Основы ООП в C++

Объектно-ориентированное программирование

Идея классов является основой объектно-ориентированного программирования (ООП).

Класс является типом данных определяемым пользователем. *В классе задаются* свойства и поведение какого-либо предмета или процесса в виде полей данных (аналогично структуре) и функций для работы с ними (методов).

Интерфейсом класса являются заголовки его методов.

Конкретные величины типа данных «класс» называются *экземплярами класса, или объектами.*

Основными принципами ООП являются:

1. Инкапсуляция;
2. Наследование;
3. Полиморфизм.

Описание класса

Класс является абстрактным типом данных, определяемым пользователем, и представляет собой модель реального объекта в виде данных и функций для работы с ними.

Данные класса называются ***полями (свойствами)***, *а функции* класса — ***методами.*** Поля и методы называются элементами класса.

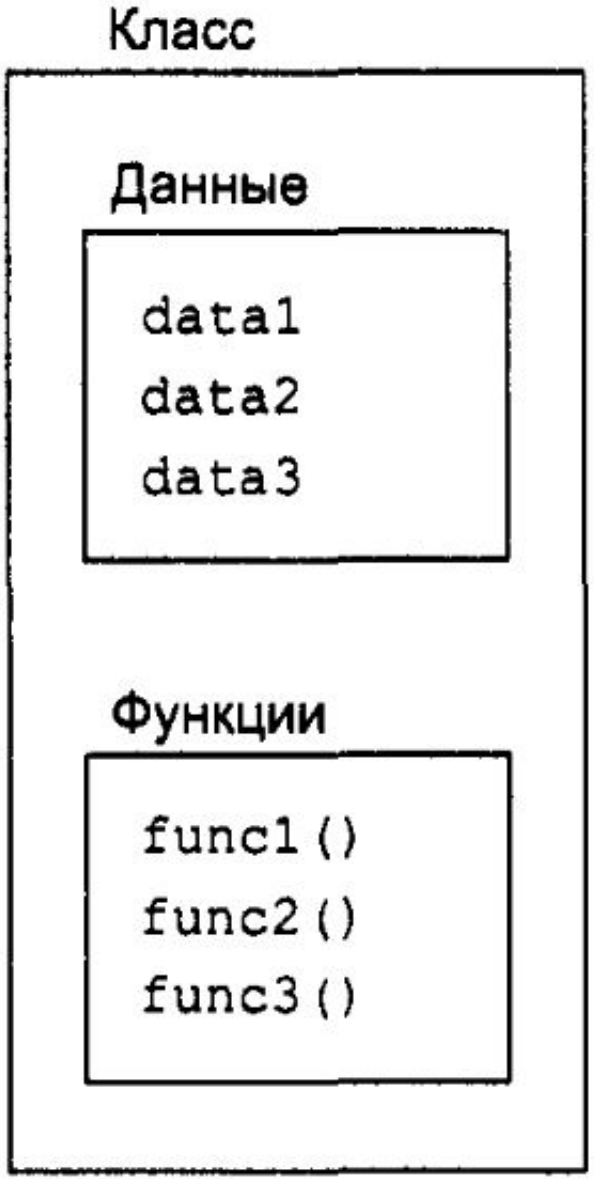


Рис. Класс содержит данные и функции

При описании класса реализуется одно из ключевых понятий ООП - ***инкапсуляция***. Для начала дадим формальное определение этого понятия:

Инкапсуляция - это механизм, который объединяет данные и методы, манипулирующие этими данными, и защищает и то и другое от внешнего вмешательства или неправильного использования.

Описание класса выглядит так:

class <имя>{

[private:]

<описание скрытых элементов>

public:

<описание доступных элементов>

} ;

Где спецификаторы доступа **private** и **public** управляют видимостью элементов класса. Элементы, описанные после служебного слова **private,** видимы только внутри класса. Этот вид доступа принят в классе по умолчанию.

Термин **private** понимается в том смысле, что данные заключены внутри класса и защищены от несанкционированного доступа функций, расположенных вне класса. Такие данные доступны только внутри класса.

Данные, описанные ключевым словом **public**, напротив, доступны за пределами класса.

