# Глава 5.
# **Объектная модель C++**

МГТУ им. Н.Э. Баумана
Факультет Информатика и системы управления
Кафедра Компьютерные системы и сети
Лектор: д.т.н., проф.
Иванова Галина Сергеевна

# 5.1 Описание класса и создание объектов

Формат описания класса:

```
class <Имя класса>
{  private:     <Внутренние компоненты класса>;
   protected:  <Защищенные компоненты класса>;
   public:      <Общедоступные компоненты класса>;
};
```

***Внутренние*** компоненты класса – поля и методы – доступны только методам класса (В Delphi Pascal – в пределах модуля).

***Защищенные*** – доступны методам своего класса и методам классов-наследников.

***Общедоступные*** – доступны в пределах видимости, в том числе из программы и методов других классов.

# Пример 5.1. Класс Книга

Диаграмма классов

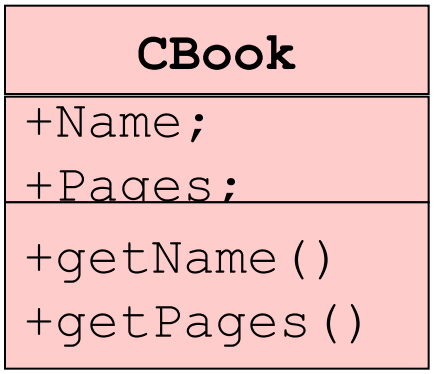
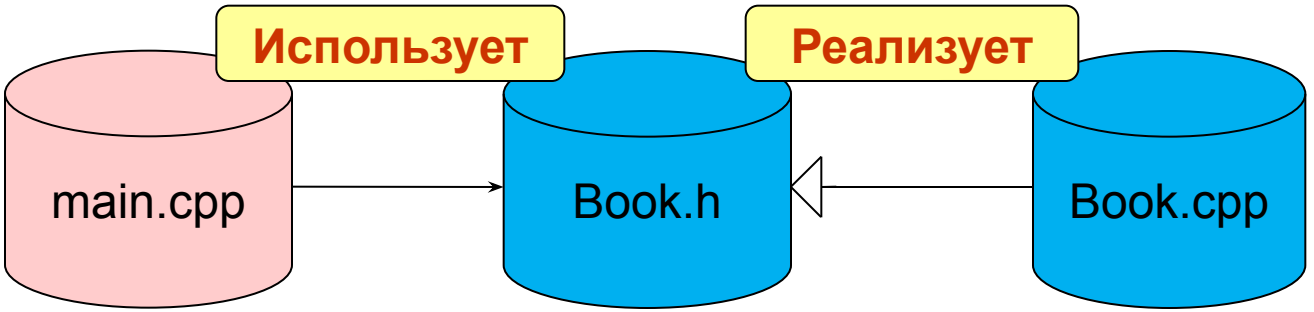



Диаграмма компоновки




Основная программа

Объявление класса и описание inline методов

Описание методов класса, не описанных при объявлении класса

# Объявление класса

Файл Book.h:

```
#ifndef Book_h
#define Book_h

class CBook
{
 public:  char Name[30];
   int Pages;
   char *getName() // метод по умолчанию считается inline,
   {               // так как его тело описано в классе
     return Name;
   }
   int getPages(); // тело метода будет описано в book.cpp
};
#endif
```

Защита от повторной компиляции описания класса (начало)

Защита от повторной компиляции описания класса (завершение)

# Описание методов в специальном файле

```
// Файл Book.cpp – секция реализации модуля book

#include "book.h"
// тело метода
int CBook::getPages()
{
   return Pages;
}
```

В отличие от методов, описанных в классе, методы, описанные в файле реализации модуля, не являются по умолчанию объявленными **inline**. При необходимости директива **inline** указывается явно при объявлении метода в классе.

# Объявление неинициализированных объектов при отсутствии в классе конструктора

Объявление объектов:

**<Имя класса> <Список переменных и/или указателей>;**

**Пример:**

```
#include "book.h"

int main()
{
  CBook a,   // простая переменная-объект
       *b,   // указатель на объект (память под объект
             не выделена)
       c[5]; // массив из 5 объектов в статической памяти
    …
```

# Инициализация общедоступных полей объектов при объявлении в случае отсутствия в класса конструктора

Для инициализации общедоступных полей объектов может быть использована та же конструкция, что и для инициализации полей структуры.

**Пример:**

```
#include "book.h"
```

Необязательно!

```
int main()
{ CBook A = {"J.London. V.1",366};
  CBook C[]= {{"J.London. V.3",367},
        {"J.London. V.4",321},
        {"J.London. V.5",356}};
... }
```

Память будет выделена под массив из трех объектов

# Обращение к полям и методам класса

а) простой объект:

**<Имя объекта>.<Имя поля или метода>**

б) указатель на объект:

**<Имя указателя на объект> -><Имя поля или метода>**

в) массив объектов

**<Имя массива>[<Индекс>] .<Имя поля или метода>**

Внутренний указатель на поля объекта:

**Указатель** `this` **(C++)** ⇔ `Self` **(Паскаль)** **Ссылка**

**Пример:** `this->Pos`

# Основная программа

```
#include "book.h"
#include <stdio.h>
int main()
{
   // объявление инициализированного объекта
   CBook B={"J.London. Smoke Bellew",267};
   printf("%s %d\n",B.getName(),B.getPages());
   // объявление массива инициализированных объектов
   CBook C[]= {{"J.London. V.3",367},
         {"J.London. V.4",321},
         {"J.London. V.5",356}};
   for(int i=0;i<3;i++)
       printf("%s %d\n",C[i].getName(),C[i].getPages());
   return 0;
}
```

# 5.2 Конструкторы и деструкторы

**Конструктор** – метод, автоматически вызываемый при выделении памяти под объект. Используется для инициализации полей объекта. Автоматический вызов страхует программиста от ошибки оставить поля неинициализированными.

```
class CBook
{protected: char Name[30];
        int Pages;
 public:
   CBook(const char *name,int pages) {
         Pages=pages;  strcpy(Name,name);
   }…
```

При создании объектов классов с конструкторами параметры записываются в круглых скобках:

```
int main()
{  CBook D("J.London. Smoke Bellew",267);
```

# Конструкторы без параметров

Конструкторы, как и другие функции, можно перегружать. Специальный конструктор без параметров (инициализирующий или неинициализирующий) используется для создания объектов, которым при выделении памяти не могут быть переданы значения полей.

