Наследование классов — очень 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. Эту тему вполне возможно освоить новичкам. Необходимо только познакомиться с синтаксисом и некоторыми особенностями. Ниже приведен простой код программы, который мы детально разберем под листингом. В этой программе созданы два класса: базовый — FirstClass  и производный от него SecondClass.

Пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | #include <iostream>  using namespace std;    class FirstClass    // базовый класс  {  protected:          // спецификатор доступа к элементу value      int value;  public:      FirstClass()      {          value = 0;      }        FirstClass( int input )      {          value = input;      }        void show\_value()      {          cout << value << endl;      }  };    class SecondClass : public FirstClass   // производный класс  {  public:      SecondClass() : FirstClass ()   // конструктор класса SecondClass вызывает конструктор класса FirstClass      {}        SecondClass(int inputS) : FirstClass (inputS)   // inputS передается в конструктор с параметром класса FirstClass      {}        void ValueSqr () // возводит value в квадрат. Без спецификатора доступа protected эта функция не могла бы изменить значение value      {          value \*= value;      }  };    int main()  {      setlocale(LC\_ALL, "rus");        FirstClass F\_object(3);     // объект базового класса      cout << "value F\_object = ";      F\_object.show\_value();        SecondClass S\_object(4);    // объект производного класса      cout << "value S\_object = ";      S\_object.show\_value();  // вызов метода базового класса        S\_object.ValueSqr();        // возводим value в квадрат      cout << "квадрат value S\_object = ";      S\_object.show\_value();        //F\_object.ValueSqr();      // базовый класс не имеет доступа к методам производного класса        cout << endl;      return 0;  } |

Разбирать пример будем по-порядку. Ранее мы работали только со спецификаторами доступа private и  public.  В **строке 6** мы встретили новый для нас спецификатор доступа protected. Он отличается от private тем, что разрешает доступ к элементам базового класса из производных классов. Если бы элемент value находился в поле private, то доступ к нему был бы закрыт и мы бы не могли изменить его значение через объект класса SecondClass, используя функцию ValueSqr() , определённую в **строках 34 — 37**.

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

|  |  |  |  |
| --- | --- | --- | --- |
|  | **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 |

Если вы создаёте класс, который в дальнейшем планируете использовать, как базовый, то объявляйте в нём поле protected вместо private. Иначе объекты производного класса не смогут обращаться к элементам базового.

Ниже, в **строках 8 — 22**, определены методы базового класса. [Конструктор](http://cppstudio.com/post/6964/) без параметров FirstClass(), конструктор с параметром FirstClass( int input )  и метод void show\_value(), который выводит значение value на экран.

Определение производного находится в **строках 25 — 38**. Синтаксис наследования такой — class   Имя\_Производного\_Класса    :    спецификатор доступа    Имя\_Базового\_Класса   {  } ; Двоеточие :  не путайте с двойным двоеточием :: (определение области действия).  Используя этот оператор мы показываем, наследником какого класса является производный класс.

Важной особенностью производного класса, является то, что хоть он и может использовать все методы и элементы полей protected и public базового класса, но он не может обратиться к конструктору с параметрами. Если конструкторы в производном классе не определены, при создании объекта сработает конструктор без аргументов базового класса. А если нам надо сразу при создании объекта производного класса внести данные, то для него необходимо определить свои конструкторы. В нашем примере показано, как же мы всё-таки можем использовать уже готовые конструкторы базового класса, чтобы не набирать код конструкторов снова — **строки 28 — 32**. Для этого при определении конструктора производного класса после его имени следует поставить оператор : и  имя конструктора базового класса, который необходимо вызвать, при создании объекта производного класса — SecondClass() : FirstClass (){}. Тело конструктора оставляем пустым т.к. всю работу проделает конструктор базового класса.  В случае конструктора с параметром, этот параметр мы передаем в конструктор с параметром базового класса SecondClass(int inputS) : FirstClass (inputS){} — **строка  31**.

В main-функции создаем объекты базового и производного классов — FirstClass F\_object(3); и  SecondClass S\_object(4); и отображаем их значения value на экран. Как видите в **строке 50**,  объект производного класса без проблем обращается к методу show\_value() базового класса. Так, будто это его собственный метод. Ниже вызываем метод, который возводит значения value производного класса в квадрат. И выводим это изменённое значение на экран. А вот если мы захотим вызвать этот метод —  F\_object.ValueSqr(); —  для объекта базового класса, компилятор нам этого не позволит сделать и выдаст ошибку. Это еще одна важная особенность — производный класс имеет доступ к базовому классу, а базовый класс, даже «не знает» о существовании производного и не может пользоваться его кодом.

 Результат работы программы:

CppStudio.com

value F\_object = 3  
value S\_object = 4  
квадрат value S\_object = 16

В виде списка приведу  основную информацию о наследовании классов, которую важно знать:

* Наследование — это определение производного класса, который может обращаться ко всем элементам и методам базового класса за исключением тех, которые находятся в поле private;
* Производный класс еще называют потомком или подклассом, а базовый — родитель или надкласс;
* Синтаксис определения производного класса:
* class   Имя\_Производного\_Класса    :    спецификатор доступа    Имя\_Базового\_Класса   { /\*код\*/ } ;
* Производный класс имеет доступ ко всем элементам и методам базового класса, а базовый класс может использовать только свои собственные элементы и методы.
* В производном классе необходимо явно определять свои конструкторы, деструкторы и [перегруженные операторы](http://cppstudio.com/post/7958/) присваивания  из-за  того,  что  они  не  наследуются  от  базового  класса. Но  их  можно вызвать явным образом при определении конструктора, деструктора или перегрузки оператора присваивания производного класса, например таким образом (для конструктора): Конструктор\_Производного\_Класса (/\*параметры\*/)  :  Конструктор\_Базового\_Класса ( /\*параметры\*/) { } .
* Еще один важный момент при наследовании — [перегруженные функции](http://cppstudio.com/post/406/)-методы  класса потомка. В данном примере мы его не рассматривали. Но чтобы вы знали, если в  классе родителе и в его классах потомках встречаются методы с одинаковым именем, то для объектов  класса потомка компилятор будет использовать методы именно класса потомка. Перегруженные методы класса потомка, могут вызывать методы класса родителя. В таком случае важно помнить, что необходимо правильно определить область действия с помощью оператора :: .Иначе компилятор воспримет это, как вызов функцией самой себя.  Наглядно, если бы мы перегрузили в классе SecondClass функцию show\_value() — это  выглядело бы так:

|  |  |
| --- | --- |
| 1  2  3  4  5 | void show\_value()  {      if(value != 0)          FirstClass :: show\_value();  } |

Эта запись указывает компилятору  — если значение value не равно нулю — вызвать метод show\_value() класса FirstClass. А он в свою очередь, отобразит это значение на экране.