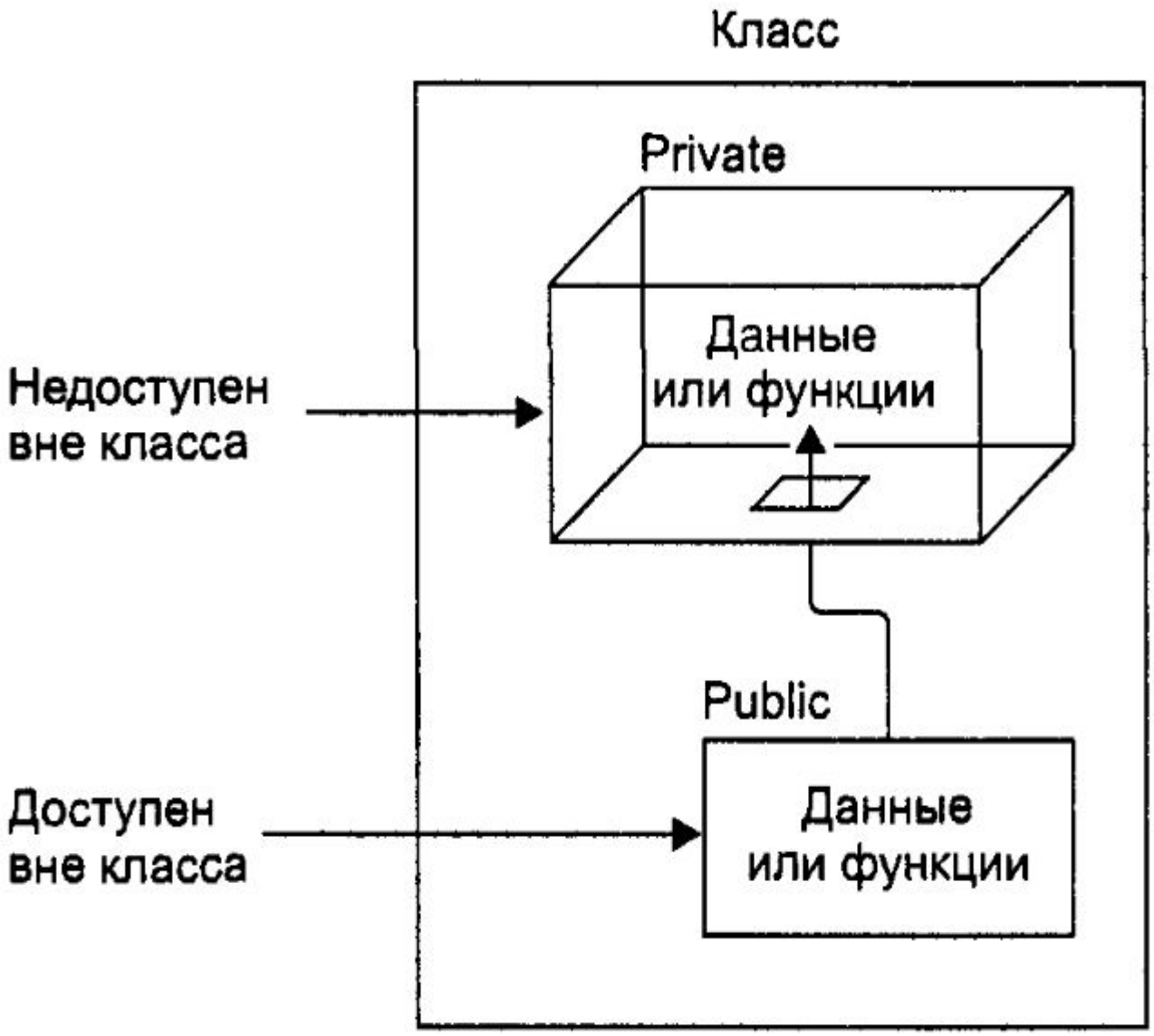


Рис. Скрытые и общедоступные классы

Поля класса:

• могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);

• могут быть описаны с модификатором const, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться.

Инициализация полей при описании <u>не допускается.</u>

Методы класса – это функции входящие в состав класса.

Описание объектов

Конкретные переменные типа «класс» называются ***экземплярами класса***, *или* ***объектами***.

Время жизни и видимость объектов зависит от вида и места их описания и подчиняется общим правилам C++.

Формат:

class <имя> переменная;

Замечание: Объект находится в таком же отношении к своему классу, в каком переменная находится по отношению к своему типу.

Пример1.

Рассмотрим **пример** описания класса **TPoint** (точка):

1) Точка характеризуется координатами: X и У – это свойства объекта;
2) Над точкой можно выполнять следующие действия:

- можно задать её координаты;
- точку можно переместить (изменив координаты);
- можно получить(узнать) координаты точки.

```
#include "stdafx.h "
class   TPoint
  { private:
                    int  x,y;
   public:
   void InitPoint ( int newx, int newy)
        { x = newx;   y = newy ;  }

   void relmove  ( int dx, int dy )
   {x+= dx;   y += dy ;  }

   int   getx ( void ) { return x ; }
   int   gety ( void ) { return y ; }
  };
```

```
int main()
  {     class   TPoint p;
        p.InitPoint(10,10);
        printf("x=%d, y=%d\n", p.getx(), p.gety());
     p.relmove(5,10);
     printf("x=%d, y=%d\n", p.getx(), p.gety());
     return 0;
  }
```

В этом классе два скрытых поля — ***x*** и ***y***, получить значения которых извне можно с помощью методов getx() и gety().

Все методы класса имеют непосредственный доступ к его скрытым полям.

C:\WINDOWS\system32\cmd.exe

```
x=10, y=10
x=15, y=20
Для продолжения нажмите любую клавишу . . .
```

Классы могут быть ***глобальными*** (объявленными вне любого блока) *и* ***локальными*** (объявленными внутри блока, например, функции или другого класса).

МЕТОДЫ КЛАССА

Вызов методов класса

При вызове методов необходимо связать метод с объектом этого класса. Поэтому имена объектов (p) связаны с именем функции (метода) операцией точка (.):

p.InitPoint(10,10); p.relmove(5,10);

Замечание: Это напоминает доступ к полям структуры.

Вызов методов **InitPoint()** и **relmove()** приводило к изменению значений полей X и У. В следующем примере задание полей в методе **Vvod()** будет осуществляться с клавиатуры:

```
C:\Dev-Cpp\fff.exe
Vvide X Y:50 60
x=50, y=60
Для продолжения нажмите любую клавишу . . .
```

Тогда вызов этого метода будет: p.Vvod();

Конструкторы

В отличии от предыдущего примера удобнее инициализировать поля объекта автоматически в момент его создания, а не явно вызовом соответствующего метода. Такой способ реализуется с помощью особого метода класса, называемого конструктором.

Конструктор - *это метод, выполняющийся автоматически в момент создания объекта*.