а) `CBook(){}` // неинициализирующий конструктор без параметров,
// используется для создания неинициализированных объектов

б) `CBook(){ Name[0]='\0'; Pages=0; }` // инициализирующий
// конструктор без параметров, создает одинаково
// инициализированные объекты

При создании объектов посредством конструкторов без параметров круглые скобки не указывают:

```
void main()
{  CBook J; …
```

# Делегирующие конструкторы

Начиная с C++11, одни конструкторы класса (делегирующие) могут вызывать другие, объявленные в том же классе:

```
class CBook
{protected: char Name[30];
      int Pages;
 public:
  CBook(const char *name,int pages) {
        Pages=pages;  strcpy(Name,name);
    }
  CBook():CBook("No name",0){}
 };
```

Это позволяет избежать дублирования кода и связанных с ним ошибок.

# Автоматически генерируемые методы. default

Начиная с C++11 для класса автоматически генерируются следующие методы:

- конструктор по умолчанию: **<Имя класса>(){}**
- копирующий конструктор;

  **<Имя класса>(<Имя класса>& <Объект>){}**
- конструктор перемещения

  **<Имя класса>(<Имя класса>&& <Объект r-value>){…}**
- оператор присваивания

  **<Имя класса>& operator=(<Имя класса>& <Объект>){…}**
- оператор перемещения

  **<Имя класса>& operator=(<Имя класса>&&**
  **<Объект r-value >){…}**
- деструктор **~<Имя класса>(){}**

Любой из них можно явно попросить сгенерировать автоматически, указав default, например:

**<Имя класса>()= default;**

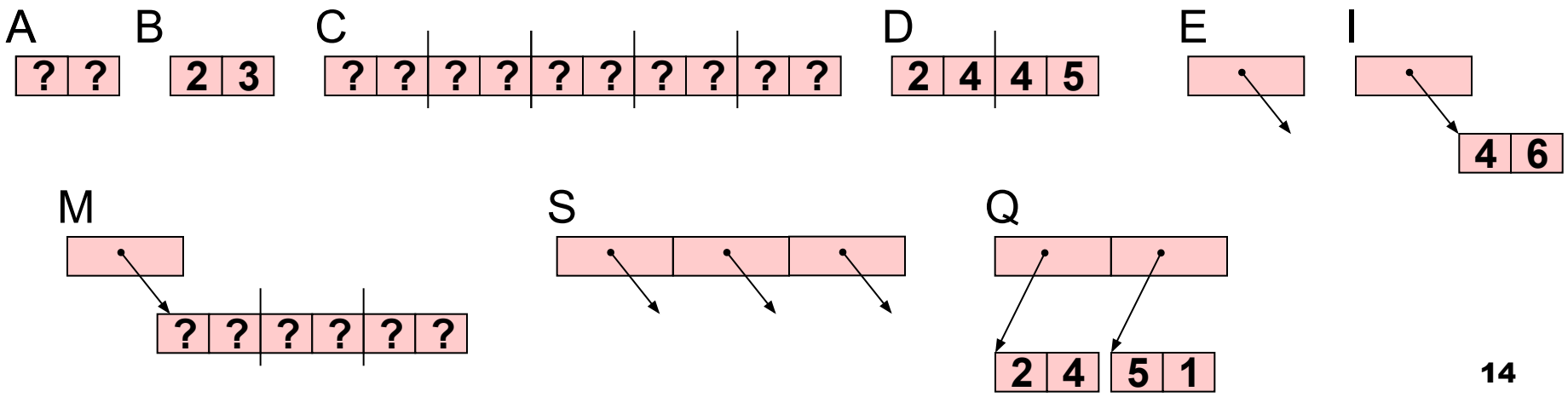
# Различные способы создания объектов

**Пример (Ex5_4):**

```
class CPoint
{ private: int x,y;
  public: CPoint(int ax,int ay){x=ax;y=ay;}
          CPoint(){}
          void SetPoint(int ax,int ay) {x=ax;y=ay;}   ...};
int main()
{ CPoint A, B(2,3), C[5],  D[2] = {CPoint(2,4),CPoint(4,5)},
    *E,*I = new CPoint(4,6), *M = new CPoint[3],
    *S[3], *Q[]={new CPoint(2,4),new CPoint(5,1)};
```

