Элементы ООП в C++

Мех-мат, 2009/2010 уч. год

Введение в ООП

ООП – это один из подходов к разработке и созданию программ.

Исторически:

1. процедурное структурирование программ, программирование «снизу-вверх»…;
2. структурное программирование, программирование «сверху – вниз»…;
3. ООП - Object Oriented Programming….

Структурное программирование – это структурирование на уровне действий, а ООП – это структурирование на уровне моделируемых объектов.

- ООП – это результат 30 летнего опыта и практики программирования, начиная с Simula 67, затем Smalltalk, Lisp, Clu, и далее Actor, Eiffel, Objective C, Object Pascal, Java, C++, C# и т.д.

Концепции, заложенные в ООП, это:

- Моделирование объектов и действий реального мира
- Наличие типов данных, определяемых пользователем
- Сокрытие деталей реализации (инкапсуляция)
- Возможность многократного использования программного кода, благодаря наследованию
- Интерпретация вызовов функций на этапе выполнения программы – полиморфизм.

Введение в ООП

- ООП сегодня используется совместно со структурным программированием и в последние десятилетия большое влияние на технологии программирования оказывает графический интерфейс пользователя **(GUI).**
- Универсальные среды разработки Windows-приложений объединяют в себе визуальное программирование, программирование с управлением по событиям **(event driven)** и программирование с управлением объектами **(object driven)….**
- Программирование с управлением по событиям заключается в программировании ответных действий на реально возникающие события в реальное время – время сеанса работы на компьютере…
- Программирование по событиям дополнило структурное программирование средствами, которые позволили разделить интерфейс пользователя и специфическую обработку данных.
- При моделировании объектов реального мира существенные преимущества имеет ООП, создание графической оболочки Windows ускорило переход к технологии ООП, так как совместное использование GUI и ООП повышает эффективность разработки приложений под Windows.

Введение в ООП

- Визуальное программирование дает возможность изображать объекты на экране до выполнения самой программы, и работать с объектами, держа их все время перед глазами, т.е. в процессе разработки программы программист имеет возможность видеть, как будет выглядеть готовая программа и при необходимости редактировать ее внешний вид.
- Таким образом, объекты живут и действуют под воздействием событий, следовательно, события управляют работой программы. События же возникают, генерируются в результате действий пользователя или системы. Реакцией на событие является выполнение некоторой программы – метода объекта, называемого обработчиком события.

- **Основная идея ООП** - это объединение (**инкапсуляция**) данных и методов их обработки в единое целое – **объект**, который может использоваться как самостоятельная программная единица или как часть другого объекта, или является базой для создания новых объектов…
- В объекте устанавливается связь между данными и действиями над ними, эта связь в ООП имеет большое смысловое значение, определяемое классом решаемых задач. **Объект – это экземпляр класса**, то есть переменная определяемого пользователем типа.

ООП в С++

Определение класса в С++ выглядит точно так же, как определение структуры, только вместо слова struct используется слово class:

```
class <имя_класса> {
/*поля данные */
/* методы */
};
```

Поля класса могут иметь любой тип, кроме типа этого же класса, но могут быть указателями и ссылками на этот класс.

Поля могут быть описаны с модификатором const, но инициализируются такие поля один раз и не должны изменяться.

Имея описание, можно символическое имя - <имя_класса> использовать для описания переменных.

Такая переменная называется объектом, или экземпляром данного класса.

Используя идентификатор типа <имя_класса>, как любой другой, созданный пользователем или стандартный, можно создавать произвольное количество простых переменных (объектов, экземпляров этого класса), массивов объектов данного класса, или указателей на объекты данного класса.

Примеры описания классов и объектов

```
class Tmoney
    { /*……*/ };
TMoney t;     /* t - простая переменная, объект класса  TMoney */
TMoney *p;        /*  p – указатель на объект класса  TMoney */
TMoney m[50];   /* m - массив объектов класса  TMoney */
```

Эти переменные так же, как и переменные встроенных типов, подчиняются правилам видимости, и время их жизни зависит от места объявления.

Их можно использовать в качестве полей структур и других классов, передавать в качестве параметров, они могут быть выходными параметрами функций.

Объявлять переменные можно и с использованием ключевого слова class (по аналогии со структурами):

```
class TMoney t;       /* простая переменная */
class TMoney *p;          /* указатель */
class TMoney m[50];           /* массив */,
```

но чаще экземпляры класса описывают без использования ключевого слова class.

Объявления и определения

Так же, как и при описании других типов, можно одновременно при описании идентификатора типа описать и величины этого типа, например:

class Tclass1 { /*поля и методы*/} p1,p2;

Переменные p1 и p2 – это простые переменные типа Tclass1.

Однако более предпочтительным считается описание:

Tclass1 p3, p4, *s, p5[20];

Заголовок описания класса, заканчивающийся точкой с запятой,

class <имя_класса>;

называют ***объявлением*** класса**.**

- Класс с идентификатором типа <имя_класса> еще не определен, еще не определены его поля и методы, поэтому объявлять переменные такого класса еще нельзя, но указатели и ссылки можно. Такое объявление используется в том случае, когда один класс зависит от второго, а определение этого второго класса недоступно….

Объявления и определения

- Определение класса и структуры внешне отличаются только ключевым словом в заголовке. Но главное отличие заключается в том, что все, что описано внутри класса, недоступно извне класса по умолчанию. Например, имея определение класса person_1 и структуры person_2:

```
class person_1          struct person_2
 {string fio;          { string fio;
  duble summa;               double summa;
    }                   }
```

- Можно определить переменную Ivanov структурного типа person_2 с инициализацией, но нельзя определить объект Petrov класса person_1 с инициализацией:

```
person_2  Ivanov = {“Иванов И.И.”, 10000.00} ;  /* можно */
person_1   Petrov = {“Петров П.П.”, 10000.00} ;  /* нельзя */
```

- Кроме того, вне определения объекта нельзя обратиться к полям объекта Petrov.fio и Petrov.summa.
- В этом заключается свойство классов **инкапсуляция** – сокрытие информации от внешнего мира.

