Глава 7

Наследование

Ранее в этой книге вы познакомились с концепцией наследования. Сейчас пришло время осветить эту тему более детально. Наследование – это один из базовых принципов ООП, и потому является одним из важнейших инструментов C++. В C++ наследование используется не только для поддержки иерархии классов, но, как вы узнаете из главы 10, и для поддержки другого важнейшего инструмента ООП – полиморфизма.

Материал, который приведен в этой главе, включает в себя следующие темы: управление доступом к базовому классу. Спецификатор доступа **protected**, множественное наследование, передача аргументов конструкторам базового класса, виртуальные базовые классы.

7.1 Управление доступом к базовому классу.

Когда один класс наследуется другим, используется следующая основная форма записи:

class ***имя_прозводного_класса: сп_доступа имя_базового _класса {***
//…
}

Здесь ***сп_доступа*** - это один из трёх ключевых слов: **public, private** или **protected.** Обсуждение спецификатора доступа **protected** отложим до следующего раздела главы. Здесь рассмотрим спецификаторы **public** и **private**.

Спецификатор доступа (access specifier) определяет то, как элементы базового класса (base class) наследуются производным классом (derived class). Если спецификатором доступа наследуемого базового класса является ключевое слово **public**, то все открытые члены базового класса остаются открытыми и в производном. Если спецификатором доступа наследуемого базового класса является ключевое слово **private**, то все открытые члены базового класса в производном классе становятся закрытыми. В обоих случаях все закрытые члены базового класса в производном классе остаются закрытыми и недоступными.

Важно понимать, что если спецификатором доступа является ключевое слово **private**, то хотя открытые члены базового класса становятся закрытыми в производном, они остаются доступными для функций – членов производного класса.

Технически спецификатор доступа не обязателен. Если спецификатор доступа не указан и производный класс определён с ключевым словом **class**, то базовый класс по умолчанию наследуется как закрытый. Если спецификатор доступа не указан и производный класс определён с ключевым словом **struct,** то базовый класс по умолчанию наследуется как открытый. Тем не менее, для ясности большинство программистов предпочитают явное задание спецификатора доступа.

Примеры

1. Здесь представлены базовый класс и наследующий его производный классы (наследование со спецификатором **public**):

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

class base {
   int x;
public:
   void setx(int n) { x = n; }
   void showx() { cout << x << '\n'; }
};

// Класс наследуется как открытый
class derived : public base {
   int y;
public:
   void sety(int n) { y = n; }
   void showy() { cout << y << '\n'; }
};

int main()
{
   derived ob;

   ob.setx(10); // доступ к члену базового класса
   ob.sety(20); // доступ к члену производного класса

   ob.showx(); // доступ к члену базового класса
   ob.showy(); // доступ к члену производного класса

   return 0;
}
```

Как показано в программе, поскольку класс **base** наследуется как открытый, открытые члены класса **base** – функции **setx()** и **showx()** – становятся открытыми производного класса **derived**и поэтому доступны из любой части программы. Следовательно, совершенно правильно вызывать эти функции из функции **main()**.

2. Важно понимать, что наследование производным классом базового как открытого совсем не означает, что для производного класса станут доступными закрытые члены базового. Например, это небольшое изменение в классе **derived** из предыдущего примера неправильно:

```
class base {
int x;
public:
void setx(int x) {x = n;}
void showx() {cout << x << '\n';}
};

//Класс наследуется как открытый
class derived: public base{
int y;
public:
void sety(int n) {y = n;}
```

```
/*Закрытые члены базового класса недоступны. x – это закрытый член базового класса и поэтому внутри производного класса он недоступен*/

void show_sum() {cout << x+y << '\n';}   //Ошибка!!!

void showy() {cout << y <<'\n';}
};
```

Здесь в производном классе **derived** сделана попытка доступа к переменной **x**, которая является закрытым членом базового класса **base**. Это неверно, поскольку закрытые члены базового класса остаются закрытыми, *независимо от того, как он наследуется.*

3. Ниже представлена слегка изменённая версия программы из примера 1. Базовый класс **base** наследуется как закрытый, т.е. с ключевым словом **private**. Такое изменение, как показано в комментариях, при компиляции ведёт к ошибке.