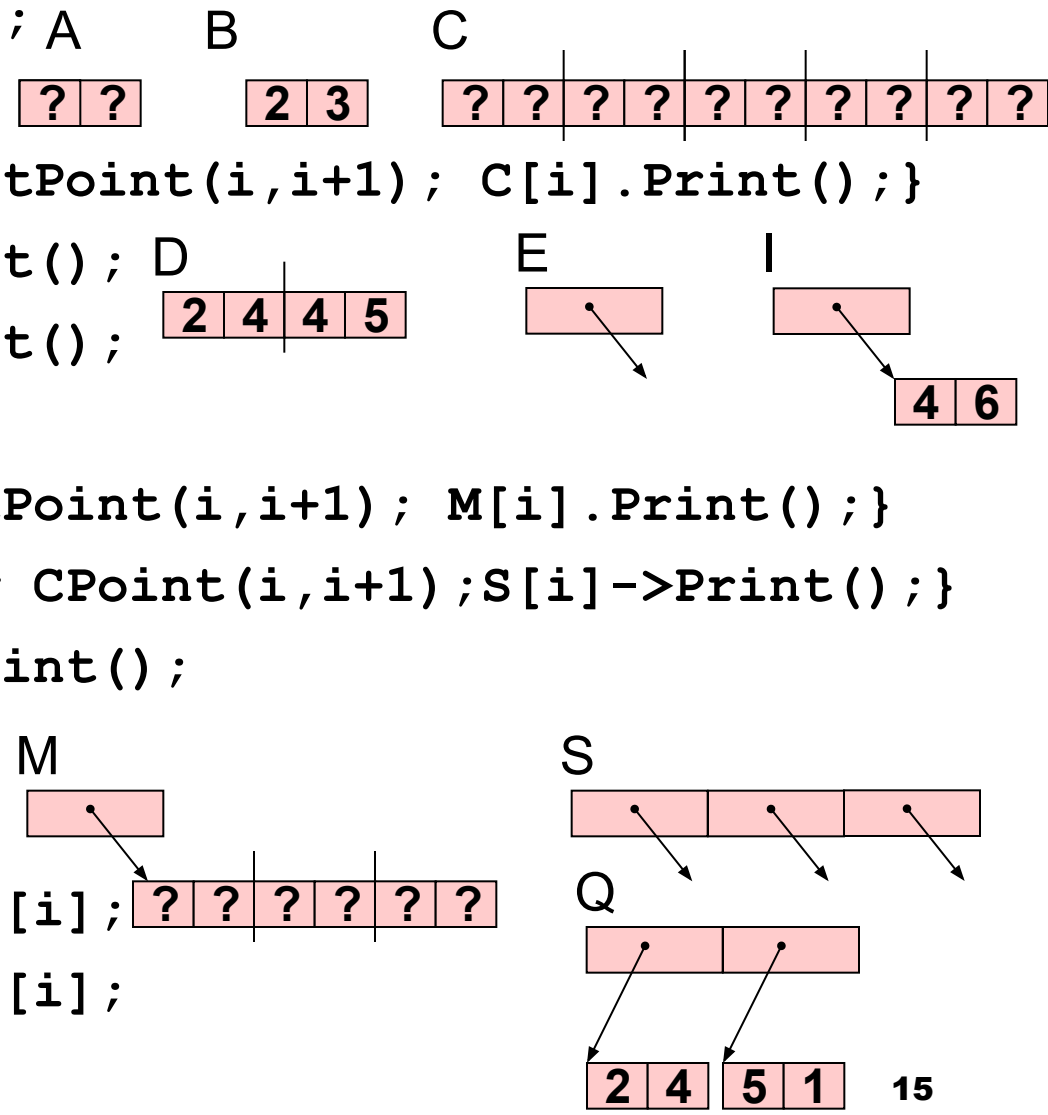
Можно {2,3}

Можно {{2,4},{4,5}},



# Распределение/освобождение памяти и инициализация объектов в программе

```
A.SetPoint(2,3); A.Print();
B.Print();
for (i=0;i<5;i++) {C[i].SetPoint(i,i+1); C[i].Print();}
for(i=0;i<2;i++) D[i].Print();
E=new CPoint(3,4); E->Print();
I->Print();
for (i=0;i<3;i++){M[i].SetPoint(i,i+1); M[i].Print();}
for (i=0;i<3;i++){S[i]=new CPoint(i,i+1);S[i]->Print();}
for (i=0;i<2;i++) Q[i]->Print();
delete E; delete I;
delete [] M;
for (i=0;i<3;i++) delete S[i];
for (i=0;i<2;i++) delete Q[i];
return 0; }
```



# Список инициализации. Инициализация объектных полей

Формат элемента списка инициализации:
**<Имя поля>(<Список выражений>)**

**Примеры:**

а) `TPoint(int ax,ay): x(ax),y(ay) {}`

б) без списка инициализации нельзя инициализировать:

```
class TLine
{ private:
      const int x; // константные поля
      int &y;      // ссылочные поля
      TPoint t;    // объектные поля
  public: TLine(int ax,int ay,int tx,int ty):
                x(ax),y(ay),t(tx,ty){}
          TLine(){}
  ...};
```

**Автоматически вызывает конструктор объектного поля без параметров TPoint( ) !**

# Объекты с динамическими полями. Деструкторы

**Деструкторы** аналогично конструкторам вызываются автоматически, но в момент освобождения памяти, выделенной под объект. Деструкторы обычно используют для освобождения памяти, выделенной под динамические поля объектов.

```
class CBook
{  char *pName; int Pages;
public:
   CBook(const char *name,int pages){
        pName=new char[30];
        strcpy(pName,name);
        Pages=pages;
   }
   ~CBook(){
        delete [] pName;
   }
};
```

# Копирующий конструктор

Автоматически вызывается:

а) при использовании объявлений типа

```
TPoint A(2,5),B=A;
```

б) при передаче параметров-объектов по значению, например:

```
void Print(TPoint R) {…}
```

Формат:

```
<Имя конструктора>(const <Имя класса> &<Имя объекта>){…}
```

**Примеры:**

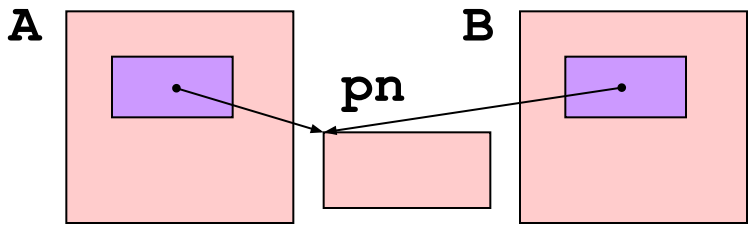
а)
```
TPoint(const TPoint &Obj)
    {x=Obj.x; y=Obj.y;}
```

б)
```
TPoint(const TPoint &Obj)
    {x=Obj.x; y=2*Obj.y;}
```

**Строится автоматически**

**Любой можем объявить сами**

# Пример обязательного определения копирующего конструктора (Ex5_05)




```
#include <stdio.h>
class TNum
{ public:int *pn;
   TNum(int n){puts("new pn"); pn=new int(n);}
   TNum(const TNum &Obj)
        {puts("copy new pn"); pn=new int(*Obj.pn);}
   ~TNum(){puts("delete pn");delete pn;}
};

void Print(TNum b) { printf("%d ",*b.pn); }

int main() {
  TNum A(1);
  Print(A);
  return 0;
}
```

# Дескриптор `explicit`

Дескриптор указывается, когда необходимо *точное соответствие* формальных и фактических параметров при вызове конструктора.