Конструктор отличается от других методов:

1) **Имя конструктора совпадает с именем класса**;

2) **У конструктора не существует возвращаемого значения.**

```
class   TPoint
   { private:
     int  x,y;
     public:
     TPoint(int newx, int newy)  // конструктор
          {x=newx;    y=newy; }
     void relmove  ( int dx, int dy )
     {x+= dx;   y += dy ;  }
     int   getx ( void ) { return x ; }
     int   gety ( void ) { return y ; }  };
int main(int argc, char *argv[])
{  class   TPoint    p(10,10); //инициализация объекта p
    printf("x=%d, y=%d\n", p.getx(), p.gety());
        …}
```

Пример 2.

В качестве примера создадим класс Counter, объекты которого могут хранить количественную меру какой-либо изменяющейся величины.

При наступлении некоторого события счетчик увеличивается на единицу. Обращение к счетчику происходит, как правило, для того, чтобы узнать текущее значение той величины, для изменения которой он предназначен.

```
class Counter
{  private:    int  mycount;
    public:
    Counter ( ):mycount(0){ }//конструктор
    void inc_count ( ) {mycount++; }   //метод
    int   get_count( ) { return mycount; }  //метод
    };
int _tmain(int argc, _TCHAR* argv[])
{  Counter c1, c2;    //описание объектов класса
  std::cout<<c1.get_count()<<std::endl;
  std::cout<<c2.get_count()<<std::endl;
      c1.inc_count();    c2.inc_count();
  std::cout<<c1.get_count()<<std::endl;
  std::cout<<c2.get_count()<<std::endl;
    return 0;}
```

Одной из наиболее часто возлагаемых на конструктор задач является инициализация полей объекта. Инициализация полей обычно реализуется с помощью списка инициализации, который располагается между заголовком и телом функции-конструктора. После заголовка ставится двоеточие. Инициализирующие значения помещены в круглые скобки после имени поля.

Counter ( ): mycount(0)

{/* тело функции*/ }

Если инициализируются несколько полей, то значения разделяются запятыми.

Деструкторы

Кроме специального метода конструктор, который вызывается при создании объекта, существует другой особый метод, автоматически вызываемый при уничтожении объекта, называемый ***деструктором***.

Деструктор имеет имя, совпадающее с именем конструктора, перед которым стоит тильда ~.

```
class   Prim
  { private: int  dat;
    public:
    Prim(): dat(0)
     {  }
   ~Prim()
     {   }
    };
```

Наиболее распространённое применение деструкторов – освобождение памяти, выделенной конструктором при создании объекта.

Определение методов может быть реализовано как внутри самого класса, так и вне класса. Во втором случае внутри класса содержится лишь прототип функции, а сама функция определяется позже. Тогда перед именем функции указывается имя класса и символ **::**

```
class   TPoint
   { private:
    int  x,y;
    public:
        TPoint(int newx, int newy)
         {x=newx;y=newy; }
    void relmove  ( int dx,  int dy  );
    int   getx ( void ) { return x ; }
    int   gety ( void ) { return y ; }
        ~Tpoint() {  }
   };
 void TPoint::relmove(int dx, int dy)
{x+= dx;   y += dy ;  }
```

ЗАМЕЧАНИЯ:

1)Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации.

2)Конструктор, вызываемый без параметров, называется конструктором по умолчанию.

3)Деструктор вызывается автоматически, когда объект выходит из области видимости:

- для ***локальных объектов*** — при выходе из блока, в котором они объявлены;
- для ***глобальных*** — как часть процедуры выхода из main( );
- для объектов, заданных через указатели деструктор вызывается неявно при использовании операции delete.

Деструктор:

- не имеет аргументов и возвращаемого значения;
- не может быть объявлен как const или static;
- не наследуется;
- может быть виртуальным.

Вернемся к примеру 2, где был создан class Counter. При решении разных задач может возникнуть необходимость инициализации счетчика разными значениями, а не только нулем. Для этого создадим еще один конструктор.

```
class Counter
{  private:    int  mycount;
    public:
    Counter ( ):mycount(0){ }
        Counter (int c): mycount(c){ }
    void inc_count ( ) {mycount++; }
    int   get_count( ) { return mycount; }
        ~Counter( ){} };
int _tmain(int argc, _TCHAR* argv[])
{  Counter c1, c2(3);
  std::cout<<c1.get_count()<<std::endl;
  std::cout<<c2.get_count()<<std::endl;
  c1.inc_count();    c2.inc_count();
  std::cout<<c1.get_count()<<std::endl;
  std::cout<<c2.get_count()<<std::endl;      return 0;}
```

Наследование

Наследование – это процесс создания новых классов, называемых наследниками или производными классами, из уже существующих или базовых классов. Производный класс получает все возможности базового класса, но может быть усовершенствован за счет добавления собственных.

Наследование – важная часть ООП. Выигрыш от него состоит в том, что **наследование позволяет использовать существующий код несколько раз.**

Базовый класс
Свойство А
Свойство Б
Свойство В
Стрелка означает наследование
Производный класс
Свойство Г
Определено в производном классе
Свойство А
Свойство Б
Свойство В
Определены в базовом, но доступны в производном

Простым называется наследование, при котором производный класс имеет одного родителя.

Формат описания производного класса:

В заголовке производного класса через двоеточие (:) указывается имя родительского класса.

```
class имя_класса :
  <спецификатор доступа> имя_базоваго_класса
{
   объявление элементов класса
};
```

При описании производного класса надо помнить, что **поля и методы** родительского класса могут быть унаследованы!!!

Рассмотрим правила наследования различных методов:

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы.

Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса по умолчанию.

Деструкторы также не наследуются, и если программист не описал в производном классе деструктор, он формируется по умолчанию.