Объявления и определения

Для работы с полями объектов в определение классов включаются специальные функции, называемые **методами**. По умолчанию *методы* класса тоже недоступны извне класса, поэтому для управления доступом к ним используются ключевые слова **public, protected и private**.

- **public** (публичный, доступный) - открывает доступ,
- **private** (частный, закрытый) запрещает доступ,
- **protected** – защищенный, определяет поля, доступ к которым возможен только в классах, производных от этого.

Доступная часть называется **интерфейсной**.

В определении класса, также как и в определении структуры, может использоваться произвольное количество этих ключевых слов, очередное ключевое слово действует до следующего.

Рассмотрим определение и использование методов на примере класса **TMoney**, методов, которые необходимы для выполнения операций с деньгами, представленными рублями и копейками. Операции нужно выполнять с копейками, а вывод осуществлять в рублях и копейках. Для этого необходимо выполнять соответствующие преобразования.

Инициализация и вывод на экран денежных сумм

Метод можно определить внутри определения класса и тогда это может выглядеть следующим образом:

```
class Tmoney
    { //закрытое по умолчанию поле
     double summa;
    // открытый метод Init
public:
  void Init (const double &t)
    {// переводим в копейки, умножая на 100, и отсекаем
     лишние знаки, функцией  floor
          double r = floor ( ( (t<0)? –t: t) * 100);
          summa = (t<0)? –r: r;  // учитываем знак
        };
      };
```

Параметр t можно задавать по умолчанию, т.е. можно было заголовок функции записать в виде:

void Init (const double &t = 125.77).

Опишем экземпляр класса, объект obj1. **Tmoney obj1 ;**

Тогда вызов метода будет выглядеть так: **obj1.Init(245.94) ;…**

- В общем виде обращение к функции записывается в виде:

  <имя объекта> .<имя метода> (<параметры>)

- Если метод не возвращает значение, как в этом случае, то обращение к нему – это самостоятельный оператор, если возвращает значение какого либо типа, то обращение к методу так же, как и обращение к функции является указателем, который можно использовать в выражениях, в операторах присваивания.

- Если методов в классе много, например, кроме инициализации при работе с денежными суммами нужно уметь умножать на число, складывать суммы, делить на число и деньги на деньги, вычисляя проценты, выводить их на экран, то определение методов внутри класса становится неудобным, загромождает определение самого класса. Удобнее вынести определение всех методов за пределы определения класса, а внутри класса оставить только прототипы функций-методов. Например:

```
class Tmoney
     { //закрытые поля
double summa ;
// открытые методы
public:
// прототипы методов
   void Init (const double &t);
Tmoney AddMoney(const Tmoney &b) ;    // сложение сумм
Tmoney SubMoney(const Tmoney &b) ;     // вычитание сумм
double DivideMoney(const Tmoney &b) ;  // деление сумм
……….
Tmoney DivByNum(const double &b) ;      // деление на число
bool IsNegativ();                       //это долг?
int CompareMoney(const Tmoney &b) ;     //сравнение сумм
……..
void DisplayMoney() ;                   // вывод на экран
      }
```

При определении метода в этом случае в заголовке метода должно присутствовать явное указание, к какому классу он принадлежит. Это осуществляется с помощью имени класса и оператора принадлежности (**Tmoney::**), указанных слева от имени метода. Обращение к внутреннему и внешнему методу ничем не отличаются.

Определение методов

Определение указанных в описании класса методов располагаются далее по тексту, например для метода инициализации:

```
void Tmoney::Init (const double &t)
   { double r = floor ((( t<0) ? –t: t)* 100);
     summa = ( t<0)? –r: r;
    }
```

Рассмотрим для примера функцию вывода на экран, разделив ее на две.

Одна из них, **toString** формирует из денежной суммы – числа строку, а вторая - **DisplayTmoney** добавляет к этой строке название денежных единиц, в данном случае рубли и выводит полученную строку на экран.

Первая функция будет одинаковой для любых денежных единиц и ее можно расположить в невидимой (приватной) части определения класса, а вторая должна располагаться в интерфейсной части и будет доступна извне класса.

```
class Tmoney
     { //закрытые поля
double summa;
string toString();
// открытые методы
public:
// прототипы методов
……..
void DisplayMoney();              // вывод на экран
      }
```

Определения метода DisplayMoney может иметь вид:

```
void Tmoney::DisplayMoney()
{string s = toString();
// обратились к закрытой ф-и toString и получили число-строку
    s += " руб. ";      // добавили обозначение рублей
    cout  << s  << endl;
}
```

Определение метода toString:

```
string Tmoney:: toString()
{   string s = " ";              // s – строка результата
     string digits = "0123456789";   // цифры для результата
   tint dig;                  //выделенная цифра
   double t = fabs(summa);      //преобразуем в положительное число
   bool negative = (summa < 0);     // запоминаем знак
         tint kop = fmod(t, 100);        // выделили копейки
         t = floor(t /= 100 );          // отсекли копейки
   if (t > 0)                  // если есть рубли
      { while (t >= 1)            // формируем рубли
      { dig = fmod(t, 10);     //получаем цифру
        t /= 10;   s = digits[dig] + s;  //присоединяем символ-цифру
       };
      };
    else s += "0";                 // нет рублей
     s += ".";  s += digits[kop/10];    // завершаем целую часть и  выделяем
копейки
      s += digits[kop % 10];      if (negative) s = ' – ' + s;     // учитываем знак
      return s;
   }
```

…Обращение к открытой функции возможно для переменных, экземпляров данного класса, например, для описанного объекта **obj1**.

```
obj1.DisplayMoney();
```

Указатель this

Рассмотрим определение метода сложения денежных сумм.
Этот метод должен складывать два числа типа **Tmoney** и выдавать результат типа **Tmoney**. Но если к нему обращаются для какого-то объекта, то он уже имеет доступ к одному полю **summa**, значит для этой функции (метода) достаточно одного параметра.
Следовательно, вызов этого метода для объекта **obj1** может быть таким.

```
Tmoney obj1,obj2,obj3;
obj1.Init(100);
obj2.Init(200);
obj3 = obj1.AddMoney(obj2);
```