**Пример.**

```
class abc{
  explicit abc(int a){…}
};
```

unsigned char

```
abc A = 'a';   // при наличии дескриптора explicit ошибка!
               // Если бы его бы не было, то ошибка не
               // была бы обнаружена
```

P.S. Операция выполняется с помощью копирующего конструктора

**Но, к сожалению:**

```
abc A1('a');  или abc A2{'a'}; // Ошибки нет!
```

# Запрет использования конструктора delete

Дескриптор указывается, когда необходимо *запретить использование конвертации типов* при вызове конструктора.

**Пример.**

```
class abc{
  explicit abc(int a){…}
  abc(char)= delete;
};
```

unsigned char

```
abc A = 'a';   // при наличии дескриптора explicit ошибка!
               // Если бы его бы не было, то ошибка не
               // была бы обнаружена
```

Теперь:

```
abc A1('a');  или abc A2{'a'}; // Ошибка!
```

Позволяет запретить также использование конструктора копирования и перегруженного оператора!!!

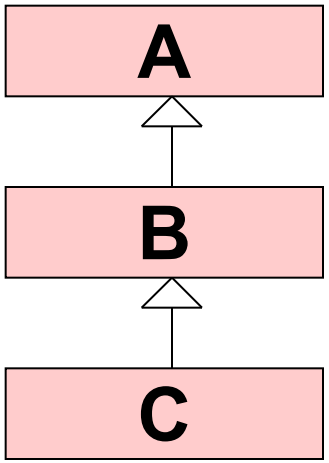
# 5.3 Наследование

Если указано, то запрещает дальнейшее наследование

```
class <Имя производного класса> [final]:
        <Вид наследования> <Имя базового класса>{…};
```

| Вид наследования | Объявление компонента в базовом классе | Видимость компонента в производном классе |
|---|---|---|
| private | private<br>protected<br>public | не видимы<br>private<br>private |
| protected | private<br>protected<br>public | не видимы<br>protected<br>protected |
| public | private<br>protected<br>public | не видимы<br>protected<br>public |

# Конструкторы и деструкторы производных классов




**A() : [<Конструкторы полей>] {…}**

**B() :A() ,<Конструкторы полей>{ }**

**C() :B() ,<Конструкторы полей>{ }**

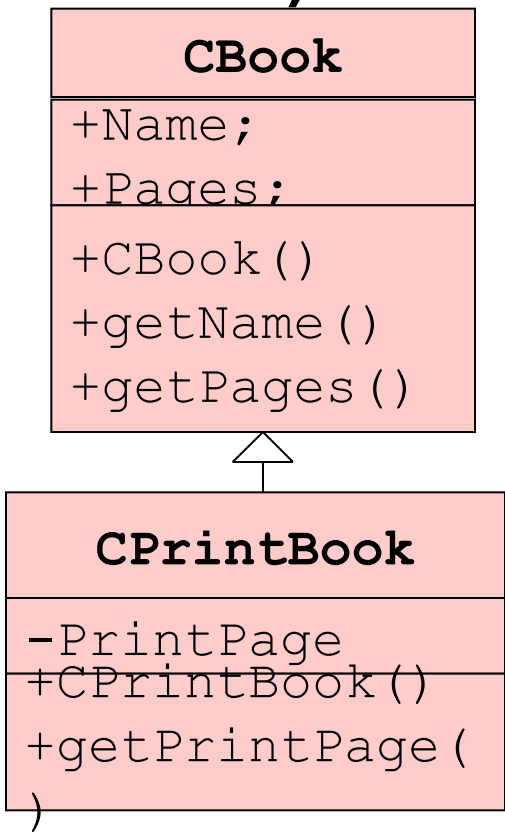
При объявлении объектов производного класса всегда вызывается конструктор базового класса, используемый для инициализации наследуемых полей.

Если в списке инициализации конструктора производного класса вызов конструктора базового класса есть, то вызывается он.

Если в списке инициализации конструктора производного класса вызов конструктора базового отсутствует, то автоматически вызывается конструктор базового класса без параметров!

# Пример наследования Ex5_02 (PrintBook.h)

```
#ifndef printbook_h
#define printbook_h
#include "Book.h"
class CPrintBook:public CBook{
 private:int PrintPages;
 public:
   CPrintBook(const char *name,int pages):
          CBook(name,pages){
              PrintPages=Pages/16;
   }
   int getPrintPages(){
       return PrintPages;
   }
};
#endif
```




