# Тема 2.1 Пакетирование, наследование и полиморфизм

## Наследование в C++

Наследование классов – мощная возможность в объектно-ориентированном программировании. Оно позволяет создавать производные классы (классы наследники), взяв за основу все методы и элементы базового класса (класса родителя). Таким образом, экономится масса времени на написание и отладку кода новой программы. Объекты производного класса свободно могут использовать всё, что создано и отлажено в базовом классе. При этом, мы можем в производный класс, дописать необходимый код для усовершенствования программы: добавить новые элементы, методы и т.д.. Базовый класс останется нетронутым.

Ниже приведен простой код программы. В этой программе созданы два класса: базовый — Class1 и производный от него Class2.

```
#include <iostream>
using namespace std;

class Class1{        // базовый класс
protected:           // спецификатор доступа к элементу znachenie
    int znachenie;
public:
    Class1() {znachenie = 0;}
    Class1(int vvod){znachenie = vvod;}
    void vivod_znach(){cout << znachenie << endl;}
};

class Class2 : public Class1{    // производный класс
public:

    Class2() : Class1(){}   // конструктор класса Class2 вызывает конструктор
класса Class1

    Class2(int vvod_2) : Class1 (vvod_2){} // vvod_2 передается в конструктор
с параметром класса Class1

    void ZnachSqr (){znachenie* = znachenie;} // возводит znachenie в
квадрат. Без спецификатора доступа protected эта функция не могла бы изменить
значение znachenie

};

int main(){

    Class1 object1(3);      // объект базового класса
    cout << "znachenie object1 = ";
    object1.vivod_znach();
```

```
    Class2 object2(4);    // объект производного класса
    cout << "znachenie object2 = ";
    object2.vivod_znach();  // вызов метода базового класса

    object2.ZnachSqr();        // возводим znachenie в квадрат
    cout << "kvadrat znacheniya object = ";
    object2.vivod_znach();

    //object1.ZnachSqr(); //  базовый класс не имеет доступа к методам производного класса
    return 0;
}
```

## 1 Модификаторы доступа при наследовании и их роль

Класс – это тип данных, объединяющий данные и методы их обработки. Инкапсуляция позволяет скрыть информацию, к которой не предусмотрен прямой доступ из других классов. В языке C++ предусмотрено несколько модификаторов доступа, которые определяют кто имеет право использовать следующие за ними объявления членов класса и некоторых других элементов языка.

- public –значает, что следующие за ним определения доступны всем.
- private – делает следующие за ним определения доступными только внутри класса.
- protected – защищённые методы или переменные доступны только внутри класса, где они были объявлены и из его производных классов.

Функции-члены (их прототип определены в классе. Они меняют доступ к закрытым полям класса, и при обращении к ним используется расширение доступа (может это можно назвать принадлежностью класса). В главной функции main обращение происходит через точку как к методу класса или стрелочку (указатель) после элемента типа класса). Функции (можно свободно описать до main и обращаться на прямую из main)

Чтобы было наглядней, отличия спецификаторов доступа можно отобразить в таблице:

**Таблица 1**

| | private | protected | public |
|---|---|---|---|
| **Доступ из тела класса** | открыт | открыт | открыт |
| **Доступ из производных классов** | закрыт | открыт | открыт |
| **Доступ из внешних функций и классов** | закрыт | закрыт | открыт |

При наследовании public в класс-потомок передаются все поля в таком виде в котором они записаны в родителе. private поле тоже туда передается, но напрямую наследник ничего с ней сделать не может.

Используя public наследование мы передаем потомкам всё что есть в основном классе в таком виде, как и записано в основном классе. Получаем клон основного класса. Разница в том что элементы основного класса к элементам своего клона отношения не имеют.

Используя private наследование можно создать первого потомка от которого дальнейшее наследование будет бессмысленно. Если первый потомок получает возможность работы с некоторыми элементами, переданными по механизму наследования, то потомки первого потомка таких возможностей не получают. Кроме того, потомки первого потомка даже лишены возможности узнавать кто их первый родитель. Предполагается, что потомки класса B не должны даже знать о существовании класса A (либо потомков класса B вообще не должно быть).

Используя protected наследование, программист предполагает, что внутри всех потомков и потомков потомков и потомков потомков потомков и т.д. будут использоваться только такие элементы, передаваемые механизмом наследования, которые будут защищены от внешнего воздействия извне своих классов.

## 2 Множественное наследование

C++ позволяет порождать класс из нескольких базовых классов. Один класс может наследовать атрибуты двух и более классов одновременно. Для этого используется список базовых классов, в котором каждый из базовых классов отделен от других запятой. Общая форма множественного наследования имеет вид:

```
class имя_порожденного_класса: список базовых классов {...};
```

В следующем примере класс Z наследует оба класса X и Y:

```
#include <iostream.h>
class X {
 protected:
       int a;
 public:
 void make_a(int i) { a = i; }
};

class Y {
 protected:
       int b;
 public:
 void make_b(int i) { b = i; }
};

// Z наследует как от X, так и от Y
```

```
class Z: public X, public Y {
 public:
 int make_ab() { return a*b; }
};

int main(){
 Z i;
 i.make_a(10);
 i.make_b(12);
 cout << i.make_ab();
 return 0;
}
```

Поскольку класс Z наследует оба класса X и Y, то он имеет доступ к публичным и защищенным членам обоих классов X и Y. В предыдущем примере ни один из классов не содержал конструкторов. Однако ситуация становится более сложной, когда базовый класс содержит конструктор. Например, изменим предыдущий пример таким образом, чтобы классы X, Y и Z содержали конструкторы:

```
#include <iostream.h>
class X {
protected:
      int a;
public:
      X() {a = 10; cout << "Initializing X\n";}
};

class Y {
protected:
      int b;
public:
      Y() {cout << "Initializing Y\n"; b = 20;}
};
// Z наследует как от X, так и от Y
class Z: public X, public Y {
public:
      Z() { cout << "Initializing Z\n"; }

int make_ab() { return a*b; }
};

int main(){
      Z i;
      cout << i.make_ab();
      return 0;
}
```

Программа выдаст на экран следующий результат:

```
Initializing X
Initializing Y
```

```
Initializing Z
200
```

Обратим внимание, что конструкторы базовых классов вызываются в том порядке, в котором они указаны в списке при объявлении класса Z. В общем случае, когда используется список базовых классов, их конструкторы вызываются слева направо. Деструкторы вызываются в обратном порядке — справа налево.