В этом случае определение метода **AddMoney** должно было иметь вид:

```
Tmoney Tmoney::AddMoney(const Tmoney &b)
{ Tmoney a = b;
//локальному параметру a присвоили значение  объекта-параметра
   a.summa += summa;
// сложили  с суммой текущего объекта
   return a;
}
```

Второй вариант определения этого метода:

```
Tmoney Tmoney::AddMoney(const Tmoney &b)
  { Tmoney a = *this;
// локальному параметру а присвоили значение текущего объекта
    a.summa += b.summa;
// сложили с суммой объекта-параметра
    return a;
  }
```

С помощью указателя **this** инициализировали локальную переменную **a** текущим объектом, то есть тем объектом, для которого этот метод будет вызван (присвоили полям локального объекта значения полей текущего объекта).

Каждая функция – метод, как говорит автор языка С++ Б. Страуструп, знает для какого объекта она вызвана, и может явно на него ссылаться. Ключевое слово **this** является указателем на объект, для которого этот метод вызван. Таким образом метод получает дополнительный неявный параметр - указатель на текущий объект, который иногда используют для обращения к полям объекта так:

this -> summa.

Перегрузка методов

- Одним из основных принципов ООП является ***полиморфизм,*** а одним из проявлений принципа полиморфизма является возможность переопределения функций, называемая ***перегрузкой*** функций. Методы это функции – члены класса и они также могут перегружаться. Рассмотрим перегрузку методов на примере метода суммирования денег **AddMoney**. Определим с этим же именем метод суммирования денег с числом, где число это денежная сумма – константа. Такой метод можно определить следующим образом:

```
Tmoney Tmoney::AddMoney(const double  &b)
      { Tmoney a;        // объявили локальный объект
        a.Init(b);
// инициализировали локальный объект формальным параметром
        return a.AddMoney(*this);
// сложили с суммой текущего объекта и возвратили его в качестве результата
        }
```

В теле перегруженного метода используются другие методы класса, метод инициализации и метод сложения с параметром – текущим объектом, заданным с помощью указателя this. Возврат результата суммирования можно реализовать и с помощью оператора:

```
return (*this).AddMoney(a);
```

Определение класса Tmoney и набора методов

```
……
typedef unsigned int tint;
class Tmoney
{ double summa;
   // приватные функции
   string ToString();
   double round (const double &r)
  { double t = fabs(r);  t = (t – floor(t) < 0/5)? floor(t) : ceil(t);
          return (r<0)? – t : t ;
        }
 public:
void Init(const double &t);
Tmoney AddMoney(const Tmoney &b);       // сложение сумм
Tmoney AddMoney(const double &r);        // сложение с константой
Tmoney SubMoney(const Tmoney &b);        // вычитание сумм
double DivideMoney(const Tmoney &b);     // деление сумм
Tmoney DivByNum(const double &b);        // деление на число
Tmoney MultByNum(const double &b);       // умножение на число
int CompareMoney(const Tmoney &b);       //сравнение сумм
bool IsNegativ();                        //это долг?
void ReadMoney();   void DisplayMoney();   // ввод и вывод
 };
```

Определение методов

```
// метод инициализации
void Tmoney::Init(const double &t);
{ summa = round(t*100); };
// вычитание сумм
Tmoney Tmoney::SubMoney(const Tmoney &b)
 { Tmoney t = *this;
    t.summa - = b.summa;
         return t;
       };
//деление сумм
double Tmoney::DivideMoney(const Tmoney &b)
 { return fabs (summa) / fabs (b.summa); };
// деление на число
Tmoney Tmoney::DivByNum(const double &b)
 {  Tmoney t = *this;
    if (b>0) t.summa = round (summa/b);
    return t;
     };
```

Определения методов

```
// сравнение сумм
int Tmoney::CompareMoney(const Tmoney &b)
 { int sign = 0;
    if(summa < b.summa) sign = -1;
    else if (summa > b.summa) sign = 1;
     return sign;
 };
// ввод
void Tmoney::ReadMoney()
 { tint k;
    cout << “рубли ” ; cin >> summa;        //ввод рублей
    bool negative = (summa < 0);          // сумма отрицательная
    summa = floar(fabs(summa)*100);  // формирование рублей
    cout << “коп ”; cin >> k;            // ввод копеек
    if (k < 100) summa += k;             // добавление копеек
    summa = negative ? – summa : +summa;  // учитываем знак
 };
……..return *this
```

Использование классов

.... **Пример 1**. пример различных способов инициализации.

```
#include<cstring>
#include<iostream>
#include<math.h>
using namespace std;
typedef unsigned int tint;
class Tmoney
 {   public:
 double summa;
        double round (const double &r)
    { double t = fabs(r);
      t = (t - floor(t) < 0.5)? floor(t) : ceil(t);
      return (r<0)? - t : t ;
          };
 void Init(const double &t);
 };
// метод инициализации
void  Tmoney::Init(const double &t)
 { summa = round(t*100); };
```

Использование классов

```
int main ()
{char c1;
 Tmoney t, p, s, z;     // объявление объектов t, p, s, z
   t.Init  (100.55);     // иниц-ия методом (t.summa = 100.55)
   p.Init(1000.66);
   Tmoney x = t;        //иниц-ия другой пер-ой (x.summa = t.summa)
   cout << t.summa <<“\t” <<p.summa << “\t” << endl;
   s.Init(0.0);          // обнуление:   s.summa = 0.0
   z.Init(0.0);          //……z.summa = 0.0
   cout << x.summa <<“\t” <<s.summa <<“\t” <<z.summa << endl;
   c1 = cin.get();
   return 0;
}
```

Для того, чтобы можно было использовать вывод

cout << t.summa <<'\t' …..

в определении класса поле summa описано как доступное, интерфейсное, перед ним записан признак доступа **public**.