**Инициализация наследуемых полей базового класса в списке инициализации**

# Вызов конструкторов и деструкторов для объектов производных классов (Ex5_06)

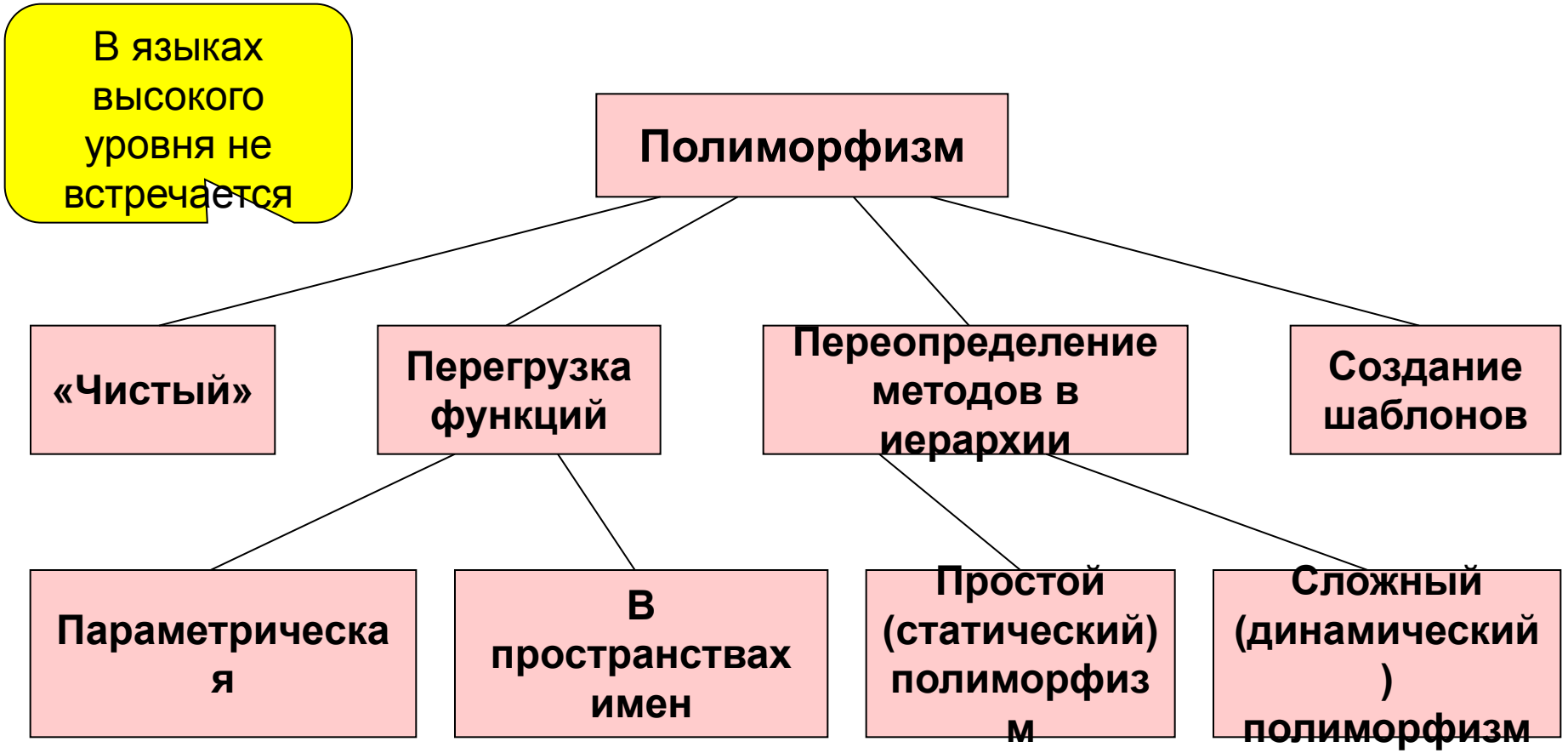
```
#include <stdio.h>
class TNum
   { public:   int n;
        TNum(int an):n(an) {puts("TNum(an)");}
        TNum() {puts("TNum()");}
        ~TNum(){puts("~TNum");}
   };
class TNum2:public TNum
   { public:   int nn;
        TNum2(int an):nn(an) {puts("TNum2(an)");}
        ~TNum2(){puts("~TNum2");}
   };

int main()
{  TNum2 A(1);
   return 0;
}
```

Неявный вызов конструктора TNum()

TNum()
TNum2(an)
~TNum2
~TNum

# 5.4 Полиморфизм. Полиморфное наследование

***Полиморфизм*** – «многоформие», т.е. свойство изменения формы. В программировании встречаются следующие виды полиморфизма:



# Полиморфное наследование

**В Паскале:**

**простой полиморфизм**

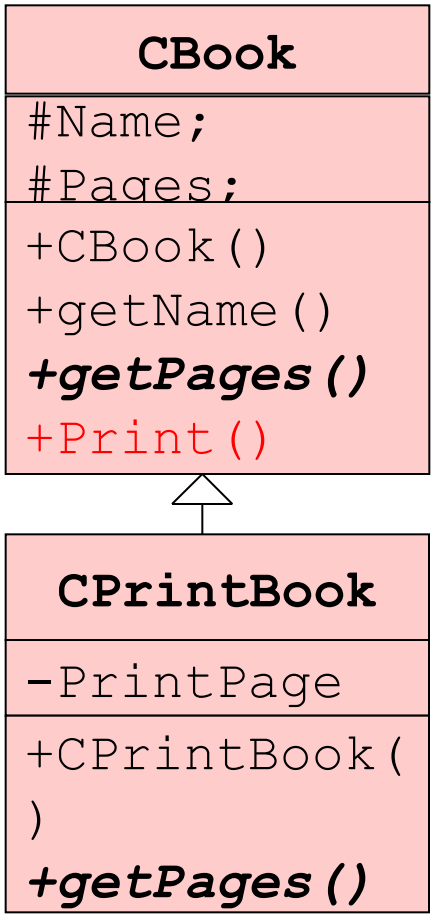
**сложный полиморфизм**

**В C++:**

**переопределение методов**

**виртуализация методов**

**Пример использования сложного полиморфизма (Ex5_03):**




Переопределяемый и переопределяющий методы

Наследуемый метод, вызывающий переопределяемый

# Объявление класса CBook (файл Book.h)

```
#ifndef book_h
#define book_h
#include <string.h>
#include <stdio.h>
class CBook
{ protected: char Name[30];int Pages;
public:
   CBook(const char *name,int pages):Pages(pages){
        strcpy(Name,name);
   }
   char *getName(){ return Name; }
   void Print(){
    printf("%s %d\n",getName(),getPages());
   }
   virtual int getPages();
   virtual ~CBook(){}  //обязателен, если есть виртуальный метод
};
#endif
```

Виртуальный метод нельзя описывать в теле класса

# Класс CBook (файл Book.cpp)

```
#include "Book.h"

   int CBook::getPages() {
        return Pages;
   }
```

# Класс CPrintBook (файл PrintBook.h)

```
#ifndef printbook_h
#define printbook_h
#include "Book.h"

class CPrintBook:public CBook{
private:   int PrintPages;
public:
   CPrintBook(const char *name,int pages):
   CBook(name,pages){
    PrintPages=Pages/16;
   }
   virtual int getPages()override;
};
#endif
```

Объявление Virtual не обязательно!