## 3 Виртуальные базовые классы

В C++ ключевое слово virtual используется для объявления виртуальных функций, которые будут переопределены в производных классах. Однако ключевое слово virtual также имеет другое использование, позволяющее определить виртуальный базовый класс. Для того чтобы понять, что из себя представляет виртуальный базовый класс и почему ключевое слово virtual имеет второе значение рассмотрим короткую некорректную программу:

```
// программа содержит ошибку и не будет компилироваться
#include <iostream.h>
class base {
      public:
            int i;
};
// d1 наследует base.
class d1 : public base {
      public:
            int j;
};
// d2 наследует base.
class d2 : public base {
      public:
            int k;
};
/* d3 наследует как d1, так и d2 . Это означает, что в d3 имеется две
копии base! */
class d3 : public d1, public d2 {
      public:
            int m;
};

int main(){
      d3 d;
      d.i = 10; // неопределенность, какое i?
      d.j = 20;
      d.k = 30;
      d.m = 40;
      cout << d.i << " "; // также неопределенность, какое i?
      cout << d.j << " " << d.k << " ";
      cout << d.m;
```

```
        return 0;
    }
```

Как показывает комментарий в данной программе, оба класса d1 и d2 наследуют класс base. Однако класс d3 наследует оба класса d1 и d2. Это означает, что в классе d3 представлены две копии класса base. Поэтому в выражении типа: «`d.i = 20;`» не ясно, какое именно i имеется в виду — относящееся к d1 или же относящееся к d2? Поскольку имеется две копии класса base в объекте d, то там имеются также две переменные d.i. Как видно, инструкция является двусмысленной в силу описанного наследования.

Имеется два способа исправить программу. Первый заключается в использовании оператора области видимости для переменной i с дальнейшим выбором вручную одного из i. Например, следующая версия программы компилируется и исполняется так, как это необходимо:

```
#include <iostream.h>
class base {
 public:
      int i;
};
// d1 наследует base.
class d1 : public base {
 public:
      int j;
};
// d2 наследует base.
class d2 : public base {
 public:
      int k;
};
/* d3 наследует как d1, так и d2. Это означает, что в d3 имеется две
копии base! */
class d3 : public d1, public d2 {
 public:
      int m;
};

int main(){
 d3 d;
 d.d2::i = 10; // область видимости определена, используется i для d2
 d.j = 20;
 d.k = 30;
 d.m = 40;
cout <<d.d2::i<<" ";//область видимости определена, используется i для d2
 cout << d.j << " " << d.k << " ";
 cout << d.m;
 return 0;
}
```

Как можно видеть, используя оператор области видимости « :: », в программе вручную выбирается версия d2 класса base. Тем не менее, данное решение порождает более глубокие вопросы: что если требуется только одна копия класса base? Имеется ли какой-либо способ предотвратить вклю-чение двух копий в класс d3? Как можно было догадаться, ответ на этот вопрос положительный. Решение достигается путем использования виртуального базового класса.

Когда два или более класса порождаются от одного общего базового класса, можно предотвратить включение нескольких копий базового класса в объект-потомок этих классов путем объяв-ления базового класса виртуальным при его наследовании. Например, ниже приведена другая версия предыдущей программы, в которой d3 содержит только одну копию класса base:

```
#include <iostream.h>
class base {
 public:
       int i;
};
// d1 наследует base как virtual
class d1 : virtual public base {
 public:
       int j;
};
// d2 наследует base как virtual
class d2 : virtual public base {
 public:
       int k;
};
/* d3 наследует как d1 так и d2. Тем не менее в d3 имеется только одна
копия base! */
class d3 : public d1, public d2 {
 public:
       int m;
};

int main(){
 d3 d;
 d.i = 10; // неопределенности больше нет
 d.j = 20;
 d.k = 30;
 d.m = 40;
 cout << d.i << " "; // неопределенности больше нет
 cout << d.j << " " << d.k << " ";
 cout << d.m;
 return 0;
}
```

Как видно, ключевое слово virtual предшествует спецификации наследуемого класса. Теперь оба класса d1 и d2 наследуют класс base как

виртуальный. Любое множественное наследование с их участием порождает теперь включение только одной копии класса base. Поэтому в классе d3 имеется только одна копия класса base, и, следовательно, d.i = 10 теперь не является двусмыслен-ным выражением.

Необходимо иметь в виду еще одно обстоятельство: хотя оба класса d1 и d2 используют класс base как виртуальный, тем не менее всякий объект класса d1 или d2 будет содержать в себе base. Например, следующий код абсолютно корректен:

```
// определение класса типа d1
d1 myclass;
myclass.i = 100;
```

Обычные и виртуальные базовые классы отличаются друг от друга только тогда, когда какой- либо объект наследует базовый класс более одного раза. При использовании виртуального базового класса только одна копия базового класса содержится в объекте. В случае использования обычного базового класса в объекте могут содержаться несколько копий.