```
// В этой программе есть ошибка
#include <iostream>
using namespace std;

class base {
   int x;
public:
   void setx(int n) { x = n; }
   void showx() { cout << x << '\n'; }
};

// Класс наследуется как закрытый
class derived : private base {
   int y;
public:
   void sety(int n) { y = n; }
   void showy() { cout << y << '\n'; }
};

int main()
{
   derived ob;

   ob.setx(10); // ОШИБКА - теперь закрыто для производного класса
   ob.sety(20); // правильный доступ к члену производного класса

   ob.showx(); // ОШИБКА - теперь закрыто для производного класса
   ob.showy(); // правильный доступ к члену производного класса

   return 0;
}
```

Как отражено в комментариях к этой (неправильной программе), функции **showx()** и **setx()** становятся закрытыми в производном классе и недоступными вне его.

Запомните, что функции **showx()** и **setx()** в базовом классе **base** по-прежнему остаются открытыми независимо от того, как они наследуются производным классом. Это означает, что объект типа **base** мог бы получить доступ к этим функциям в любом месте программы. Однако для объектов типа **derived** они становятся закрытыми. Например, в данном фрагменте:

```
base base_ob;
        base_ob.setx(1); //правильно, поскольку объект base_ob имеет тип base
```

вызов функции **setx()** правилен, поскольку функция **setx()** – это открытый член класса **base**.

4. Как мы уже узнали, хотя открытые члены базового класса при наследовании с использованием спецификатора **private** в производном классе становятся закрытыми, *внутри* производного класса они остаются доступными. Например, ниже представлена исправленная версия предыдущей программы:

```
// Исправленная версия программы
#include <iostream>
using namespace std;

class base {
   int x;
public:
   void setx(int n) { x = n; }
   void showx() { cout << x << '\n'; }
};

// Класс наследуется как закрытый
class derived : private base {
   int y;
public:
   // переменная setx доступна внутри класса derived
   void setxy(int n, int m) { setx(n); y = m; }
   // переменная showx доступна внутри класса derived
   void showxy() { showx(); cout << y << '\n'; }
};

int main()
{
   derived ob;

   ob.setxy(10, 20);

   ob.showxy();

   return 0;
}
```

В данном случае функции **showx()** и **setx()** доступны *внутри* производного класса, что совершенно правильно, поскольку они являются закрытыми членами этого класса.

7.2 Защищённые члены класса

Как вы узнали из предыдущего раздела, у производного класса нет доступа к закрытым членам базового. Это означает. Что если производному классу необходим доступ к некоторым членам базового, то эти члены должны быть открытыми. Однако возможна ситуация, когда необходимо, чтобы члены базового класса, оставаясь закрытыми, были доступны для производного класса. Для реализации этой идеи в C++ включён спецификатор доступа **protected** (защищённый).

Спецификатор доступа **protected** эквивалентен спецификатору **private** с единственным исключением: защищённые члены базового класса доступны для членов всех производных классов этого базового класса. Вне базового класса или производных классов защищённые члены недоступны.

Спецификатор доступа **protected** может находиться в любом месте объявления класса, хотя обычно его располагают после объявления закрытых членов (задаваемых по умолчанию) и перед объявлением открытых членов. Ниже показана полная основная форма объявления класса:

class ***имя_класа*** **{**
    // закрытые члены
protected: ***// необязательный спецификатор***
    // защищённые члены
public:
    // открытые члены
};

Когда базовый класс наследуется производным классом как открытый (**public**), защищённый член базового класса становится защищённым членом производного класса. Когда базовый класс наследуется как закрытый (**private**), то защищённый член базового класса становится закрытым членом производного класса.

Базовый класс может также наследоваться производным классом как защищённый (**protected**). В этом случае открытые и защищённые члены базового класса становятся защищёнными членами производного класса. (Естественно, что закрытые члены базового класса остаются закрытыми, и они не доступны для производного класса.)

Спецификатор доступа **protected** можно также использовать со структурами.

Примеры

1. В этой программе проиллюстрирован доступ к открытым, закрытым и защищённым членам класса:

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

class samp {
  // члены класса, закрытые по умолчанию
  int a;
protected: // тоже закрытые члены класса samp
  int b;
public:
  int c;

  samp(int n, int m) { a = n; b = m; }
  int geta() { return a; }
  int getb() { return b; }
};

int main()
{
  samp ob(10, 20);

  // ob.b = 99; Ошибка! Переменная b защищена и поэтому закрыта
  ob.c = 30; // Правильно! Переменная c является открытым членом

  cout << ob.geta() << ' ';
  cout << ob.getb() << ' ' << ob.c << '\n';
```

```
   return 0;
}
```

Как вы могли заметить, выделенная в комментарий строка содержит инструкцию, недопустимую в функции **main()**, поскольку переменная **b** является защищённой и таким образом по-прежнему закрытой для класса **samp**.

