
};
```

```
class Derived : public Base
{
    int z;
    public:
        Derived() : Base(2, 4), z(5) {}

        ..
};
```