Простой полиморфизм

В языке C++ предусмотрен м е х а н и з м п о л и м о р ф и з м а, обеспечивающий возможность определения разных описаний некоторого единого по названию метода для классов различных уровней иерархии. При этом различают *простой полиморфизм,* базирующийся на механизме раннего связывания, и *сложный полиморфизм,* использующий механизм позднего связывания.

Простой (можно использовать термин "статический") полиморфизм поддерживается языком C++ на этапе компиляции и реализуется с помощью *механизма переопределения (перегрузки) функций. Поэтому такие полиморфные функции называются в C++* переопределяемыми. В соответствии с общими правилами переопределения функций они должны отличаться *сигнатурой, т. е. количеством, типом и порядком следования* передаваемых параметров.

Пример. Использование раннего связывания. Пусть функции выполняют одинаковые по смыслу операции – выводят главные поля объекта на экран, следовательно, их можно назвать одним именем, например print().
Метод print(), таким образом, станет полиморфным – переопределяемым в производном классе. Отдельное определение метода в своем классе называют *аспектом* полиморфного

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <string.h>
using namespace std;
typedef unsigned long dlong;
class Tlong // Класс Целое число
{
public:
dlong num; // числовое поле класса
Tlong(){} // неинициализирующий конструктор
Tlong(dlong an):num(an){} // конструктор
~Tlong(){} // деструктор
void print(void) // вывод значения поля
{ cout<<" Целое число : "<<num<<endl; }
void setnum(dlong an) // инициализации поля
{ num=an; }
};
class Treal: public Tlong // Класс Вещественное
{
public:
    dlong drob; // дробная часть числа
char *real; // запись вещественного числа
Treal(){} // неинициализирующий конструктор
Treal(char *st) :Tlong() // конструктор
{ setnumv(st); }
~Treal() // деструктор
{ delete real; }
void print(); // вывод вещественного числа (переопределяется)
void setnumv(char * st); // инициализация полей класса
};
```

```
void Treal::setnumv(char * st)
{
    int l; char *ptr;
l=strlen(st); real=new char[l+1]; strcpy(real,st);
ptr=strchr(real,'.'); *ptr='\0';
drob=dlong(atol(ptr+1));
num=dlong(atol(real));
*ptr='.';
}
void Treal::print()
{
cout<<"Вещественное число: "<<real<<endl;
cout<<"Целая часть: "; Tlong::print();
cout<<"Дробная часть: "<<drob<<endl;
}
int main ()
{
setlocale(0,"russian");
Tlong i(174832); // простой объект базового класса
i.print(); // явный вызов переопределяемого метода
Treal a("1748.5932"); // простой объект производного класса
a.print(); // явный вызов переопределенного метода
}
```

```
 Целое число : 174832
Вещественное число: 1748.5932
Целая часть:  Целое число : 1748
Дробная часть: 5932

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Сложный полиморфизм

Однако использование переопределенных методов не всегда безопасно. Известны случаи, когда при переопределении методов возникают ошибки, связанные с некорректным определением типа объекта на этапе компиляции, а следовательно и требуемого аспекта вызываемого метода.

Пример.
Для демострации ошибок, возникающих при некорректном использовании простого полиморфизма введем в описание базового класса предыдущего примера новый метод show(). Этот метод будет вызывать статический полиморфный метод print() и наследоваться в производных классах. Кроме этого добавим в программу внешнюю функцию show_ext() с параметром – ссылкой на базовый класс, чтобы показать особенности раннего связывания.

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <string.h>
using namespace std;
typedef unsigned long dlong;
class Tlong // Класс Целое число
{
public:
dlong num; // числовое поле класса
Tlong(){} // неинициализирующий конструктор
Tlong(dlong an):num(an){} // конструктор
~Tlong(){} // деструктор
void print(void) // вывод значения поля
{ cout<<" Целое число : "<<num<<endl; }
void setnum(dlong an) // инициализации поля
{ num=an; }
void show() // метод, вызывающий переопределяемый метод
{ print(); }
};
class Treal: public Tlong // Класс Вещественное число
{
public:
dlong drob; // дробная часть числа
char *real; // запись вещественного числа
Treal(){} // неинициализирующий конструктор
Treal(char *st) :Tlong() // конструктор
{ setnumv(st); }
~Treal() // деструктор
{ delete real; }
void print(); // вывод вещественного числа (переопределяется)
void setnumv(char * st); // инициализация полей класса
};
```