2. В следующей программе показано, что происходит, если защищённые члены класса наследуются как открытые:

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

class base {
protected:     // закрытые члены класса base,
   int a,b;   // но для производного класса они доступны
public:
   void setab(int n, int m) { a = n; b = m; }
};

class derived : public base {
   int c;
public:
   void setc(int n) { c = n; }

   // эта функция имеет доступ к переменным a и b класса base
   void showabc() {
      cout << a << ' ' << b << ' ' << c << '\n';
   }
};

int main()
{
   derived ob;

   /* Переменные a и b здесь недоступны, поскольку являются закрытыми членами классов base и derived
*/

   ob.setab(1, 2);
   ob.setc(3);

   ob.showabc();

   return 0;
}
```

Поскольку переменные **a** и **b** в классе **base** защищены и наследуются производным классом **derived** как открытые члены, они доступны для использования функциями – членами класса **derived**. Однако вне двух этих классов они в полной мере мере закрыты и недоступны.

3. Как упоминалось ранее, если базовый класс наследуется как защищённый. Открытые и защищённые члены базового класса становятся защищёнными членами производного класса. Например, в слегка изменённой версии программы из предыдущего примера класс **base** наследуется не как открытый, а как защищённый:

```
// Эта программа компилироваться не будет
#include <iostream>
```

```
using namespace std;

class base {
protected:     // закрытые члены класса base,
   int a,b;   // но для производного класса они доступны
public:
   void setab(int n, int m) { a = n; b = m; }
};

class derived : protected base { // класс base наследуется как защищенный
   int c;
public:
   void setc(int n) { c = n; }

   // эта функция имеет доступ к переменным a и b класса base
   void showabc() {
      cout << a << ' ' << b << ' ' << c << '\n';
   }
};

int main()
{
   derived ob;

   // ОШИБКА: теперь функция setab()
   // является защищенным членом класса base
   ob.setab(1, 2); // функция setab() здесь недоступна

   ob.setc(3);

   ob.showabc();

   return 0;
}
```

Как указано в комментариях, поскольку класс **base** наследуется как защищённый, его открытые и защищённые элементы становятся защищёнными членами производного класса **derived** и следовательно внутри функции **main()** они недоступны.

7.3 Конструкторы, деструкторы и наследование

Базовый класс, производный класс или оба класса вместе могут иметь конструкторы и/или деструкторы. В этой главе исследуется несколько следствий такого положения.

Если у базового и у производного классов имеются конструкторы и деструкторы, то конструкторы выполняются в порядке наследования, а деструкторы – в обратном порядке. Таким образом, конструктор базового класса выполняется раньше конструктора производного класса. Для деструкторов правилен обратный порядок: деструктор производного класса выполняется раньше деструктора базового класса.

Последовательность выполнения конструкторов и деструкторов достаточно очевидна. Поскольку базовый класс «не знает» о существовании производного класса и возможно становится основой для любой инициализации, выполняемой в производном классе. Поэтому иниализация в базовом классе должна выполняться первой.

С другой стороны, деструктор производного класса должен выполняться раньше деструктора базового класса потому, что базовый класс лежит в основе производного. Если бы деструктор базового класса выполнялся первым, это бы разрушило производный класс. Таким образом, деструктор производного класса должен вызываться до того, как объект прекратит своё существование.

Пока что ни в одном из предыдущих примеров мы не передавали аргументы для конструктора производного или базового класса. Однако это вполне возможно. Когда инициализация проводится только в производном классе, аргументы передаются обычным образом. Однако при необходимости передать аргумент конструктору базового класса ситуация несколько усложняется. Во-первых, все необходимые аргументы базового и производного классов передаются конструктору производного класса. Затем, используя расширенную форму объявления конструктора производного класса, соответствующие аргументы передаются дальше в базовый класс. Синтаксис передачи аргументов из производного в базовый класс показан ниже:

конструктор_произв_класса(список-арг): базов_класс(список_арг) {
//тело конструктора производного класса
}

Для базового и производного классов допустимо использовать одни и те же аргументы. Кроме того, для производного класса допустимо игнорирование всех аргументов и передача их напрямую в базовый класс.

Примеры

1. В этой очень короткой программе показано, в каком порядке выполняются конструкторы и деструкторы базового и производного классов:

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

class base {
public:
   base() { cout << "Работа конструктора базового класса\n"; }
   ~base() { cout << "Работа деструктора базового класса\n"; }
};

class derived : public base {
public:
   derived() { cout << "Работа конструктора производного класса\n"; }
   ~derived() { cout << "Работа деструктора производного класса\n"; }
};

int main()
{
   derived o;

   return 0;
}
```