Использование классов

Пример 2. (фрагмент программы)

```
// подключение библиотек
// определение класса Tmoney и его методов
int main ()
{ Tmoney t, p, s, z;
 t.Init  (100.55);   p.Init(1000.66);    Tmoney x = t;
    z = t.AddMoney(p);      // z.summa = t.summa  + p.summa
    z.DisplayMoney();
     s.Init(0.0);   z.Init(0.0);     // обнуление  s.summa и  z.summa
     s = t.AddMoney(p);   // s.summa = t.summa  + p.summa
     s.DisplayMoney();     // вывод на экран значения s.summa
     t.AddMoney(p).DisplayMoney();
 //суммирование и вывод на экран - t.summa+p.summa
     s = t.DivByNum(2);    s.DisplayMoney();
     z = p.MultByNum(3.15);   z.DisplayMoney();
  s.ReadMoney();      s.DisplayMoney();
     double d = p.DivMoney(s);
     cout <<d << endl;
     return 0;
}
```

Здесь реализовано определение и инициализация объектов с помощью метода и с помощью другой переменной, но реализованный ранее метод инициализации Init не позволяет определить инициализацию переменной типа Tmoney привычным образом, то есть, нельзя записать: **Tmoney u = 250.00;**
Такой вариант будет возможен, если в определении класса будет содержаться специальный метод, называемый ***конструктором***.
Рассмотрим вариант реализации метода **Init**, который позволит использовать еще один способ инициализации объектов.

```
Tmoney  Tmoney::Init(const double &t);
    { summa = round(t*100);
      return  *this;                                    };
```

В этом случае, кроме приведенных ранее способов инициализации, возможен и такой: **Tmoney t = z.Init(200.00),**
Но в этом случае мы инициализируем сразу два объекта: z и t. Это значит, что остается правильной конструкция **t.Init(200.00);**
В Примере 2. сложным кажется оператор **t.AddMoney(p).DisplayMoney();**
Работает он следующим образом: вначале выполняется (слева направо) метод суммирования **t.AddMoney(p)** и полученный результат нигде не сохраняется, а является временным объектом для выполнения второго метода – вывода на экран, то есть на экран будет выведена сумма:
t.summa + p.summa. ……

Использование классов: массивы объектов

Пример 3.

```
// подключение библиотек
// определение класса Tmoney и его методов
int main ()
{ Tmoney a[5], sa;        // объявление массива a  и переменной sa
   for (int i = 0; i < 5; i++)// инициализация элементов массива
 a[i].Init(i);
   for (int i = 0; i < 5; i++)// вывод элементов массива на экран
  a[i].DisplayMoney();
   sa.Init(0.0);                  // обнуление sa.summa = 0
 for (int i = 0; i < 5; i++)  // суммирование элементов массива
  sa = sa.AddMoney(a[i]);
   sa.DisplayMoney();     // вывод полученной суммы на экран
   sa.Init(0.0);
   for (int i = 0; i < 5; i++)// 2 вариант сум-ия элементов массива
  sa = a[i].AddMoney(sa);    sa.DisplayMoney();
   while(cin.get() != '\n');//задержка экрана до нажатия на enter
return 0;
}
```

Использование классов: работа с указателями на объекты

Пример4.

```
// подключение библиотек
// определение класса Tmoney и его методов
int main ()
{ Tmoney *pt = new Tmoney (), *pp, *ps;      // объявление указателей
     pp = new Tmoney ();   ps = new Tmoney (); //и создание объектов
     pt->Init(1000.24);  pt->DisplayMoney();  // инициализация и вывод
    pp->Init(2000.25);   pp->DisplayMoney(); // инициализация и вывод
    ps->Init(0.0);   ps->DisplayMoney();   // инициализация и вывод
    *ps = (*pt).AddMoney(*pp);                //сложение
    ps->DisplayMoney();
    ps->ReadMoney();  ps->DisplayMoney();      // ввод и вывод
    delete pp; delete ps; delete pt;    // уничтожение объектов
    while(cin.get() != '\n');       //задержка экрана до нажатия на enter
    return 0;
    }
```

Для работы с указателями на объекты используются операция доступа к указателю **(->)** и операция разыменования **(*)**.

Указатели **(*pt)** и **(*pp)** взяты в скобки для того, чтобы явно указать последовательность выполнения операций в выражении **(*pt).AddMoney(*pp),** так как операция **(.)** имеет более высокий приоритет, чем операция разыменования **(*)**. В примере для каждого указателя операцией new создается динамический объект типа **Tmoney**, а затем выполняется его инициализация. Объект создается с помощью конструктора - метода, имеющего тоже имя, что и тип класса – **Tmoney ().**

```
Tmoney *pt, *pp, *ps;          // объявление указателей
Tmoney *pt = new Tmoney ();
// объявление указателя pt и создание объект а
pp = new Tmoney;               // создание объектов pp, ps
ps = new Tmoney;
```

Пустые скобки после имени конструктора можно не писать, но при использовании конструктора с пустыми скобками, созданный объект инициализируется нулями, а при использовании конструктора без скобок начальные значения полей данных не определены.

После работы с динамическими объектами их необходимо удалить, это выполняется здесь операторами **delete**

Использование классов: *включение* или *композиция*

Пример, в котором класс используется для описания поля в определении другого класса. Такая возможность называется *включением* или *композицией*. Определим класс, моделирующий счет в банке, в нем обязательными полями должны быть номер счета и сумма на счете. А в качестве методов рассмотрим методы инициализации, суммирования (добавления денег на счет) и вывода на экран полей класса. **Пример 5.**

```
typedef  unsigned long ulong;
class Tcount
{    Tmoney summa;    // сумма на счете – объект типа Tmoney
ulong NumCount;        // номер счета
public:
void DisplayCount()          // вывод на экран полей класса
   {   cout << ”Number: ” << NumCount << “  .summa:  ”;
       summa.DisplayMoney();
    };
// инициализация номера счета и суммы на счете
void Init (ulong number, const Tmoney &s)
   { NumCount = number;   summa = s;      };
// сложение денежных сумм на счете
Tcount  AddSumma(Tmoney s)
   { Tcount t = * this;
     t.summa = t.summa.AddMoney(s); //  «вложенный» доступ
     return t;
   };
}
```

В этом примере методы класса **Tmoney** использовались в методах класса **Tcount**.