```
35  void Treal::setnumv(char * st)
36  {
37      int l; char *ptr;
38  l=strlen(st); real=new char[l+1]; strcpy(real,st);
39  ptr=strchr(real,'.');
40  drob=dlong(atol(ptr+1)); *ptr='\0';
41  num=dlong(atol(real));
42  *ptr='.';
43  }
44  void Treal::print()
45  {
46  cout<<"Вещественное число: "<<real<<endl;
47  cout<<"Целая часть: "; Tlong::print();
48  cout<<"Дробная часть: "<<drok<<endl;
49  }
50  // Внешняя функция с параметром - ссылкой на базовый класс Tlong
51  void show_ext(Tlong &par)
52  { par.print(); }
53  int main ()
54  {
55  setlocale(0,"russian");
56  Tlong i(174832); // простой объект базового класса
57  i.show(); // косвенный вызов переопределяемого метода класса
58  Treal a("1748.5932"); // простой объект производного класса
59  a.show(); // выводит только целую часть числа (ошибка!)
60  Treal *pa=new Treal("456789.1234321"); /* указатель на объект
61  производного класса */
62  pa->print(); // явный вызов переопределенного метода
63  pa->show(); // выводит только целую часть числа (ошибка!)
64  delete pa; // вызывает деструктор производного класса
65  Tlong *pb=new Treal("234567.34765");/* указатель
66  базового класса, объект производного класса */
67  pb->print(); // выводит только целую часть числа (ошибка!)
68  delete pb; // неявно вызывается деструктор класса Tlong (ошибка!)
69  show_ext(a); // выводит только целую часть числа (ошибка!)
70  }
```

```
 Целое число : 174832
 Целое число : 1748
Вещественное число: 456789.1234321
Целая часть:  Целое число : 456789
Дробная часть: 1234321
 Целое число : 456789
 Целое число : 234567
 Целое число : 1748

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Виртуальные функции – определение

Виртуальная функция задается точно также как и обычная, только в начале определения такой функции ставится ключевое слово ***virtual***.

Виртуальная функция объявляется внутри базового класса.

Виртуальная функция не может быть статической.

Если виртуальная функция переопределяется в производных классах, то она автоматически в них становится виртуальной и в этом случае нет необходимости в использовании ключевого слова ***virtual***.

Деструкторы могут быть виртуальными, а конструкторы – нет.