После выполнения программы на экран выводится следующее:

```
Работа конструктора базового класса
Работа конструктора производного класса
```

```
Работа деструктора производного класса
Работа деструктора базового класса
```

Как видите, конструкторы выполняются в порядке наследования, а деструкторы – в обратном порядке.

2. В этой программе показана передача аргумента конструктору производного класса:

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

class base {
public:
   base() { cout << "Работа конструктора базового класса\n"; }
   ~base() { cout << "Работа деструктора базового класса\n"; }
};

class derived : public base {
   int j;
public:
   derived(int n) {
      cout << "Работа конструктора производного класса\n";
      j = n;
   }
   ~derived() { cout << "Работа деструктора производного класса\n"; }
   void showj() { cout << j << '\n'; }
};

int main()
{
   derived o(10);

   o.showj();

   return 0;
}
```

Обратите внимание, что аргумент передаётся конструктору производного класса обычным образом.

3. В следующем примере у конструкторов производного и базового классов имеются аргументы. В этом особом случае оба конструктора используют один и тот же аргумент, и производный класс промто передаёт этот аргумент в базовый класс.

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

class base {
   int i;
public:
   base(int n) {
      cout << "Работа конструктора базового класса\n";
      i = n;
   }
   ~base() { cout << "Работа деструктора базового класса\n"; }
   void showi() { cout << i << '\n'; }
};

class derived : public base {
   int j;
```

```
public:
   derived(int n) : base(n) { // передача аргумента в базовый класс
      cout << "Работа конструктора производного класса\n";
      j = n;
   }
   ~derived() { cout << "Работа деструктора производного класса\n"; }
   void showj() { cout << j << '\n'; }
};

int main()
{
   derived o(10);

   o.showi();
   o.showj();

   return 0;
}
```

Обратите особое внимание на объявление конструктора производного класса. Отметьте, как параметр **n** (который получает аргумент при инициализации) используется в конструкторе **derived()** и передаётся конструктору **base().**

4. Обычно конструкторы базового и производного классов *не* используют один и тот же аргумент. В этом случае, при необходимости передать каждому конструктору класса один или несколько аргументов, вы должны передать конструктору производного класса *все* аргументы, необходимые конструкторам *обоих* классов. Затем конструктор производного класса просто передаёт конструктору базового класса те аргументы, которые ему требуются. Например, в представленной ниже программе показано, как передать один аргумент конструктору производного класса, а другой – конструктору базового класса:

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

class base {
   int i;
public:
   base(int n) {
      cout << "Работа конструктора базового класса\n";
      i = n;
   }
   ~base() { cout << "Работа деструктора базового класса\n"; }
   void showi() { cout << i << '\n'; }
};

class derived : public base {
   int j;
public:
   derived(int n, int m) : base(m) { // передача аргумента
                                     // в базовый класс
      cout << "Работа конструктора производного класса\n";
      j = n;
   }
   ~derived() { cout << "Работа деструктора производного класса\n"; }
   void showj() { cout << j << '\n'; }
};

int main()
{
```

```
   derived o(10, 20);

   o.showi();
   o.showj();

   return 0;
}
```

5. Конструктору производного класса совершенно нет необходимости как-то обрабатывать аргумент. Предназначенный для передачи в базовый класс. Если производному классу этот аргумент не нужен, он его просто игнорирует и передаёт в базовый класс. Например. В это фрагменте параметр **n** конструктором **derived()** не используется. Вместо этого он просто передаётся конструктору **base():**