Остальные методы могут наследоваться или переопределяться.

Поля, унаследованные из родительского класса, и которые определены в базовом классе как private, недоступны функциям производного класса.

Если они должны использоваться функциями, определенными в производном классе, можно либо описать их в базовом классе как **protected**, либо явно переопределить их в производном классе.

Вернемся к примеру счетчиков. Предположим, что нам понадобился метод, уменьшающий значение счетчика. Разработаем новый класс CountDown.

```
class Counter
{  protected:    int  mycount;
    public:
    Counter ( ):mycount(0){ }   //конструктор 1
        Counter(int c): mycount(c){ } //конструктор 2
     void inc_count ( ) {mycount++; }
  int   get_count( ) { return mycount; }
};
class CountDown: public Counter
{   public:
                void dec_count ( )
                        {mycount--; }
  };
```

Новый класс наследует: поле mycount и методы get_count( ) и inc_count().

```
int main(int argc, char *argv[])
{  Counter c1, c2(3);
   CountDown  c3;
   std::cout<<c1.get_count()<<std::endl;
   std::cout<<c2.get_count()<<std::endl;
   std::cout<<c3.get_count()<<std::endl;
   c1.inc_count();    c2.inc_count();
   c3.dec_count();
   std::cout<<c1.get_count()<<std::endl;
   std::cout<<c2.get_count()<<std::endl;
   std::cout<<c3.get_count()<<std::endl;
  return 0;
}
```

Для объекта c3 доступен новый метод: **c3.dec_count();**
Но можно применять и унаследованные методы, например: **std::cout<<c3.get_count()<<std::endl;**
Можно добавить в программу еще две строчки:

```
c3.inc_count();
std::cout<<"!!!"<<c3.get_count()<<std::endl;
```

Тогда значение счетчика c3 увеличится!
Переменная c3 – это объект класса CountDown. В классе CountDown нет конструктора. Если в производном классе не определен конструктор, то используется конструктор базового класса без параметров.
Но мы не сможем воспользоваться конструктором с параметром из базового класса.

Поэтому, если возникает необходимость инициализации объекта c3 каким-либо другим значением, мы должны написать новый конструктор:

```
class CountDown: public Counter //определение класса
{   public:
    CountDown ( ):Counter(0){ }
     CountDown(int c): Counter(c){ } //конструктор2
    void dec_count ( ) {mycount--; }
 };
```

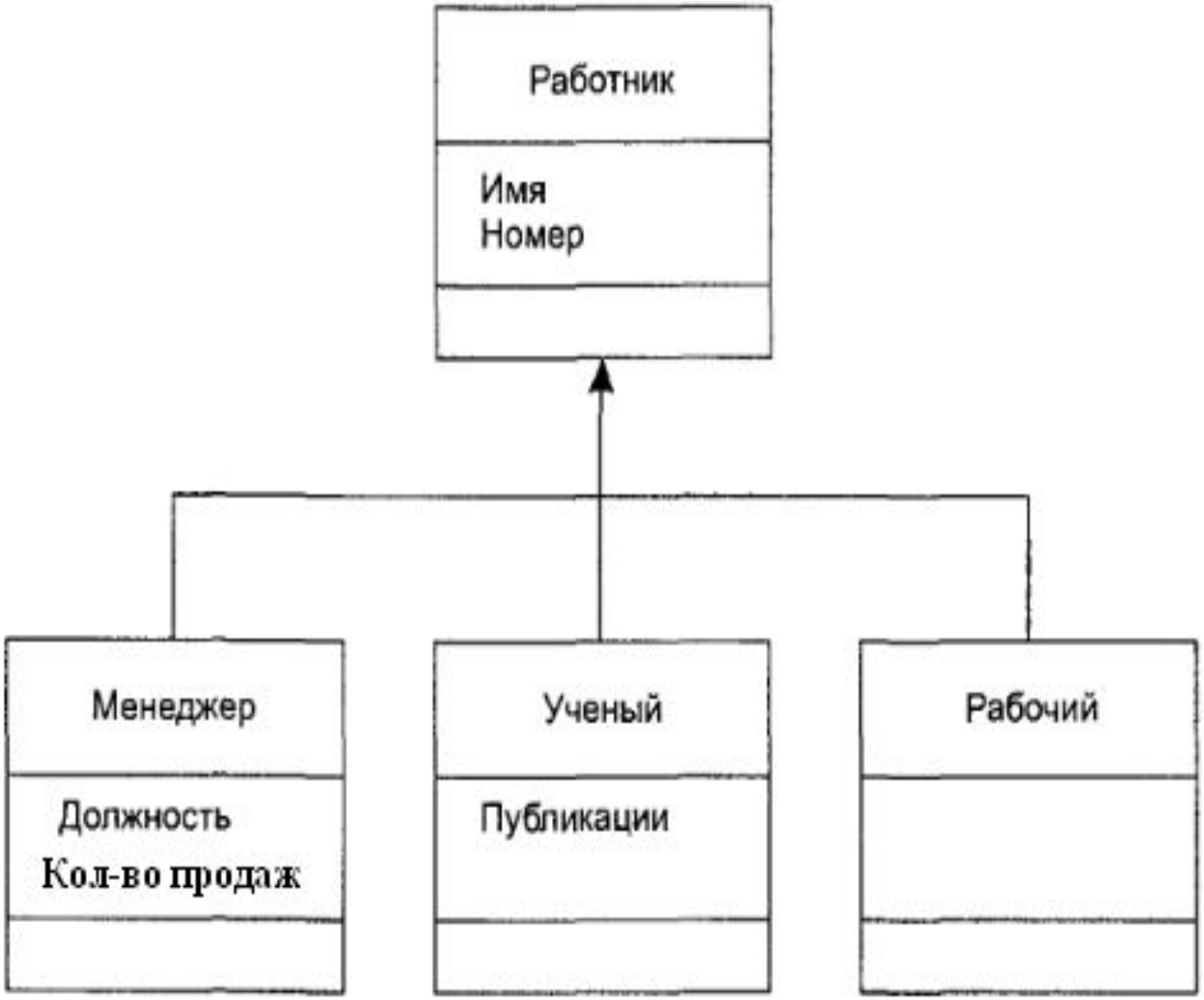
Так как действия в конструкторах базового и производного классов совпадают, то мы подключаем вызов конструкторов базового класса, для выполнения нужных действий.