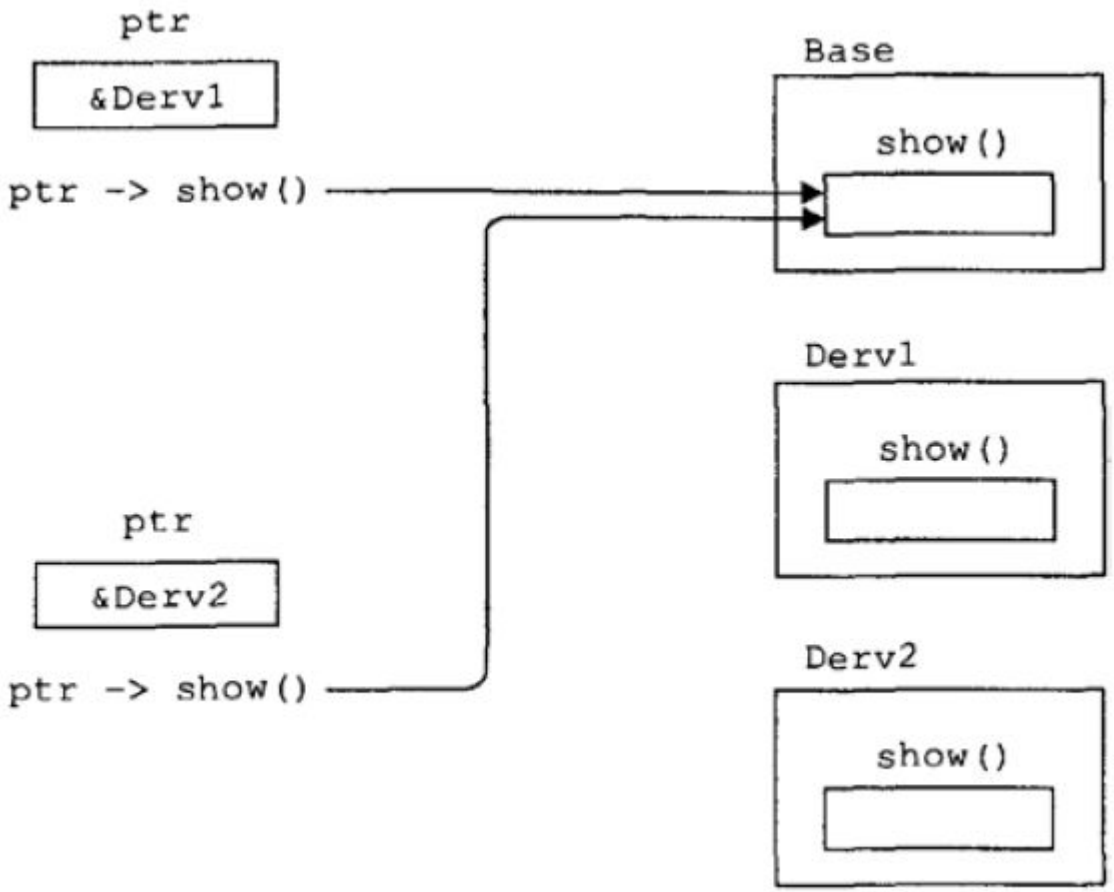
Виртуальная функция может быть вызвана как обычная.

Доступ к обычным методам через указатели

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////
class Base                  //Базовый класс
      {
      public:
   void  show()             //Обычная функция
      {  cout << "Base\n"; }
      };
////////////////////////////////////////////////
class Derv1 : public Base //Производный класс 1
      {
      public:
   void  show()
      {  cout << "Derv1\n"; }
      };
////////////////////////////////////////////////
class Derv2 : public Base //Производный класс 2
      {
      public:
   void  show()
      {  cout << "Derv2\n"; }
      };
////////////////////////////////////////////////
int main()
      {
      Derv1 dv1;            //Объект производного класса 1
      Derv2 dv2;            //Объект производного класса 2
      Base* ptr;            //Указатель на базовый класс

      ptr = &dv1;           //Адрес dv1 занести в указатель
      ptr->show();          //Выполнить show()
      ptr = &dv2;           //Адрес dv2 занести в указатель
      ptr->show() ;          //Выполнить show()
      return 0;
      }
```

```
Base
Base

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Доступ через указатель без использования виртуальных функций