```
class base {
  int i;
public:
  base(int n)  {
  cout << "Работа конструктора базового класса\n";
   i = n;
  }

  ~base() { cout << "Работа деструктора базового класса \n"; }
  void showi() { cout <<  i << '\n'; }
};

class derived: public base {
   int j;
public:
   derived(int n): base(n) { // передача аргумента в базовый класс
   cout << "Работа конструктора производного класса\n";
   j = 0;  // аргумент n здесь не используется
   }

   ~derived() { cout << "Работа деструктора производного класса \n"; }
   void showj() { cout << j << '\n';}
};
```

7.4 Множественное наследование

Имеются два способа, посредством которых производный класс может наследовать более одного базового класса. Во-первых, производный класс может использоваться в качестве базового для другого производного класса, создавая многоуровневую иерархию классов. В этом случае говорят, что исходный класс является *косвенным (indirect)* базовым классом для второго производного класса. (Отметьте, что любой класс – независимо от того, как он создан – может использоваться в качестве базового класса.) Во-вторых, производный класс может прямо наследовать более одного базового класса. В такой ситуации созданию производного класса помогает комбинация двух или более базовых классов. Ниже исследуются результаты, к которым приводит наследование нескольких базовых классов.

Когда класс используется как базовый для производного, который, в свою очередь, является базовым для другого производного класса, конструкторы всех трёх классов вызываются в порядке наследования. (Это положение является обобщением ранее исследованного принципа). Деструкторы вызываются в обратном порядке. Таким образом, если класс **B1** наследуется классом **D1**, а **D1** – классом **D2**, то конструктор класса **B1**

вызывается первым, за ним конструктор класса **D1**. За которым. В свою очередь, конструктор класса **D2.** Деструкторы вызываются в обратном порядке.

Если производный класс напрямую наследует несколько базовых классов, используется такое расширенное объявление:

class ***имя_производного_класса:*** ***сп_доступа имя_базового_класса1,***
сп_доступа имя_базового_класса2,
…, сп_доступа имя_базового_классаN
{
//… тело класса
}

Здесь ***имя_базового_класса1…имя_базового_классаN*** – имена базовых классов, ***сп_доступа*** – спецификатор доступа, который может быть разным у разных базовых классов. Когда наследуется несколько базовых классов, конструкторы выполняются слева направо в том порядке, который задан в объявлении производного класса. Деструкторы выполняются в обратном порядке.

Когда класс наследует несколько базовых классов, конструкторам которых необходимы аргументы, производный класс передаёт эти аргументы, используя расширенную фоому объявления конструктора производного класса:

констр_произв_класса (список_арг): ***имя_базового_класса1 (список_арг),***
имя_базового_класса2 (список_арг),
…, имя_базового_классаN (список_арг)

{
//…тело конструктора производного класыса
}

Здесь ***имя_базового_класса1…имя_базового_классаN*** - имена базовых классов.

Примеры

1. В этом примере производный класс наследует класс, производный от другого класса. Обратите внимание, как аргументы передаются по цепочке от класса **D2** к классу **B1.**

```
// Множественное наследование
#include <iostream>
using namespace std;

class B1 {
   int a;
public:
   B1(int x) { a = x; }
   int geta() { return a; }
};

// Прямое наследование базового класса
class D1 : public B1 {
   int b;
```

```
public:
   D1(int x, int y) : B1(y) // передача переменной y классу B1
   {
      b = x;
   }
   int getb() { return b; }
};

// Прямое наследование производного класса
// и косвенное наследование базового класса
class D2 : public D1 {
   int c;
public:
   D2(int x, int y, int z) : D1(y, z) // передача аргументов
                             // классу D1