Использование классов Tcount и Tmoney

Пример 6.

```
// подключение библиотек
// определение классов Tmoney и Tcount и их методов
int main ()
{ Tmoney  p;          // объявление объекта типа Tmoney
   p.Init(100.15);          // инициализация денег p.summa
   Tcount tt, pp;      // объявление объектов типа Tcount
// инициализация полей объекта tt (номера счета и суммы на счете)
   tt.Init(1, p);
   tt = tt.AddSumma(p);      // добавление денег на счет
   tt.DisplayCount();     // вывод на экран значений полей объекта tt
// инициализация объекта pp выражением
   pp.Init(2, p.MultByNum(1.5));
// вывод на экран значений полей объекта pp
   pp.DisplayCount();
   return 0;
}
```

Использование классов, наследование

```
1 #include<iostream>
2 using namespace std;
3 class DataConvert {
4   protected:
5   int convert(char ch) {return ch - '0';};
6   };
7   class DataStore: DataConvert {
8   private:
9   static const int maxs = 9;
10  int size  ;
11  int ci;
12  int store[maxs];
13  public:
14    int initial(char a)
15        { ci = 0;  return size = convert(a); };
16    void save(char a)
17        { store[ci++] = convert(a); };
18    int setprint()
19        { ci = 0; return size; };
20    int printval()
21        { return store[ci++]; };
```

```
 22          int sum()
 23          { int arrsum;
 24            arrsum = 0;
 25            for(ci=0; ci<size; ci++) arrsum = arrsum + store[ci];
 26            return arrsum;
 27          };
28        };
 29       int main()
 30       {  int j,k;
 31           DataStore x;
 32           while (( k = x.initial(cin.get())) != 0 )
33             { cout <<k <<'\n';
34               for (j=0; j<k; j++)
35                  { x.save(cin.get());};
 36           cout <<'\n';
 37          for (j=x.setprint(); j>0; j--)
 38             { cout << x.printval( )<< '\t';};
 39           cout <<":SUM=" << x.sum() << endl;
 40            while(cin.get() != '\n');
 41           }
 42         return 0;
 43       }
```

Описание примера – использование наследования

1 и 2 строки – подключение заголовочного файла для потоков cin и cout, и определние пространства имен.
Строки 3 – 6 определяют класс DataConvert, в котором определен один метод – функция convert, преобразующая символ – цифру в цифру (char-цифру в int-цифру, например, conver(‘1’) = 1), причем этот метод имеет *защищенный* класс доступа protected, это значит, что он может использоваться в любом классе, производном от класса DataConvert, в любом классе, являющемся его наследником. Наследование – это одно из основных свойств ООП.
В строке 7 содержится заголовок определения класса DataStore

class DataStore: DataConvert

После имени определяемого класса записывается двоеточие и имя класса, поля и методы которого наследует, может использовать этот определяемый класс. Строки 8-12 описывают поля – данные как частные, приватные, которые могут использоваться только в классе DataStore. Строки с 14 по 28 определяют методы класса, причем все методы класса будут доступны вне класса, об этом говорит ключевое слово public в стр.13.

Строки 16 и 17 определяют класс save, для создания массива store, который будет невиден вне определения класса, говорят, что он инкапсулирован в классе, к нему можно обращаться только с помощью методов.

Строки 18 и 19 определяют метод setprint, который задает количество выводимых элементов массива и начальное значение индекса.

20 и 21 строки определяют метод printval, использующийся для вывода текущего элемента массива. Для вывода всех элементов массива он в основной программе вызывается в цикле.

Суммирование элементов массива осуществляется в методе sum, который определен строками 22-28.

Головная программа, функция main – это строки 29 – 43. Цикл в строке 32 с помощью функции get класса cin, определенного в заголовочном файле **iostream**, вводит символ-цифру, преобразует его в цифру с помощью метода **x.initial**, присваивает переменной k, сравнивает его с нулем, и если k не равно нулю, то происходит обработка массива, строки 33 -37. В строках 33-41 в цикле вводятся элементы массива, методом **x.save(cin.get).** Функция – параметр вводит символ с экрана, метод **save** преобразует его в цифру и присваивает очередному элементу массива.

Строки 36-38 в цикле по j от размерности массива, определенного методом **setprint**, до нуля выводит на экран значения элементов массива методом **x.printval**.

Строка 39 обращается к методу **x.sum**, и выводит полученное значение суммы элементов массива на экран.

Конструкторы и деструкторы

- *Конструктор* – это специальный метод, который предназначен для инициализации объекта, он имеет имя, совпадающее с именем класса. Конструктор не должен возвращать значение, даже void, нельзя получать указатель на конструктор.
- Класс может иметь несколько конструктов с различными параметрами для реализации различных типов инициализации, в этом случае при вызове конструктора происходит перегрузка конструктора.
- Конструктор может быть без параметров, в этом случае он называется конструктором по умолчанию.
- Параметры конструкторов могут быть любого типа, кроме типа этого же класса.
- В заголовке конструктора можно задавать значения параметров по умолчанию, если в классе несколько конструкторов, то только один может иметь значения по умолчанию.
- Если программист не создал в классе конструктор, то автоматически создается и вызывается конструктор по умолчанию для инициализации полей данных объекта. Но если в таком классе использованы константы или ссылки, то компилятор выдаст сообщение об ошибке, так как их необходимо инициализировать конкретными значениями, конструктор по умолчанию не сможет этого сделать, значит, в этом случае в классе должен быть хотя бы один конструктор.

