Дополнительные возможности в ООП C++

C++ 11: override и final

Ограничение доступа к элементам базового типа

«Наследование» конструкторов

Механизм обработки исключений

Сокрытие при множественном наследии

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

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

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

```
cout << boolalpha;

Bullterrier bullie("Bullie");
bullie.angry = true;


// Метод не доступен из-за неоднозначности
cout << bullie.isAngry();


cout << bullie.Animal::isAngry();
cout << bullie.Dog::isAngry();


bullie.Dog::Animal::angry = true;
cout << bullie.Dog::Animal::isAngry();
```

C++ 11: «наследование» конструкторов

- При наследовании часто возникает ситуация, когда в производном классе вводятся конструкторы, совпадающие по сигнатуре с конструкторами базового класса, которые делегируют работу этим конструкторам

```
class Teacher : public Person
{
public:
    Teacher(const char* name) : Person(name) {}
    Teacher(const char* name, int birth) : Person(name, birth) {}
};
```

- В C++11 появилась возможность «наследования» конструкторов для достижения эффекта сквозной передачи параметров

```
class Teacher : public Person
{
public:
    using Person::Person;
};
```

C++ 11: override

- В C++ 11 введен модификатор override, который можно использовать при переопределении виртуальных методов в производных классах
- Модификатор используется для **явного указания** на то, что метод является *переопределением*, а не новым методом класса (*сокрытием*)
- Компилятор сможет указать на ошибку, если есть какое-то несоответствие методов в производном и базовом классах (ошибка в имени, аргументах), а также если в базовом классе метод не обозначен как виртуальный.

```
struct Student : Person
{
    string toString() const override;
    ..
};
```

- Использование модификатора override никак не влияет на эффективность или производительность программы, но помогает избежать непреднамеренных ошибок.

C++ 11: final

- Модификатор final для переопределенного метода позволяет запретить дальнейшее переопределение в производных классах.

```
struct Student : public Person
{
  string toString() const override final;
}
```

- Модификатор final для класса позволяет запретить наследование

```
class Teacher final : public Person
{ ..
};
```

C++ 11: public -> private, protected -> public

- Язык C++ предоставляет возможность изменить спецификатор доступа родительского члена в дочернем классе(*).
- В стандарте C++ 11 это делается с помощью «using-объявления» (ранее для этого использовался подход access-объявлений).

```
class Car : public Vehicle {
    public:
        Vehicle::maxSpeed;
        Vehicle::setWeight;
        ..
```

- *Изменять спецификаторы можно только для тех элементов, к которым есть доступ у производного класса

```
class Vehicle {
    protected:
        float weight;
        int maxSpeed;
        void setWeight(float w);
};
class Car : public Vehicle {
    public:
        string modelName;
        string manufactoryName;
        using Vehicle::maxSpeed;
        using Vehicle::setWeight;
    private:
        using Vehicle::weight;
};
```

«Удаление» функций базового класса

- Для ограничения доступа к функциям базового типа через объект производного типа можно использовать ключевое слово delete

```
class Car : public Vehicle {
    void setParams(float, int) = delete;
    using Vehicle::setSpeed;
    void setWeight(float w) { Vehicle::setWeight(w); }
    ..
};
```

- В этом случае доступ к методу возможен только с использованием полного имени (с указанием имени базового класса).

```
car.setWeight(1455);
car.setParams(2340, 200);
car.setSpeed(100);
car.Vehicle::setParams(2340, 200);
car.Vehicle::setSpeed(100);
car.Vehicle::weight = 3500;
```

“Исключительные” ситуации

- Каким образом сигнализировать об ошибке в функции?
  - Возвращать предопределенное значение (0, nullptr, 666)

```
// Если нельзя построить прямоугольник, возвращаем определенный элемент
// Например, возвращаем Rectangle(0,0,0,0)
Rectangle getRectangle(Point* points, int size);
// Если нельзя построить прямоугольник, возвращаем nullptr (0)
Rectangle* getRectangleOrNull(Point* points, int size);
```

```
// Для не-числа возвращается 0, как и для «0» ☹
char s[] = "afdddf";
int v = atoi(s);
```

  - Использовать несколько «выходных» параметров

```
// Если можно создать, заполняем существующий rect
bool getRectangleIfPosible(Point points[], int size, Rectangle& rect);
// Если можно построить, создаем объект rect в динамической памяти
bool getRectangleIfPosible(Point points[], int size, Rectangle** rect);
```

```
bool tryParse(const char* c_str, int& n);
```

Механизм *исключений*

- Исключение – особый способ завершения функции в ответ:
  - на непредвиденное или аварийное событие;
  - ошибку в логике работы;
- При возникновении ситуации, которая воспринимается как «ошибочная», можно сгенерировать исключение:

```
throw exception();
```

- При возникновении или возбуждении исключения – все активные функции досрочно завершаются одна за другой, до тех пор пока не найдется такая, которая сможет справиться с данным типом исключения

```
int parseInt(const char* c_str)
{
    int n = atoi(c_str);
    if(!n && strcmp(c_str, "0"))
        throw "It's not a number";
    return n;
}
```

```
int main() {
  int x = f3();
  ..
}
```

```
int f3() {
  int number = parseInt("xxx11");
  return number * 10;
}
```

Механизм обработки исключений

- Фрагмент кода, в котором возможно возникновение исключительной ситуации, обрамляется специальным образом (**try**-блок) и снабжается указаниями о том, как следует обрабатывать исключения (**catch**-блок)

```
try
{
    int number = parseInt("xxx11");
    number *= 10;
}
catch(const char* str)
{
    std::cerr << str << '\n';
}
```

- Глубина вызовов функций в «опасном» фрагменте не имеет значения

catch-блоки

- Catch-обработчики рассматриваются строго по порядку, один за другим.
- В зависимости от типа исключения может сработать какой-то один из catch-обработчиков или **ни одного**, если соответствия по типу не найдено.
- Если исключение не обработано ни в одном catch-блоке, то оно распространяется дальше по коду (досрочно завершаем текущую функцию и ищем try-catch во «родительской» функции)

```
try
{
    f("xxx11");
}
catch(int n)
{
   cout << "Number of error: " << n << endl;
}
catch(const char* str)
{
   cout << str << '\n';
}
catch(...)
{
   cout << "Undefined error" << endl;
}
```

Механизм исключений

- Модель обработки исключений является *невозобновляемой*: при возникновении исключения автоматически уничтожаются все локальные переменные, введенные в try-блоке (с учетом вызовов функций); вызываются деструкторы локальных переменных
- Catch-обработчики отлавливают исключения только в try-блоке
- После выполнения catch-блока управление передается первому оператору после последнего catch-блока
- Можно *перевбросить* исключение в catch-блоке для последующей обработки

```
catch(int n)
{
    if(n == -1)
        throw;
    cout << "Number of error: " << n << endl;
}
```

std::exception

- При работе с исключениями можно использовать любой тип (*обладающий конструктором копирования).
- Рекомендуемый подход – использование специального типа **exception** и производных от него типов (можно вводить свои производные классы).
- В этом случае можно вводить catch-блоки для каждого отдельного типа исключения

```
try
 {
        /* code */
 }
 catch(bad_cast) { .. }
 catch(out_of_range) { .. }
 catch(const std::exception& e)
 {
       cerr << e.what() << '\n';
 }
```

- Из-за возможности неявного преобразования от производного класса к базовому нужно выстроить catch-блоки в правильном порядке: от самого специфичного типа к самому общему (**exception**)