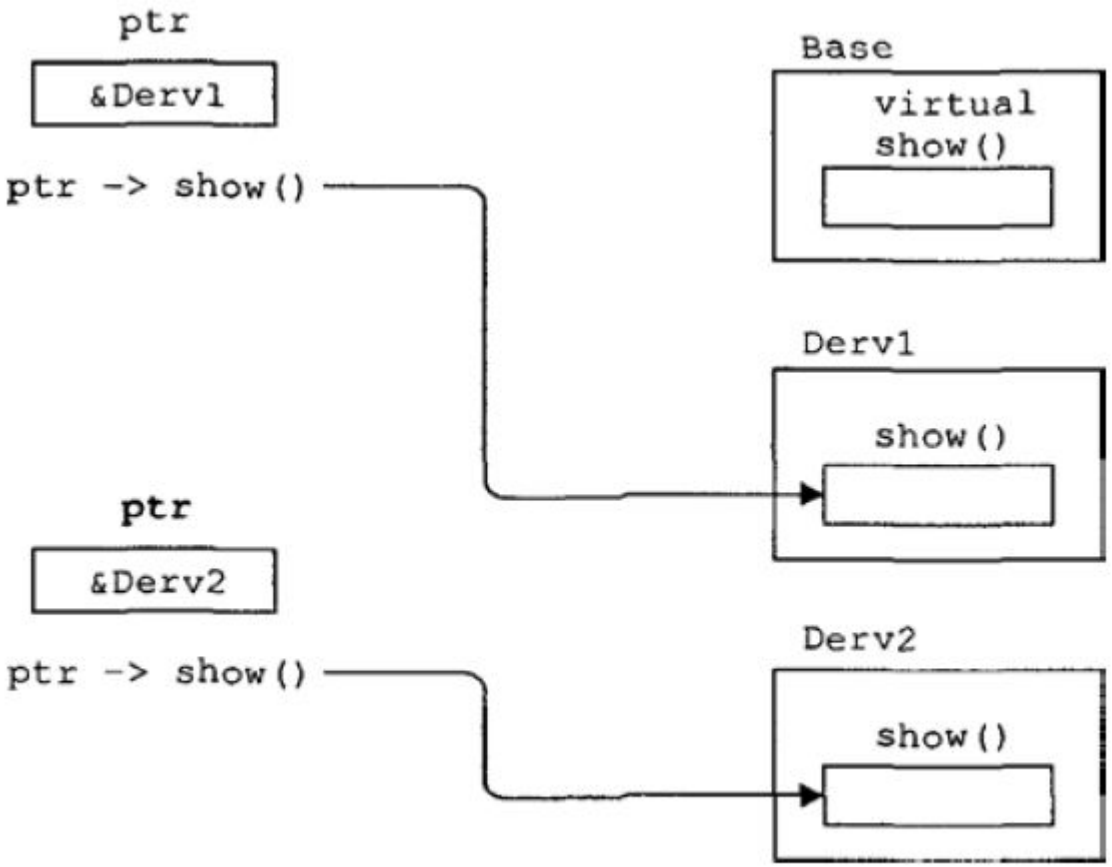


Доступ к виртуальным методам через указатели

```
// Доступ к виртуальным функциям через указатели
#include <iostream>
using namespace std;
////////////////////////////////////////////////
class Base                    //Базовый класс
    {
    public:
  virtual void  show()      //Виртуальная функция
      {  cout << "Base\n"; }
    };
////////////////////////////////////////////////
class Derv1 : public Base //Производный класс 1
    {
    public:
  void  show()
      {  cout << "Derv1\n"; }
    };
////////////////////////////////////////////////
class Derv2 : public Base //Производный класс 2
    {
    public:
  void  show()
      {  cout << "Derv2\n"; }
    };
////////////////////////////////////////////////
int main()
    {
    Derv1 dv1;              //Объект производного класса 1
    Derv2 dv2;              //Объект производного класса 2
    Base* ptr;              //Указатель на базовый класс

    ptr = &dv1;             //Адрес dv1 занести в указатель
    ptr->show();             //Выполнить show()
    ptr = &dv2;             //Адрес dv2 занести в указатель
    ptr->show();            //Выполнить show()
    return 0;
    }
```

```
Derv1
Derv2

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Доступ через указатель к виртуальным функциям



Виртуальные функции и полиморфизм

С понятием виртуальных функций тесно связано такое понятие как полиморфизм. На рассмотренном примере это будет выглядеть следующим образом: если используется указатель на базовый класс, в котором определена виртуальная функция, и эта функция переопределена в производных классах, то при адресации указателя базового класса на экземпляры производных, будет вызываться функция, соответствующая каждому производному классу.

Виртуальная и перегруженная функция – в чем отличие? Виртуальная функция должна полностью повторяться в производных классах, т.е. имя функции, список аргументов и возвращаемое значение обязательно должны совпадать, иначе такая функция будет считаться просто перегруженной функцией и не будет являться виртуальной.

Виртуальные функции - пример

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

class B {
public:
   virtual void f(int) { cout << " fB(int)\n"; }
   virtual void f(double) { cout << " fB(dbl)\n"; }
};

class D : public B {
public:
   virtual void f(int) { cout << " fD(int)\n"; }
};
```

```
int main()
{
  D d;
  B b, *pb = &d;
  b.f(6);
  d.f(6);
  b.f(6.2);
  d.f(6.2);
  pb->f(6);
  pb->f(6.2);
}
```

Результат выполнения программы:

```
 fB(int)
 fD(int)
 fB(dbl)
 fD(int)
 fD(int)
 fB(dbl)
```