Иерархия классов

На основе принципа наследования может быть построена иерархия классов.

Рассмотрим пример базы данных служащих некоторой компании. В ней существует три категории служащих: менеджеры, занимающиеся продажами, ученые, занимающиеся исследованиями и рабочие, занятые изготовлением товаров.

Иерархия будет состоять из базового типа: **employee** и трех производных классов: **manager, scientist** и **laborer**.


Работник
Имя
Номер
Менеджер
Должность
Кол-во продаж
Ученый
Публикации
Рабочий

```
#include <iostream>
using namespace std;
const int len=80;
class employee
{  protected:
      int nom;
      char name[len];
   public:
     void getdata()
     { cout<<"vvod N: ";     cin>>nom;
       cout<<"vvod FIO: “;  cin>>name; }
       void putdata()
       {cout<<" N: " <<nom;  cout<<"\n FIO:
"<<name;
       }       };
```

```
class manager:public employee
{  private:
          char title[len];
          int kol;
     public:
     void getdata()
     { employee::getdata();
       cout<<"vvod dolgnosty: ";  cin>>title;
      cout<<"vvod kolvo: ";         cin>>kol;
      }
      void putdata()
      { employee::putdata();
      cout<<"\n dolgnosty: "<<title;
      cout<<"\n kol-vo prodag: " <<kol;
      }    };
```

```
class scientist:public employee
{  private:
        int pubs;
    public:
    void getdata()
    { employee::getdata();
      cout<<"vvod kolva pubs: ";  cin>>pubs;
      }
      void putdata()
      { employee::putdata();
      cout<<"\n publication: "<<pubs;
      }
      };
class laborer:public employee
{ };
```

```
int main(int argc, char *argv[])
{  employee x;     manager y;
   scientist z;       laborer w;
   cout<<"vvod svedenij o 4 sotrudnikax:\n";
  x.getdata();
  y.getdata();
  z.getdata();
  w.getdata();
 cout<<"vivod  information about sotrudnikax:\n";
  x.putdata();
  y.putdata();
  z.putdata();
  w.putdata();
  return 0;
}
```

Производный класс может являться базовым для других производных классов.

Например:

```
class A
{  … };
class B: public A
{ …};
class C:public B
{…};
```

Здесь класс B является производным класса A, а класс C производным класса B.

Класс может являться производным как одного базового класса, так и нескольких базовых классов (**множественное наследование**).

Например:

```
class A
{  … };
class B
{ …};
class C: public A, public B
{…};
```

Базовые классы перечисляются через запятую после знака **:**

Работа с объектами чаще всего производится через указатели, например:

employee *p;

Указателю на базовый класс можно присвоить значение адреса объекта любого производного класса:

p = new laborer; или p=&y;

Где y описана, как: manager y;

Обращение к методу через указатель имеет вид:

p->getdata();

p->putdata();

Виртуальные методы. Полиморфизм.

Полиморфизм — один из важнейших механизмов ООП. Полиморфизм реализуется с помощью наследования классов и виртуальных методов.

Полиморфизм состоит в том, что с помощью одного и того же обращения к методу выполняются различные действия в зависимости от типа, на который ссылается указатель в каждый момент времени.

Рассмотрим пример иерархии классов, где каждый класс имеет метод с одним именем.

```
class Base
{
    public:
      void show()
          { cout<<“Родитель\n";
          }
};
class derv1:public Base
{public:
          void show()
          { cout<<"Сын первый\n";
          }
};
```

```
class derv2:public Base
{public:
        void show()
        { cout<<"Сын второй\n";
        }
};
int main(int argc, char *argv[])
{
   derv1   s1;        derv2    s2;
   Base *ptr;
   ptr=&s1;
   ptr->show();
   ptr=&s2;
   ptr->show();
```