Указывает, что метод перекрывающий

# Класс CPrintBook (файл PrintBook.cpp)

```
#include "Book.h"

   int CPrintBook::getPages(){
       return PrintPages;
   }
```

# Основная программа

```
#include "PrintBook.h"

int main()
{
   CBook F("J.London. Smoke Bellew",267);
   F.Print();
   CPrintBook D("J.London. Smoke Bellew",267);
   D.Print();

   return 0;
}
```

```
J.London. Smoke Bellew 267
J.London. Smoke Bellew 16
```

# Дескрипторы override и final

Дескрипторы используются для предоставления дополнительной информации компилятору.

Совокупность аспектов виртуального метода или просто *виртуальный метод* – множество совпадающих по прототипу (одноименных методов с одинаковыми списками параметров) методов, объявляемых в иерархии.

Дескриптор **`override`** – используется для обозначения совокупности аспектов виртуального метода в иерархии, начиная со второго аспекта.

Дескриптор **`final`** – при использовании в методе обозначает последний аспект семейства виртуальных методов в иерархии.

# Уточняющие описания override и final

```
class A{
  public:
    virtual void func(){}        // исходные аспекты виртуальных методов
    virtual void f(int){}
    virtual void q()const{}
    virtual ~A(){} };
class B:public A{
  public:
    virtual void f(int) override{}  // переопределяющий метод
    virtual void fumc() override{}
    virtual void f(long) override{}
    virtual void f(int) const override{}
    virtual int f(int) override{}
    virtual void g() const final;  // последний аспект метода
    virtual void g(long);        // новое семейство виртуальных функций
    virtual ~B()override {}
};
```

Использование описателей позволяет компилятору находить ошибки!

Обязательно виртуальный!!!

# Абстрактные методы и классы

Абстрактный метод.

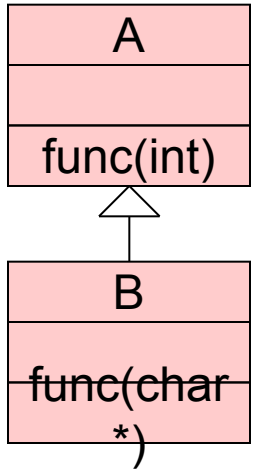
```
class AClass
     { …
        virtual int Fun(int,int)=0;
     }
```

Класс, содержащий абстрактный метод, называется абстрактным.

Объекты абстрактного класса создавать запрещено !

# Использование пространств имен для перегрузки методов класса (Ex5_09)

```
#include <iostream>
class A{
  public:    void func(int ch);
};
class B : public A{
  public:
       void func(const char *str);
       using A::func;     // перегрузка B::func
};
void A::func(int ch){    // метод базового класса
       std::cout << "Symbol\n";
}
void B::func(const char *str){ // метод производного класса
       std::cout << "String\n";
}
int main() {
    B b;
    b.func(25);       // вызов  A::func()
    b.func("ccc");    // вызов  B::func()
    return 0;
}
```



# 5.5 Константные объекты и перегрузка методов для них

C++ разрешает создавать константные объекты, например:

```
<Класс> const a(1); или
const <Класс> a(1);
```

Для константных объектов возможно написание специальных методов, в которых недопустимо изменение полей объекта, например:

```
class A {
private:    int x;
public:
  void f(int a) const {// в метод передается константный объект
    x = a;  // ошибка компиляции !!!
  }
};
```

# Перегрузка методов для константного объекта (Ex5_10)

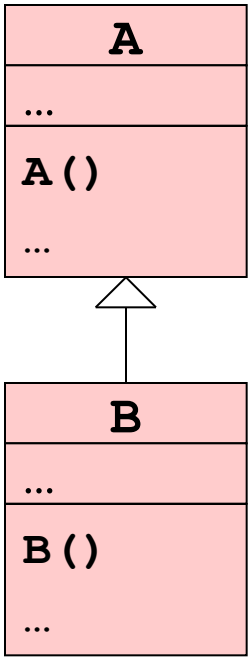
```
#include <iostream>
using namespace std;
class A
{
 private:  int x;
 public:
    A(int a) { x = a; cout << "A(int) // x=" << x << endl;  }
    void f() { cout << "f() // x=" << x << endl;  }
    void f() const { cout << "f() const // x=" << x << endl; }
};
int main(){
  A a1(1);
  a1.f();
  A const a2(2);
  a2.f();
  return 0
}
```

Метод перегружен для константного объекта

```
A(int) // x=1
f() // x=1
A(int) // x=2
f() const // x=2
```

# 5.6 Приведение типов объекта

Различают приведения:
↑- восходящее;
↓- нисходящее.




В C++ для приведения типов используют:

1) **(<Тип>)<Переменная> - используется в Си/C++ для любых типов, ничего не проверяет;**

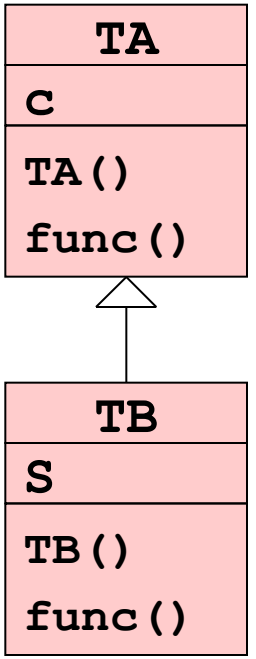
2) **static_cast <Тип>(<Переменная>)- используется в C++ для любых типов, ничего не проверяет;**

3) **reinterpret_cast <Тип указателя> (<Указатель или интегральный тип>)- используется в C++ для указателей, ничего не проверяет;**

4) **dynamic_cast <Тип указателя на объект> (<Указатель на объект>) - используется в C++ для полиморфных классов, требует указания опции компилятора /GR (см. меню Project/Properties/Configuration Properties/ C_C++/Language/Enable Run-Time Type Info = Yes), если приведение невозможно, то возвращает nullptr.**