Виртуальные функции – пример 2

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

class A {
public:
 virtual void print() { cout << " Base A\n”; }
};

class D : public A {
public:
   void print() { cout << " Derived B \n”; }
};

class D1 : public A {
public:
   void fun() { cout << " Nothing\n”; }
};
```

```
int main()
{
  A obj1;
  D obj2;
  D1 obj3;
  A *ptr;
  ptr = &obj1;   ptr->print();
  ptr = &obj2;   ptr->print();
  p = &obj3;   p->print();   p->fun();

}
```

Где ошибка?

Виртуальные функции – пример 2’

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

class A {
public:
 virtual void print(){cout << " Base A\n”; }
};

class D : public A {
public:
   void print(){cout << " DerivedB \n”; }
};

class D1 : public A {
public:
   void fun(){cout << " Nothing\n”; }
};
```

```
void main()
{
  A obj1; D obj2; D1 obj3;   A *ptr;  D1 *p;
  ptr = &obj1;   ptr->print();
  ptr = &obj2;   ptr->print();
  p = &obj3;   p->print();   p->fun();
}
```

Результат выполнения программы:

```
Base A
Derived B
Base A
Nothing
```

Если виртуальная функция не определена в производном классе, однако осуществляется ее вызов, то вызовется виртуальная функция из базового класса.

Чисто виртуальные функции

Часто возникают ситуации, при которых виртуальные функции, определенные в базовых классах не используются, а иногда и не содержат никаких действий, а являются лишь шаблонами для конкретных реализаций виртуальных функций в производных классах.

Для того, чтобы подчеркнуть, что в программе не предусматривается вызов виртуальной функции для базового класса, используются чисто виртуальные функции.

Для их определения используют следующую запись:

virtual возвращаемый_тип имя_функции (аргументы) =0;

Данная запись при компиляции воспринимается как отсутствие определения виртуальной функции для базового класса.

Вызов такой функции будет восприниматься как ошибка.

Если в базовом классе определен прототип чисто виртуальной функции, то она должна быть обязательно определена для всех производных классов.

Если в базовом классе определена хотя бы одна чисто виртуальная функция, то такой класс называется абстрактным базовым классом. Для такого класса нельзя создать экземпляр.

Абстрактные классы и чистые виртуальные функции

```cpp
// Чистая виртуальная функция
#include <iostream>
using namespace std;
class Base                      //базовый класс
  {
  public:
    virtual void show() = 0; //чистая виртуальная
                             //функция
  };
////////////////////////////////////////////////////
class Derv1 : public Base       //порожденный класс 1
  {
  public:
    void show()
      { cout << "Derv1\n"; }
  };
////////////////////////////////////////////////////
class Derv2 : public Base       //порожденный класс 2
  {
  public:
    void show()
      { cout << "Derv2\n"; }
  };
////////////////////////////////////////////////////
```

```cpp
int main()
  {
//  Base bad;                   //невозможно создать объект
                                //из абстрактного класса
  Base* arr[2];                 //массив указателей на
                                //базовый класс
  Derv1 dv1;                    //Объект производного класса 1
  Derv2 dv2;                    //Объект производного класса 2

  arr[0] = &dv1;                //Занести адрес dv1 в массив
  arr[1] = &dv2;                //Занести адрес dv2 в массив

  arr[0]->show();               //Выполнить функцию show()
  arr[1]->show();               //над обоими объектами
  return 0;
  }
```

```
Derv1
Derv2

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Раннее и позднее связывание

Особый интерес к использованию виртуальных функций появляется при обслуживании случайных событий. Яркий пример – ожидание ввода с клавиатуры. В зависимости от того, каков будет ввод данных, к примеру, должны будут вызываться виртуальные функции из разных производных классов с использованием указателя на базовый.

Под процессами раннего связывания понимают те процессы, которые могут быть предопределены на этапе компиляции. Пример – при использовании вызова обычных функций. Компилятор заранее знает адрес вызываемой функции и помещает его в место вызова этой функции.

Позднее связывание реализуется при вызовах виртуальных функций. Процессы, относящиеся к позднему связыванию, определяются только на стадии выполнения программы.

Так, например, компилятор заранее может не предугадать вызов виртуальной функции для конкретных экземпляров производных классов.

Адрес виртуальной функции вычисляется только при работе программы.

Виртуальные деструкторы

Виртуальные деструкторы необходимы в случаях использования указателей на базовые классы при выделении динамической памяти под объекты производных классов