…

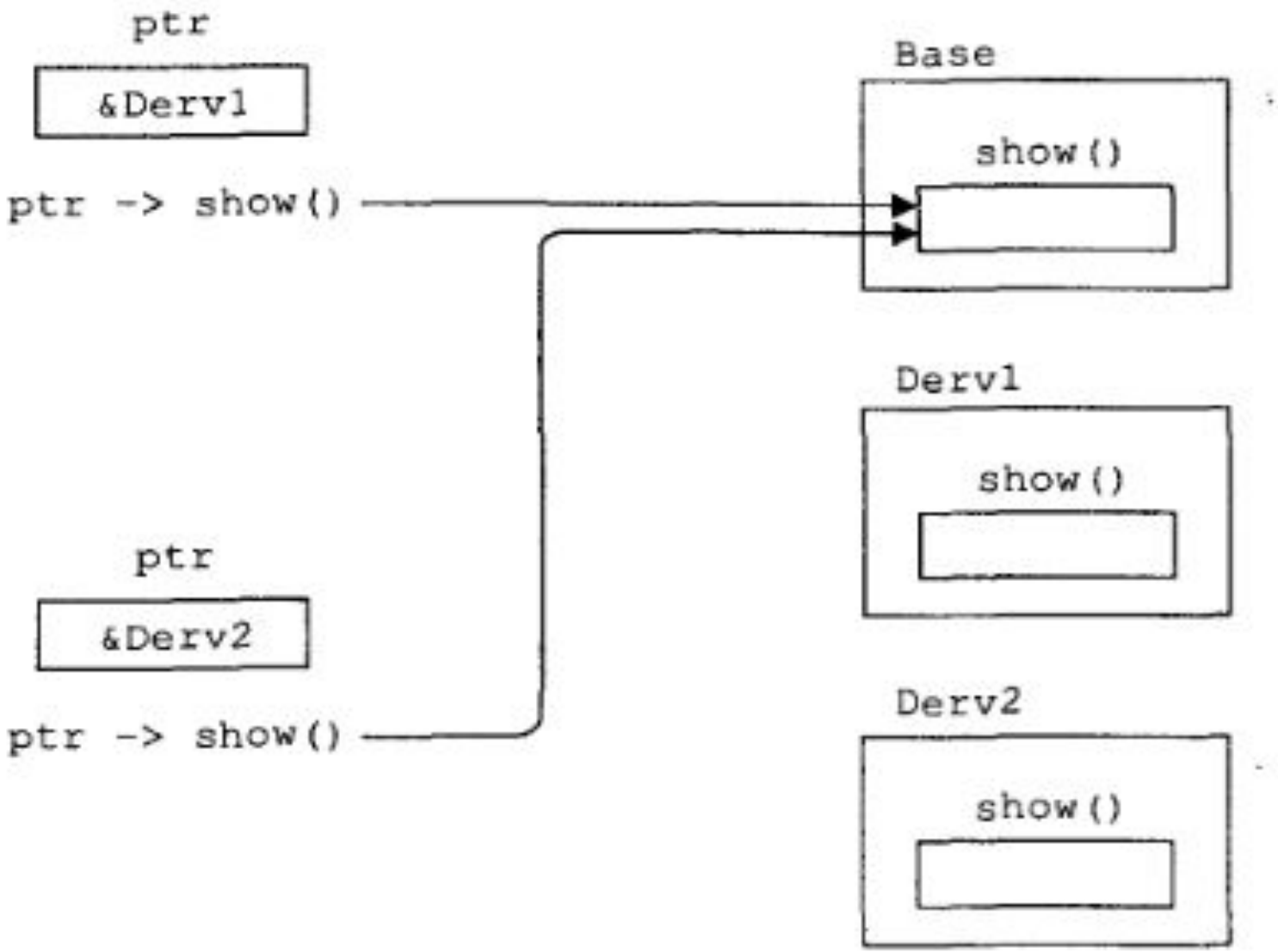
Итак, классы derv1 и derv2 являются наследниками класса Base. В каждом из трех классов имеется метод **show().** В main() созданы объекты порожденных классов s1 и s2 и указатель на класс Base. Затем адрес объекта порожденного класса мы заносим в указатель базового класса: **ptr=&s1;**

Какая же функция будет выполняться в следующей строке:

ptr->show(); Base::show( ) или derv1::show( )?

В этом случае компилятор выбирает метод, удовлетворяющий типу указателя (Base::show())

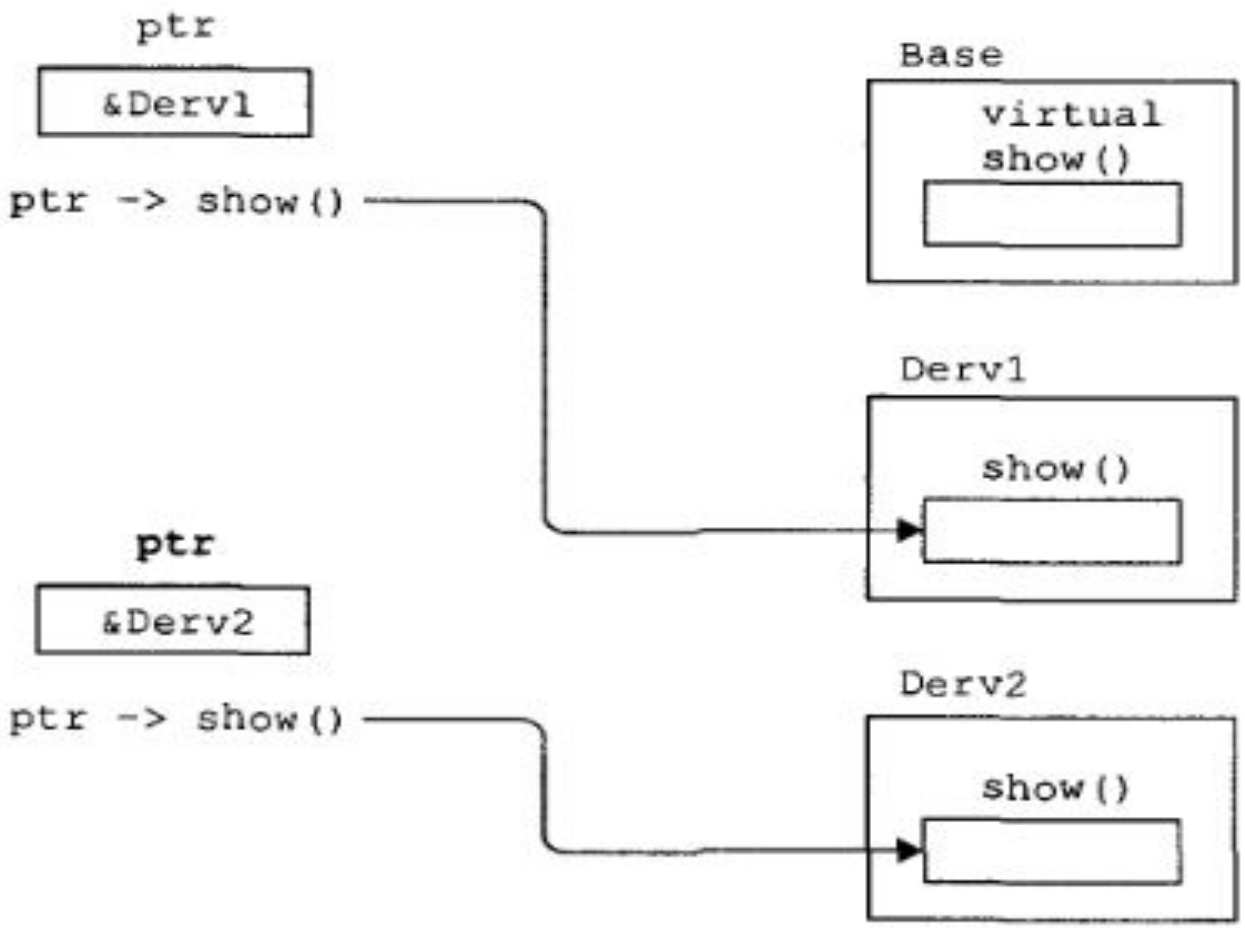
Этот процесс ,называется ***ранним связыванием***.