   {
      c = x;
   }

/* Поскольку базовые классы наследуются как открытые, класс D2 имеет доступ к открытым элементам
классов B1 и D1 */
   void show() {
      cout << geta() << ' ' << getb() << ' ';
      cout << c << '\n';
   }
};

int main()
{
   D2 ob(1, 2, 3);

   ob.show();
   // функции geta() и getb() здесь тоже открыты
   cout << ob.geta() << ' ' << ob.getb() << ' ';

   return 0;
}
```

Вызов функции **ob.show()** выводит на экран значения **3 2 1.** В этом примере класс **B1** является косвенным базовым классом для класса **D2**. Отметьте, что класс **D2** имеет доступ к открытым членам классов **D1** и **B1.** Как вы уже должны знать, при наследовании открытых членов базового класса они становятся открытыми членами производного класса. Поэтому, если класс **D1** наследует класс **B1**, то функция **geta()** становится открытым членом класса **D1** и затем открытым членом класса **D2.**

Как показано в программе, каждый класс иерархии классов должен передавать все аргументы, необходимые каждому предшествующему базовому классу. Невыполнение этого правила приведёт к ошибке компиляции программы.

Здесь показана иерархия классов предыдущей программы:

Перед тем как двигаться дальше, необходимо небольшое замечание о стиле изображения графов наследования в C++. Обратите внимание на то, что в предыдущем графе стрелки направлены не вниз, а вверх.

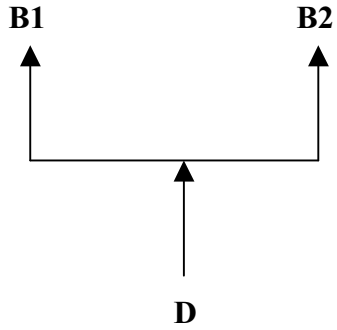
Традиционно программисты C++ изображают отношения наследования в виде прямых графов, стрелки которых направлены от производного к базовому классу. Хотя новички могут посчитать такой подход излишне схематичным, именно он обычно практикуется в C++.

2. Здесь представлена переработанная версия предыдущей программы, в которой производный класс прямо наследует два базовых класса:

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

// Создание первого базового класса
class B1 {
   int a;
public:
   B1(int x) { a = x; }
   int geta() { return a; }
};

// Создание второго базового класса
class B2 {
   int b;
public:
   B2(int x)
   {
      b = x;
   }
   int getb() { return b; }
};

// Прямое наследование двух базовых классов
class D : public B1, public B2 {
   int c;
public:
   // здесь переменные z и y
   // напрямую передаются классам B1 и B2
   D(int x, int y, int z) : B1(z), B2(y)
   {
      c = x;
   }

/* Поскольку базовые классы наследуются как открытые, класс D имеет доступ к открытым элементам классов B1 и B2
*/
```

```
  void show() {
    cout << geta() << ' ' << getb() << ' ';
    cout << c << '\n';
  }
};

int main()
{
  D ob(1, 2, 3);

  ob.show();

  return 0;
}
```

В этой версии программы класс **D** передаёт аргументы по отдельности классам **B1** и **B2.** Теперь иерархия классов выглядит таким образом:



3. В следующей программе показан порядок, в котором вызываются конструкторы и деструкторы, когда производный класс прямо наследует несколько базовых классов:ы

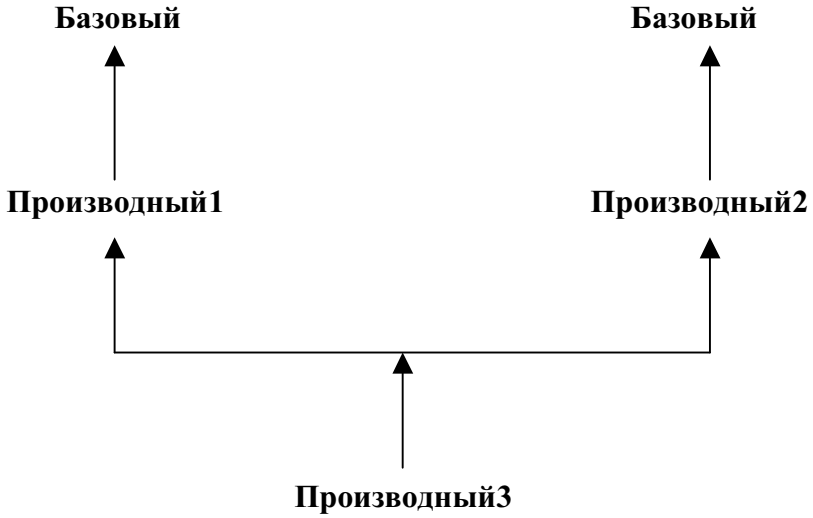
```
#include <iostream>
using namespace std;

class B1 {
public:
  B1() { cout << "Работа конструктора класса B1\n"; }
  ~B1() { cout << "Работа деструктора класса B1\n"; }
};

class B2 {
  int b;
public:
  B2() { cout << "Работа конструктора класса B2\n"; }
  ~B2() { cout << "Работа деструктора класса B2\n"; }
};

// Наследование двух базовых классов
class D : public B1, public B2 {
public:
  D() { cout << "Работа конструктора класса D\n"; }
  ~D() { cout << "Работа деструктора класса D\n"; }
};

int main()
{
  D ob;

  return 0;
}
```