```
#include <iostream>
using namespace std;
class A {
public:
  A(char* t)
  {cout << " Base A " <<t << endl; }
  virtual void print()=0;
  virtual ~A() {cout <<" Base destructor\n"; }
};

class Dn : public A {
  int n;
public:
  Dn(int n, char *id="number") : A(id)
   {this->n = n;}
   void print(){cout <<" n=" << n << endl; }
   ~Dn () { cout << "Dn destruction\n"; }
};
class Ds : public A {
  char *str;
public:
  Ds(char *str, char *id="string") : A(id)
  {this->str = str;}
  void print(){cout << " str=" << str << endl; }
  ~Ds () { cout << "Ds destruction\n"; }
};
```

```
void f()
{
  A  *ptr1, *ptr2;
  ptr1 = new Dn(5);
  ptr2 = new Ds("hello");
  ptr1->print();
  ptr2->print();
  delete ptr1;
  delete ptr2;
}
int main() {
  f();
}
```

```
 Base A number
 Base A string
 n=5
 str=hello
Dn destruction
 Base destructor
Ds destruction
 Base destructor

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Виртуальные деструкторы

В случае **не** виртуальных деструкторов при удалении объектов вызывался бы деструктор только базового класса!

Если деструктор базового класса объявлен как виртуальный, все деструкторы производных классов тоже будут виртуальными.

Как и для виртуальных функций, если в производном классе не описан деструктор, то при удалении объекта будет вызван деструктор базового класса. Если деструктор описан, то сперва произойдет вызов деструктора производного, а затем базового класса.

```
//Тест невиртуальных и виртуальных деструкторов
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Base
   {
   public:
      ~Base()                  //невиртуальный деструктор
//    virtual ~Base()          //виртуальный деструктор
         { cout << "Base delete\n"; }
   };
////////////////////////////////////////////////////////////
class Derv : public Base
   {
   public:
      ~Derv()
         { cout << "Derv delete\n"; }
   };
////////////////////////////////////////////////////////////
int main()
   {
   Base* pBase = new Derv;
   delete pBase;
   return 0;
   }
```

```
Base delete

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

```cpp
//Тест невиртуальных и виртуальных деструкторов
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base
   {
   public:
//      ~Base()                  //невиртуальный деструктор
    virtual ~Base()           //виртуальный деструктор
         { cout << "Base delete\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv : public Base
  {
   public:
      ~Derv()
         { cout << "Derv delete\n"; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Base* pBase = new Derv;
   delete pBase;
   return 0;
   }
```

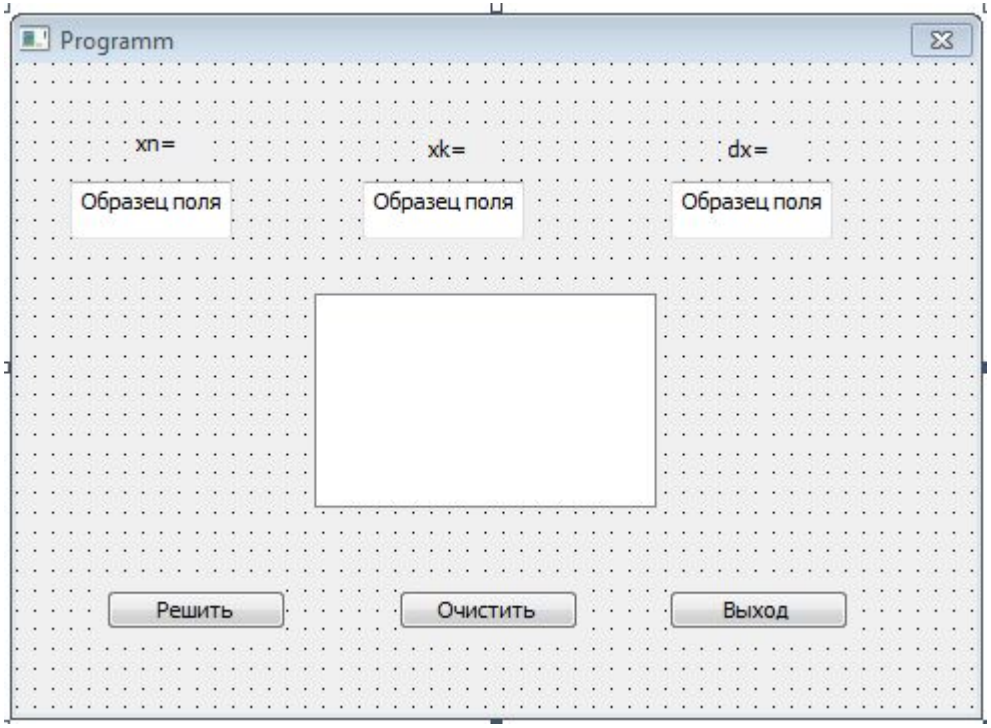
```
Derv delete
Base delete