В C++ реализован механизм *позднего связывания, когда разрешение ссылок на* метод происходит на этапе выполнения программы в зависимости от конкретного типа объекта, вызвавшего метод. Этот механизм реализован с помощью виртуальных методов.

Для определения *виртуального метода* используется спецификатор ***virtual***, например:

```
class base
{
    public:
        virtual void show()
         { cout<<"base\n";
         }
};
```



Этот процесс ,называется ***поздним связыванием.***

Если в базовом классе метод определен как виртуальный, метод, определенный в производном классе с ***тем же именем и набором параметров***, автоматически становится виртуальным, а с отличающимся набором параметров — обычным.

Для каждого класса (не объекта!), содержащего хотя бы один виртуальный метод, компилятор создает ***таблицу виртуальных методов (vtbl)***, в которой для каждого виртуального метода записан его адрес в памяти.

Рекомендуется делать виртуальными деструкторы для того, чтобы гарантировать правильное освобождение памяти из-под динамического объекта, поскольку в этом случае в любой момент времени будет выбран деструктор, соответствующий фактическому типу объекта. Деструктор передает операции **delete** размер объекта.

Обзор стандартных библиотек C++

Развитие объектно - ориентированного программирования привело к созданию широкого набора библиотек.

Библиотека STL/CLR представляет собой упакованную библиотеку стандартных шаблонов (STL), входящую в состав стандартной библиотеки C++.

Библиотека содержит пять основных видов компонентов:

алгоритм (*algorithm*): определяет вычислительную процедуру.

контейнер (*container*): управляет набором объектов в памяти.

итератор (*iterator*): обеспечивает для алгоритма средство доступа к содержимому контейнера.

функциональный объект (*function object*): инкапсулирует функцию в объекте для использования другими компонентами.

адаптер (*adaptor*): адаптирует компонент для обеспечения различного интерфейса.

Алгоритмы реализуют большинство методов сортировки, поиска и т.п. (<algorithm> <functional>)

Контейнеры – это объекты, содержащие другие однотипные объекты. В контейнерных классах реализованы такие типовые структуры, как стеки <stack>, списки<list> , очереди <queue> и т.д.

Итераторы – это обобщение концепции указателей: они ссылаются на элементы контейнера <iterator>.

Рассмотрим пример, реализующий работу стека с помощью стандартной библиотеки шаблонов.

Рассмотрим основные методы класса stack:

push - метод для добавления элемента в стек.

pop – удаляет элемент из стека, но не возвращает удаленное значение.

top – возвращает значение с вершины стека.

empty – проверяет наличие элементов в стеке.

```
#include <stdafx.h>
#include <iostream>
#include <stack>
using namespace std;
int main(int argc, char *argv[])
{  stack<int>x;
for(int i=1; i<5; i++)
   x.push(i);                //записать в стек числа
 while(!x.empty())  //если стек не пуст?
{cout<<x.top()<<" ";//вывести число с вершины
x.pop();}                    //удалить число из стека
        return 0;
}
```

Библиотека ATL расшифровывается как Active Template Library. Это библиотека классов и шаблонов, предназначенная для разработки собственных компонетов. Одно из применений этой библиотеки - это создание собственных элементов ActiveX. Например, с помощью библиотеки ATL вы можете создать собственную особую кнопку (скажем, круглую) и затем использовать ее в программах.

Библиотека MFC (Microsoft Foundation Classes) предназначена в основном для создания приложений с пользовательским интерфейсом (окна, диалоги и т. п.).