- Конструкторы не наследуются потомками и их нельзя писать с модификаторами **const, static и virtual**. Конструкторы глобальных объектов необходимо вызывать до функции **main**.
- В примерах с денежными суммами использовались для инициализации различные определения метода **Init**, посмотрим, какие должны быть в классе конструкторы, чтобы можно было реализовать рассмотренные способы инициализации полей данных класса **Tmoney**.
- Было использовано три способа объявления объектов класса **Tmoney**:

1. без инициализации **Tmoney t;**
2. с инициализацией **Tmoney t = (100.25);**
3. с инициализацией другим, уже определенным объектом

Tmoney r = t;

Столько же и конструкторов должно быть в классе.

1. Конструктор без параметров, создаваемый системой по умолчанию, если в классе нет других конструкторов. Общий вид его

<имя_класса> () { };

2. Конструктор инициализации, таких в классе тоже может быть несколько.
3. Конструктор копирования

Класс **Tmoney** с конструкторами может выглядеть так:

```
class Tmoney
{  double summa;
    public:
    Tmoney () { summa = 0.0; };
    Tmoney (const double &t = 0.0);
     Tmoney (const Tmoney &r);
……….
};
```

Конструктор по умолчанию без параметров, поэтому он просто обнуляет поля данные. Определение конструктора инициализации должно совпадать с определением метода Init:

```
Tmoney::Tmoney (const double &r )
 { summa = round(r*100);  }
```

Конструктор копирования:

```
Tmoney::Tmoney (const Tmoney  &p )
 { *this = p;  }
```

Этот конструктор копирует поля уже существующего объекта p в поля вновь создаваемого объекта.

Примеры инициализации объектов типа **Tmoney**:

Tmoney d1; // инициализация полей нулями

Tmoney d2(120.55); //явный вызов конструктора инициализации

Tmoney d3 =100; //инициализация временного объекта числом, затем копирование его в d3

Tmoney d4 = Tmoney(100); //явный вызов конструктора, создание временного объекта и затем копирование в d4

Tmoney d5(d2); //конструктор копирования

Tmoney d6 = d2; // конструктор копирования

При работе с динамическими объектами одинаково выполняются операторы:

Tmoney *p =new Tmoney;

Tmoney *p =new Tmoney();

Можно создавать объект и присваивать начальные значения **Tmoney *p =new Tmoney(100.25);**

Деструктор

При создании объектов выполняются конструкторы, которые в любом классе присутствуют – создаются неявно, автоматически системой по умолчанию или определяются явно программистом.

Но объекты создаются, «живут» и объекты, выполнив свою задачу, «умирают», должны быть уничтожены. Для этой цели и существуют методы, называемые деструкторами.

Деструктор – это метод, имеющий тоже имя, что и соответствующий класс, с добавленным слева символом **тильда**.

Деструктор в классе может быть только один, он не наследуется, не имеет параметров и не выдает значение.

Деструктор вызывается автоматически при уничтожении любого объекта. Для локальных объектов это происходит при выходе из блока, в котором этот объект действовал, а для динамических объектов, которые программист создавал с помощью операции **new**, это происходит при вызове операции **delete**.

Если деструктор не определен явно программистом, то он создается автоматически системой по умолчанию. Деструктор по умолчанию имеет вид: **~<имя_класса> () {};**

Обычно деструктор определяется программистом явно в том случае, когда ему необходимо вместе с уничтожением объекта выполнить какие-то специальные действия, например, закрыть файл, открытый в конструкторе.

Наследование

- Механизм наследования присущ только ООП…Наследовать – это значит использовать поля данные и методы. Можно изменять и дополнять свойства родительских классов.
- Общие для нескольких классов свойства объединяются в один, который затем используется как базовый. По мере продвижения вниз по иерархии свойства классов конкретизируются от более общих к частным.
- Существует простое и множественное наследование. Простым называется наследование, при котором класс потомок имеет одного родителя. Множественное наследование позволяет одному классу наследовать свойства двух или более классов.

class <имя производного класса> : <ключ доступа> <имя базового класса> {<тело класса>}

- Здесь ключи доступа - это **private, protected и public**.
- Если наследование множественное, то имена базовых классов записываются через запятую и перед каждым из них может стоять ключ доступа. Например,

class X {….};
class Y {…};
class Z {….};
class W: X, protected Y, public Z {……};

Наследование

- **X,Y,Z** – это базовые классы, класс **W** может использовать определенные в них поля и методы, но с ограничениями, накладываемыми ключами доступа. Класс **X** по умолчанию имеет ключ доступа **private**.
- Но в теле классов **X,Y и Z** для различных элементов (полей-данных и методов) могут применяться разные спецификаторы доступа **private, protected и public** и влияние их проявляется при наследовании различно.
- **private**-элементы базового класса недоступны в потомках независимо от ключа доступа, к ним потомок может обращаться только через методы базового, родительского класса. Правила доступа к **protected и public-**элементам базовых классов изменяются в соответствии с ключами доступа, указанными при определении классов потомков. Это хорошо видно в следующей таблице:

Ключ доступа	Спец-ор в базовом классе	Доступ в произв-ом классе
private	**private**	**Нет доступа**
	protected	private
	public	private
protected	**private**	**Нет доступа**
	protected	protected
	public	protected
public	**private**	**Нет доступа**
	protected	Public
	public	public

Наследование

Если родительский класс наследуется с ключом **private**, то некоторые его элементы можно сделать доступными в классе потомке, описав его в теле потомка с нужным ключом доступа, например:

```
class roditel
{ ……………
public:
int metod_1 ();
……..
 };
class  potomok : private roditel
 { ……
      public:  roditel :: int metod_1();
 ……..;
 };
```

Метод с именем **metod_1** класса **roditel** будет общедоступным в объектах класса **potomok**.