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Visual C++. MFC. Программирование циклических процессов

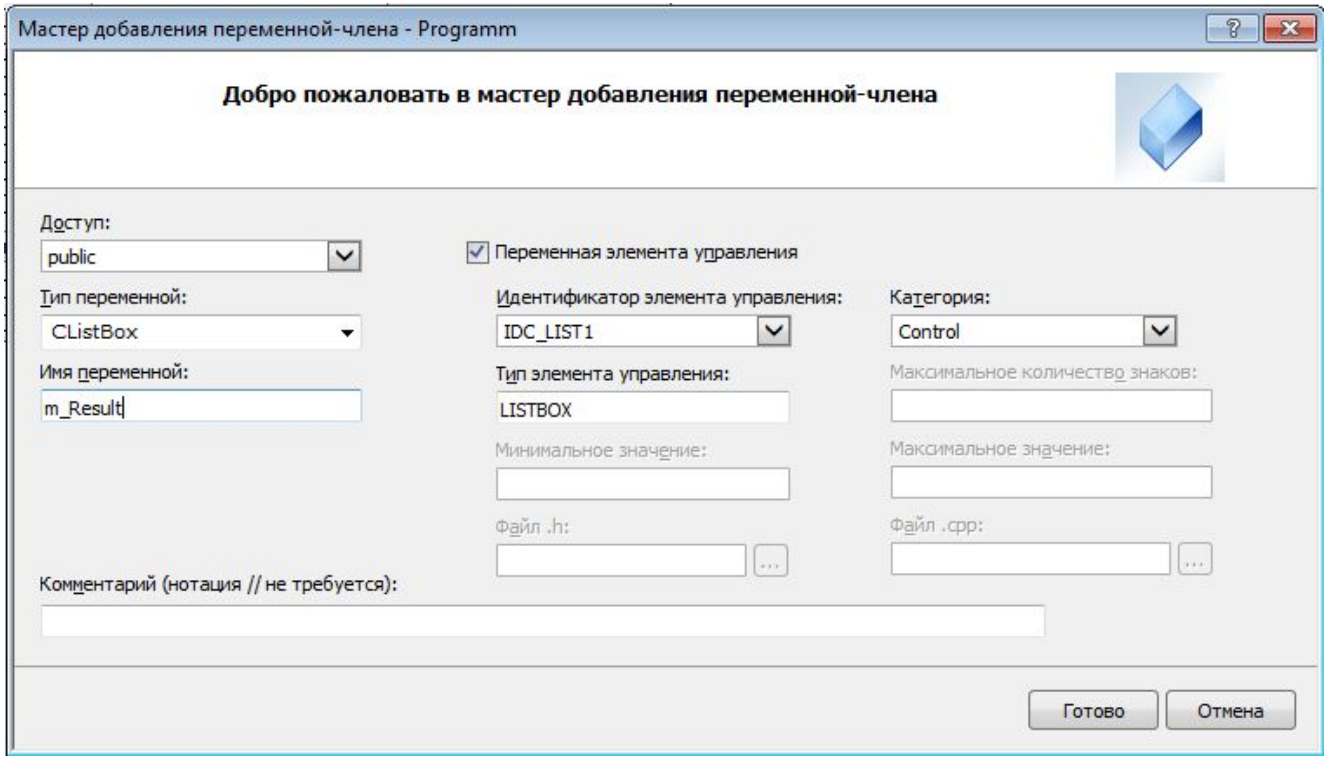
Переменная x меняется от xn до xk с шагом dx. Вывести таблицу значений x и $y=e^{\sin(x)}$, вычислить сумму и произведение y.