# Пример приведения типов объектов (Ex5_07)

```
#include <iostream.h>
#include <string.h>

class TA
{ protected:char c;
  public: TA(char ac):c(ac){}
   virtual void func(){cout<<c<<endl;}
};
class TB:public TA
{          char S[10];
  public: TB(char *aS):TA(aS[0]){strcpy(S,aS);}
        void func(){cout<<c<<' '<<S<<endl;}
};
```



# Пример приведения типов объектов(2)

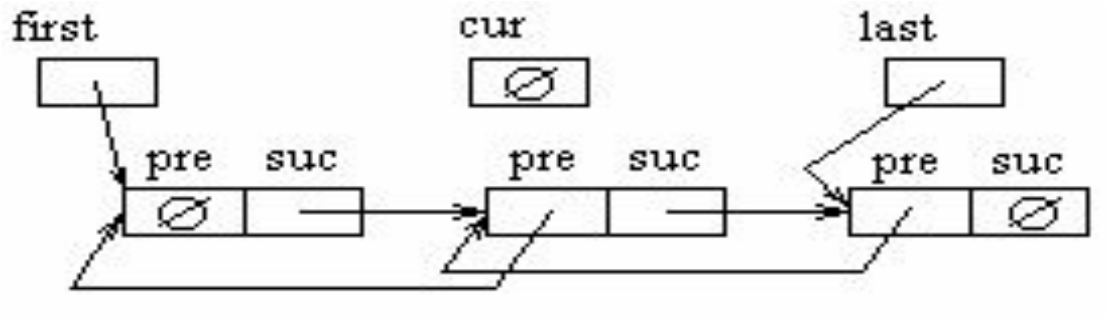
```
int main(int argc, char* argv[])
{ TA *pA=new TA('A'),*pC=new TB("AB");
  TB *pB=new TB("AC");

  ((TA *)pB)->func();
  reinterpret_cast<TA *>(pB)->func();
  static_cast<TA *>(pB)->func();
  dynamic_cast<TA *>(pB)->func();

  ((TB *)pC)->func();
  reinterpret_cast<TB *>(pC)->func();
  static_cast<TB *>(pC)->func();
  dynamic_cast<TB *>(pC)->func();

  ((TB *)pA)->func();
  reinterpret_cast<TB *>(pA)->func();
  static_cast<TB *>(pA)->func();
 // dynamic_cast<TB *>(pA)->func();
  if (TB *pD=dynamic_cast<TB *>(pA)) pD->func();
  else cout<<"Cast Error"<<endl;
  return 0;}
```