Пример наследования

```
// подключение библиотек, методы определены внутри класса
class xmax  // родительский класс
{ protected: static const int n = 2;
    int x[n];
int max_x (int x[n])
    {  int i,  s=x[0];
       for (i=0; i<n; i++) if (s<x[i])  s = x[i]; //поиск максимума
       return s;
    };
};
    class tabl: public xmax      //tabl – потомок
     {     public:   int sum_max ( int y[n][n] )  //сумма максимумов строк
             {int i, j , sum;  sum = 0;
                for (i=0; i<n; i++)
           { for (j=0; j<n; j++) x[ j ] = y[i][j];
                sum += max_x(x);
           }; return sum;
        }        };
```

```
    int main ()
{ const int m = 2;
    tabl obj1,obj2;
    int i, j, rez;    int  arr1[m][m] = {{7,5},{3,4}};    int arr2 [m][m];
    rez = obj1.sum_max(arr1);
    cout <<“rez” << rez <<“\n”;  //сумма максимумов строк arr1
    cout <<“vvedite massiv arr2” <<“\n”;
    for (i = 0; i < m; i++)
      for(j = 0; j < m; j++)  cin >> arr2 [i][j];
    rez = obj2.sum_max(arr2);
    cout << “rez=” << rez <<“\n”; // сумма максимумов строк arr2
    while (cin.get() != ”0”);
    return 0;
}
```

Наследование

Для разных методов класса существуют разные правила наследования.

Конструкторы не наследуются, поэтому производные классы должны иметь собственные конструкторы.

- Если в конструкторе потомка отсутствует явный вызов конструктора предка, то автоматически вызывается конструктор предка по умолчанию (конструктор без параметров).
- Если используется иерархия классов, то конструкторы вызываются, начиная с самого верхнего уровня.
- При множественном наследовании конструкторы вызываются в порядке указания базовых классов в определении производного класса.

- *Деструкторы* не наследуются и, если программист не описал деструктор в производном классе, то он создается автоматически и вызывает деструкторы всех базовых классов.
- Если программист описал деструктор в производном классе, то нет необходимости (в отличие от конструктора) явно вызывать деструкторы базовых классов – это выполнится автоматически.
- Для иерархии классов деструкторы вызываются в порядке строго обратном вызову конструкторов, то есть последним вызывается деструктор базового класса.

Наследование

При множественном наследовании может оказаться, что наследуются элементы различных классов с одинаковыми именами. В этом случае избежать конфликта имен позволяет использование операции доступа к области видимости, то есть явное указание класса, элемент которого должен быть использован, например: **class X**

```
        {……
         public: int ff1 ();
          ……
        };
class Y
{……
public: int ff1 ();
  ……
};
        class Z: public X, public Y
        {……
        };
int main ()
{……..
          Z obj1;
  cout << obj1.X:: ff1();     cout << obj1.Y:: ff1();
};
```

Наследование

Если у двух или более базовых классов есть общий предок **(ромбовидное наследование),** то производный от этих базовых унаследует два или соответственно более экземпляров элементов этого общего предка, что, нежелательно. Чтобы этого не случилось, нужно при наследовании общего предка определить его как виртуальный:

```
class monstr
{ …….. };
class demon: virtual public monstr
{ …….. };
class lady: virtual public monstr
{ …….. };
class baby: public demon, public lady
{ …….. };
```

Класс **baby** имеет два базовых класса **demon** и **lady**, а они в свою очередь имеют предка **monstr**, поэтому класс **baby** будет наследовать и его поля, а за счет ключевого слова **virtual**, указанных при описании классов **demon** и **lady**, поля их общего предка наследуются в классе **baby** только один раз. Множественное наследование используется часто, когда один из базовых является основным, а другие дополняют некоторыми свойствами, их называют **классами подмешивания**.

Статические, поля и методы

Статические поля и методы описываются с помощью спецификатора static, доступны в пределах класса. Статические поля используются для хранения данных, общих для всех объектов данного класса, они существуют в единственном экземпляре, то есть не дублируются для каждого экземпляра класса, память под статические поля выделяется один раз при его инициализации независимо от числа созданных объектов и даже при их отсутствии. Инициализироваться статическое поле может с помощью спецификатора доступа к области действия:

```
class  A
{ public: static int count;  };
int A::count; ….     int A::count=10……
A *x, y;
……
cout << A::count << x->count << y.count   ;
```

На экран будет выведено одно и тоже значение, обращение к статическому полю возможно с помощью имени класса **A::count** и с помощью имен объектов этого класса **x->count** и **y.count** (**x** здесь указатель на объект).

Статические, поля и методы

Статические методы предназначены для работы со статическими полями класса. Они могут обращаться только к статическим полям или другим статическим методам, так как им не передается скрытый указатель **this**. Статические методы не могут быть константными и виртуальными, т.е. их нельзя описывать со спецификаторами **const** и **virtual**. Обращение к статическим методам также как и к статическим полям возможно с помощью имени класса или имени объекта этого класса. Например,

```
class A
{  static int count: //приватное статическое поле count
 public:
      static void inc_count () {count++};
  ……….
};
 int A:: count;    // иниц-ия в глобальной области по умолчанию нулем
void f ()
{ A obj1;
  obj1.inc_count();       //обращение через имя объекта
  A::inc_count();   //обращение через имя класса
```

обращаться к полю напрямую, например, **obj1.count++** нельзя, т.к. оно скрыто по умолчанию.

Дружественные функции и дружественные классы

Прямой доступ к некоторым полям извне класса можно сделать с помощью *дружественных функций и дружественных классов.* **Дружественная функция** объявляется ключевым словом **friend** внутри класса, к элементам которого ей нужен доступ. В качестве параметра у нее должен быть объект или ссылка на объект, т. к. указатель **this** ей не передается. Дружественная функция может быть обычной функцией или методом другого класса. Размещаться она может в любом месте определения класса и на нее не распространяется действие спецификаторов доступа. Одна функция может быть дружественной нескольким классам. **class X1;** //предварительное объявление

```
class X2
{  public: void ff1(X1 &);
……..……
};
class X1     //полное определение классаX1
{  ……..…
friend void X2::ff1(X1 &);   //метод класса X2
};
```

Дружественные функции и дружественные классы

void X2::ff1(X1 &M) { M.pp = 0;} // определение метода класса X2.
Если все методы одного класса должны иметь доступ к скрытым полям другого класса, то весь первый класс объявляется дружественным другому классу с помощью ключевого слова friend.

```
class Y
{ ……..
  void f1( );
   void f2( );
}
class X
{……..  friend class Y; ……};
```

Методы **f1 и f2** являются дружественными классу **X** и имеют доступ ко всем его скрытым полям. Таким образом можно сделать доступными скрытые поля, однако по возможности лучше этим не пользоваться, так как это нарушает одно из основных свойств ООП – инкапсуляцию.