Создадим диалоговое окно (мое имеет название ListBProg) и разместим на нем следующие компоненты:

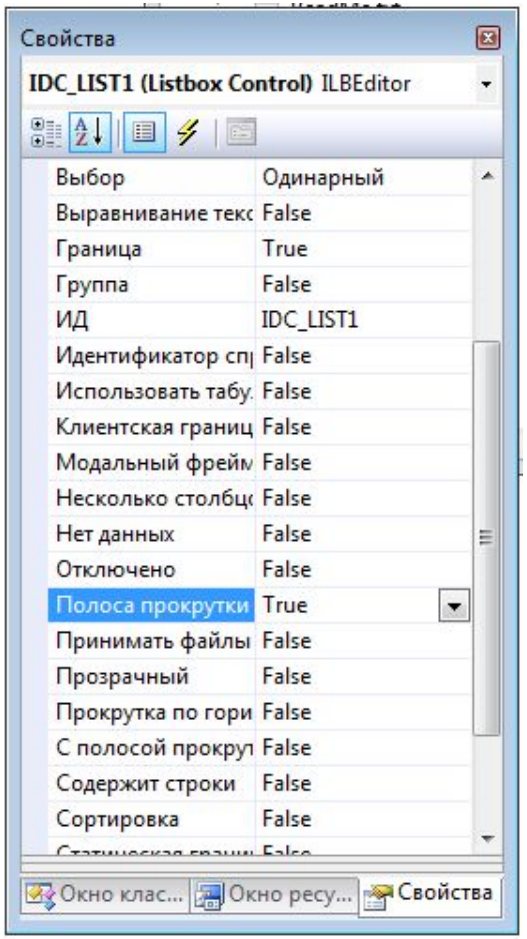
- 3 метки (*Static Text);*
- 3 поля ввода (*Edit Control);*
- 1 список (*List Box);*
- 3 кнопки (*Button).*



Добавим для полей ввода (*Edit Control) переменные m_edit_xn, m_edit_xk,m_edit_dx, которые будут возвращать значение float. В функцию кнопки «Выход» впишем строку OnOK() для того чтобы программа завершала свою работу по нажатию соответствующей клавиши.* Изменим ID кнопок «Решить» и «Очистить» на ID_Solve и ID_Clear, а также добавим переменную m_Result для компонента List Box



Установите в свойствах List Box значение «Сортировка» в false, так как иначе содержимое будет сортироваться по алфавиту, а «С полосой прокрутки» в true.



Зададим начальные значения полей ввода. Для этого откроем файл *Dlg.cpp и в функцию OnInitDialog перед оператором return впишем

```
    // TODO: добавьте дополнительную инициализацию
    m_edit_xn=-5;
    m_edit_xk=4;
    m_edit_dx=3;
    UpdateData(FALSE);
    return TRUE;  // возврат значения TRUE, если фокус не передан элементу управления
}
```

Щелкаем по кнопке «Решить» и вводим следующий текст.

```
void CProgrammDlg::OnBnClickedSolve()
{
    // TODO: добавьте свой код обработчика уведомлений
    int i; float x, xn, xk, dx, y, s, p;
CString S;
UpdateData(TRUE);
xn=m_edit_xn;
xk=m_edit_xk;
dx=m_edit_dx;
for (s=0, p=1, x=xn, i=0; x<=xk; x+=dx, i++)
{
y=exp(sin(x)); s+=y; p*=y;
S.Format (_T("x=%g y=%g"),x,y);
m_Result.AddString(S);
S.Format(_T("s=%g p=%g"), s, p);
m_Result.InsertString(0,S);
}
}
```

А для кнопки «Очистить» определите такой код:

```
void CProgrammDlg::OnBnClickedClear()
{
    // TODO: добавьте свой код обработчика уведомлений
    int j, n;
    n=m_Result.GetCount();
    for (j=0; j<n; j++)
    m_Result.DeleteString(0);
}
```