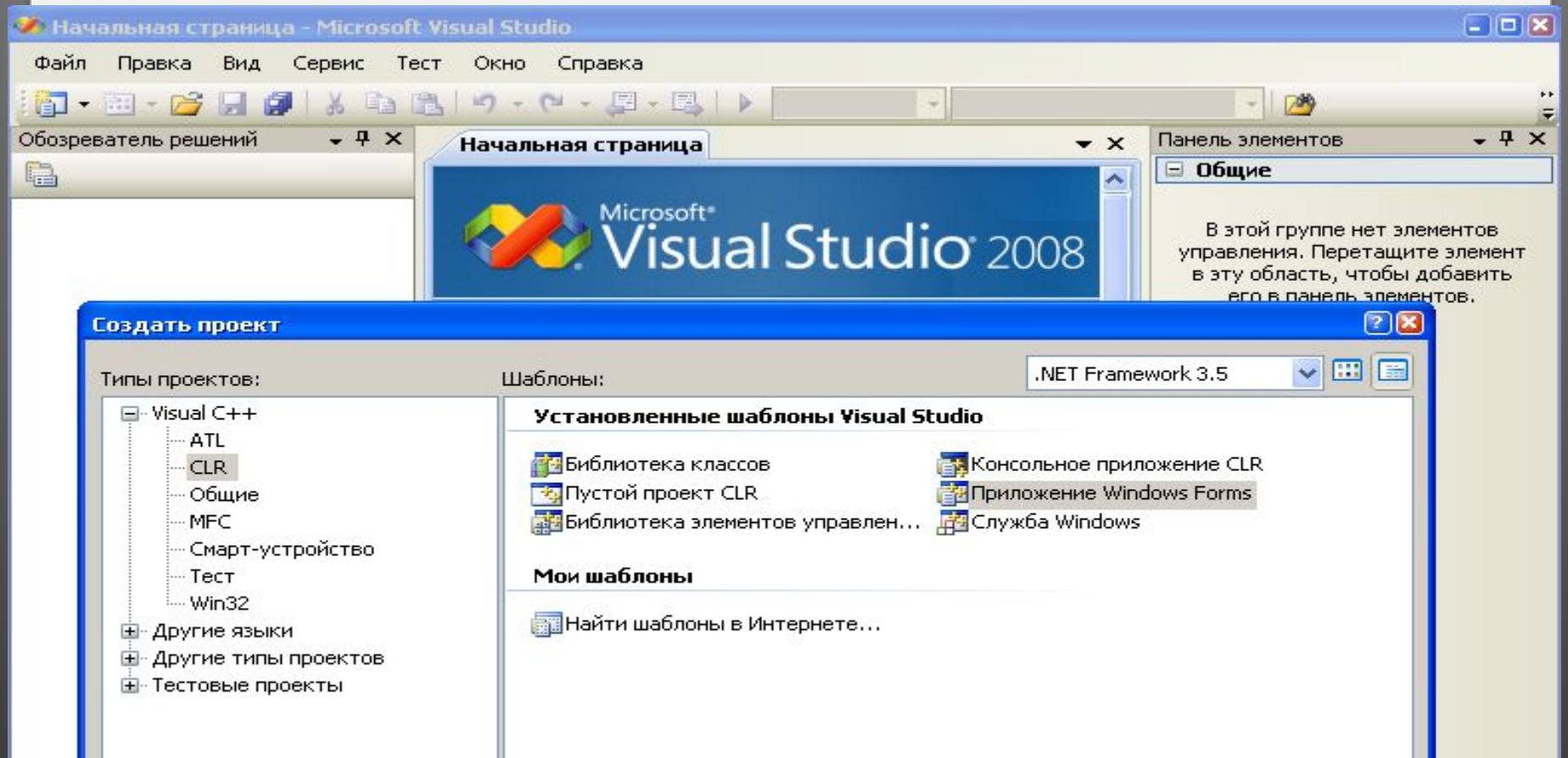
Библиотека MFC инкапсулирует многие функции API (Application Programming Interfase), с помощью которых реализуются все необходимые системные действия, такие как выделение памяти, вывод на экран, создание окон и т.п. Библиотека MFC разрабатывалась для упрощения задач, стоящих перед программистом.

Кроме этого в языке C++ имеются библиотеки для параллельного программирования: Concurrency Runtime для C++ и OpenMP (Open Multi-Processing).

Библиотека классов .NET Framework имеет следующие возможности:

- библиотека базовых классов,такие как строки, массивы и элементы форматирования;
- передача данных по сети;
- система безопасности;
- удаленная обработка;
- диагностика;
- ввод/вывод;
- базы данных;
- язык XML;
- Web-программирование;
- пользовательский интерфейс ОС Windows.

Если мы захотим создать Windows – приложение с помощью технологии .NET, то открыв Visual Studio, и выбрав Файл -> Создать -> Проект, выбираем пункт CLR, отмечаем Приложение Windows Forms и даем имя проекта:



qqq - Microsoft Visual Studio

Файл Правка Вид Проект Построение Отладка Данные Сервис Тест Окно Справка

Debug Win32

Обозреватель решений - qqq

Решение "qqq" (проектов: 1)
- qqq
  - Заголовочные файлы
    - Form1.h
      - Form1.resX
    - resource.h
    - stdafx.h
  - Файлы исходного кода
    - AssemblyInfo.cpp
    - qqq.cpp
    - stdafx.cpp
  - Файлы ресурсов
    - app.ico
    - app.rc
  - ReadMe.txt

Обозр... Окно ... Диспе...

Form1.h [Конструктор] Начальная страница

Form1

Панель элементов

- Label
- LinkLabel
- ListBox
- ListView
- MaskedTextBox
- MonthCalendar
- NotifyIcon
- NumericUpDown
- PictureBox
- ProgressBar
- RadioButton
- RichTextBox
- TextBox
- ToolTip
- TreeView
- WebBrowser
- **Контейнеры**
- **Меню и панели инструме...**

Обозревате... Панель элем...

Свойства

qqq Свойства проекта

СПАСИБО за ВНИМАНИЕ

FIN