Эта программа выводит на экран следующее:

```
Работа конструктора класса B1
Работа конструктора класса B2
Работа конструктора класса D
Работа деструктора класса D
Работа деструктора класса B2
Работа деструктора класса B1
```

7.5 Виртуальные базовые классы

При многократном прямом наследовании производным классом одного и того же базового класса может возникнуть проблема. Чтобы понять, что это за проблема, рассмотрим следующую иерархию классов:



Здесь базовый класс ***Базовый*** наследуется производными классами ***Производный1*** ***Производный2.*** Производный класс ***Производный3*** прямо наследует производные классы ***Производный1*** и ***Производный2.*** Однако это подразумевает, что класс ***Базовый*** фактически наследуется классом ***Производный3*** дважды – первый раз через класс ***Производный1***, а второй через класс ***Производный2***. Однако, если член класса ***Базовый*** будет использоваться в классе ***Производный3*** имеется две копии класса ***Базовый***, то будет ли ссылка на элемент класса ***Базовый*** относиться к классу ***Базовый***, наследуемому через класс ***Производный2***? Для преодоления этой неоднозначности в C++ включён механизм, благодаря которому в классе ***Производный3*** будет включена только одна копия класса ***Базовый***. Класс, поддерживающий этот механизм, называется *виртуальным базовым классом (virtual base class).*

В таких ситуациях, когда производный класс более одного раза косвенно наследует один и тот же базовый класс, появление двух копий базового класса в объекте производного класса можно предотвратить, если базовый класс наследуется как виртуальный для всех производных классов. Такое наследование не даёт появиться двум (или более) копиям базового класса в любом следующем производном классе, косвенно наследующем базовый класс. В этом случае перед спецификатором доступа базового класса необходимо поставить ключевое слово **virtual.**

Примеры

1. В этом примере для предотвращения появления в классе **derived3** двух копий класса **base** используется виртуальный базовый класс.

```
// В этой программе используется виртуальный базовый класс
#include <iostream>
using namespace std;

class base {
public:
   int i;
};

// Наследование класса base как виртуального
class derived1 : virtual public base {
public:
   int j;
};

// Здесь класс base тоже наследуется как виртуальный
class derived2 : virtual public base {
public:
   int k;
};

/* Здесь класс derived3 наследует как класс derived1, так и класс derived2. Однако в классе derived3 создается только одна копия класса base
*/
class derived3 : public derived1, public derived2 {
public:
   int product() { return i * j * k; }
};

int main()
{
   derived3 ob;
   // Здесь нет неоднозначности, поскольку
   // представлена только одна копия класса base
   ob.i = 10;
   ob.j = 3;
   ob.k = 5;

   cout << "Результат равен " << ob.product() << '\n';

   return 0;
}
```

Если бы классы **derived1** и **derived2** наследовали класс **base** не как виртуальный, тогда инструкция

ob.i = 10;

вызвала бы неоднозначность и при компиляции возникла бы ошибка.

2. Важно понимать, что даже если базовый класс наследуется производным как виртуальный, то копия этого базового класса всё равно существует внутри

производного. Например, по отношению к предыдущей программе этот фрагмент совершенно правилен:

```
derived1.ob;
ob.i  = 100;
```

Отличие между обычным и виртуальным базовыми классами проявляется только тогда. Когда объект наследует базовый класс более одного раза. Если используются виртуальные базовые классы, то в каждом конкретном случае (при обычном наследовании) там было бы несколько копий.ы