Виртуальные методы

Виртуальные методы – это методы, определенные с ключевым словом **virtual**. Использование их связано с реализацией полиморфизма в ООП. Работа с объектами чаще всего реализуется через указатели. Указателю на базовый класс можно присвоить адрес объекта любого производного класса, например,

```
class Pointer
{ …….
void draw (<параметры>);
};
class Line : public Pointer
{ ……
  void draw(<параметры>);
};
……..
Pointer *p;   //указатель на базовый класс
p = new Line; // указатель ссылается на производный класс Line
p -> draw (<параметры>);  // осуществляется вызов метода класса
```

Pointer в соответствии с типом указателя, а не фактическим типом объекта Line, на который он ссылается в данный момент, так как на этапе компиляции программы произошло связывание указателя p с классом Pointer. Такое связывание называется **ранним**.

Виртуальные методы

- Для того, чтобы обратиться к методу draw класса Line, можно выполнить явное преобразование типов:

  ((Line *p)) ->draw(<параметры>);
- Но это не всегда возможно сделать, так как в разное время указатель может ссылаться на объекты разных классов иерархии и во время компиляции программы конкретный класс может быть неизвестен. Таким примером может быть функция, параметром которой является указатель на объект базового класса. Во время выполнения программы на его место может быть передан в качестве фактического параметра указатель на любой производный класс.
- Для выхода из такой ситуации в ООП реализован механизм **позднего связывания**, который заключается в том, что определение ссылок, их связывание происходит не на этапе компиляции, а на этапе выполнения программы в соответствии с конкретным типом объекта, вызвавшего метод.
- Этот механизм реализован с помощью виртуальных методов.

  virtual <тип> <имя метода> (<параметры>);

Правила описания и использования виртуальных методов:

- если в базовом классе метод определен как виртуальный, то определенный в производном классе метод с тем же именем и тем же набором параметров будет виртуальным автоматически, если набор параметров будет другим, то, несмотря на тоже имя, метод будет обычным.
- виртуальные методы наследуются, а это значит, что переопределять их нужно только в том случае, если нужно выполнить другие действия. Права доступа при переопределении изменить нельзя.
- если виртуальный метод переопределен в классе-потомке, то объекты этого класса могут получить доступ к одноименному методу класса-предка с помощью операции доступа

  к области видимости.
- виртуальный метод не может быть статическим и дружественным.
- если в классе описан виртуальный метод, то он обязательно должен быть определен, возможно, только как *чисто виртуальный.*

Чисто виртуальный метод

Чисто виртуальный метод – это метод, в котором вместо тела содержатся символы **= 0**,
например: **virtual void f(int a) = 0;**
Чисто виртуальный метод должен переопределяться в производном классе, возможно тоже как чисто виртуальный. Если в классе Line метод draw определен как виртуальный, то решение о том, метод какого класса будет вызываться, зависит от типа объекта, на который ссылается указатель:

Pointer *p, *r;
r = new Pointer; // создается объект класса Pointer
p = new Line; // создается объект класса Line
r -> draw (<параметры>); // вызывается метод Pointer::draw
p -> draw (<параметры>); // вызывается метод Line::draw
p -> Pointer::draw (<параметры>) // обход механизма виртуальных методов

- Если какой-то метод класса Line будет вызывать метод draw не непосредственно, а из любого другого класса, наследованного из класса Pointer, то будет вызван метод draw класса Line. Это обеспечивается механизмом *позднего связывания*.

Позднее связывание

Механизм *позднего связывания* заключается в следующем: для каждого класса, содержащего хотя бы один виртуальный метод, создается во время компиляции таблица виртуальных методов **(vtbl),** в которой записаны адреса виртуальных методов.

Каждый объект такого класса содержит дополнительное скрытое поле - ссылку на таблицу виртуальных методов, называемую **vptr**. Это поле заполняется конкретным значением при выполнении конструктора при создании объекта.

На этапе компиляции ссылки на виртуальные методы заменяются обращениями к **vtbl** через **vptr** объекта, а на этапе выполнения в момент обращения к методу его адрес выбирается из таблицы.

Наличие виртуальных методов замедляет выполнение программы за счет обращений к vtbl. Нет правил какие методы следует делать виртуальными, а какие нет. Рекомендуется делать виртуальными деструкторы и те методы, о которых известно, что они будут переопределяться в производных классах. Методы, которые во всей иерархии останутся неизменными и те, которыми не будут пользоваться потомки, делать виртуальными нет смысла, однако при наличии в программе сложной иерархии классов сложно предугадать заранее какие методы придется переопределять.

Абстрактные классы

Механизм позднего связывания работает только при использовании указателей или ссылок на объекты. Объекты, определенные через указатели или ссылки и содержащие виртуальные методы, называются **полиморфными**.

Полиморфизм – это интерпретация вызовов функций на этапе выполнения программы – обращения к одному и тому же методу вызывают различные действия в зависимости от типа, на который ссылается указатель в момент выполнения программы.

Класс называется ***абстрактным***, если он содержит хотя бы один *чисто виртуальный* метод.

Абстрактные классы предназначены для представления общих понятий, которые должны уточняться, конкретизироваться в производных классах.

Абстрактный класс может создаваться только для того, чтобы быть **базовым**.

Нельзя создавать объекты абстрактного класса, так как нельзя обращаться прямо или косвенно к чисто виртуальным методам.

Допускается объявлять указатели и ссылки на абстрактный класс, если при инициализации не требуется создавать временный объект.

Абстрактные классы

Используя абстрактные классы, можно создать функцию, формальным параметром которой является указатель на абстрактный класс.

Тогда при выполнении программы в качестве фактического параметра может использоваться указатель на объект любого производного класса.

И это позволяет создавать полиморфные функции, работающие с объектами любого типа в пределах данной иерархии классов.