Восходящее приведение

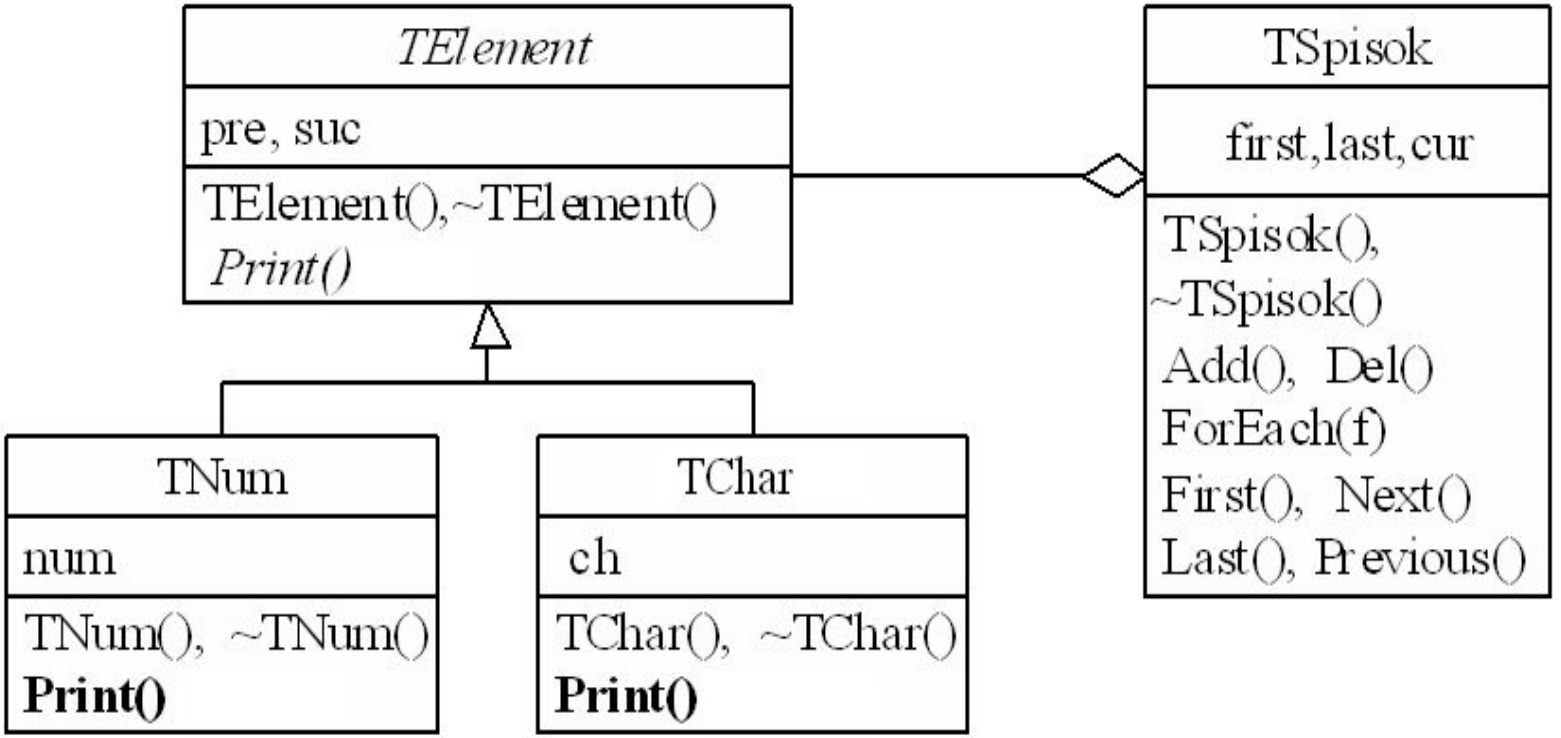
Нисходящее приведение

Ошибка! Приведение не корректно

# 5.7 Контейнер «Двусвязный список» (Ex5_08)

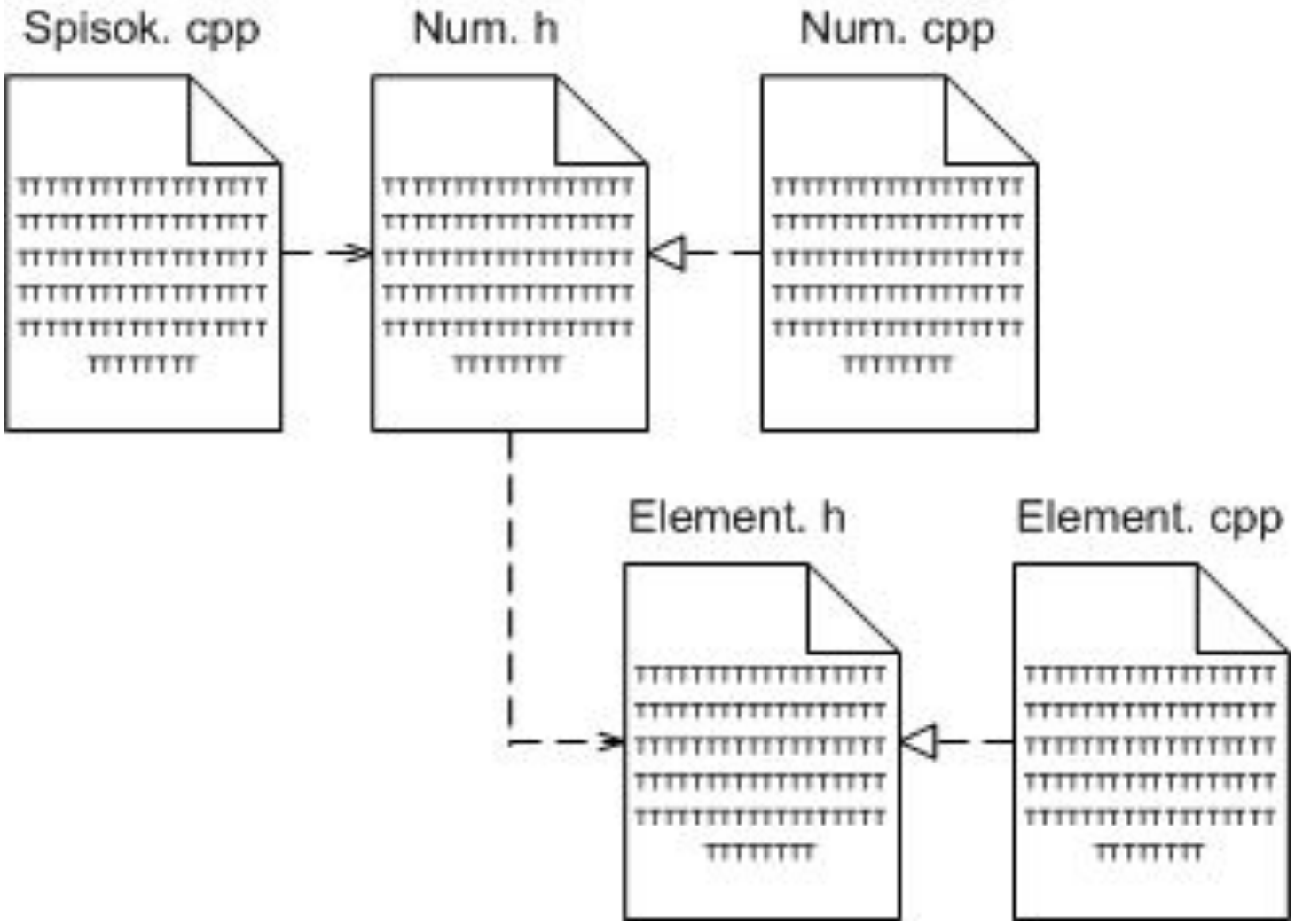



## Диаграмма классов



# Контейнер «Двусвязный список»(2)

## Диаграмма компоновки



# Файл Element.h

```
#include <stdio.h>
class TElement
  {  public:    TElement *pre,*suc;
             TElement() { pre=suc=NULL;}
             virtual ~TElement();
             virtual void Print()=0;
  };
class TSpisok
  {  private: TElement *first,*last,*cur;
     public:  TSpisok() {first=last=cur=NULL;}
          ~TSpisok();
          void Add(TElement *e);
          TElement *Del();
          void ForEach(void (*f)(TElement *e));
          TElement *First(){return cur=first;}
          TElement *Next(){return cur=cur->suc;}
          TElement *Last(){return cur=last;}
          TElement *Previous(){return cur=cur->pre;}
  };
```

# Файл Element.cpp

```
#include "Element.h"
TElement::~TElement()
{ puts("Delete TElement.");
}
TSpisok::~TSpisok()
{  puts("Delete TSpisok");
   while ((cur=Del())!=nullptr)  { cur->Print();
                                   delete(cur);  }
}
void TSpisok::Add(TElement *e)
{  if (first== nullptr) first=last=e;
   else  { e->suc=first;
           first->pre=e;
           first=e;  }
}
```

# Файл Element.cpp (2)

```
TElement *TSpisok::Del(void)
{  TElement *temp=last;
   if (last!= nullptr)
      {last=last->pre;
       if (last!=nullptr) last->suc=nullptr;
      }
   if (last==nullptr) first=nullptr;
   return temp;
}
void TSpisok::ForEach(void (*f)(TElement *e))
{  cur=first;
   while (cur!=nullptr)
      {(*f)(cur);
        cur=cur->suc;
      }
}
```

# Файл Num.h

```
#include "Element.h"
class TNum:public TElement
{ public: int num;
          TNum(int n):TElement(),num(n) {}
          ~TNum() override;
          void Print() override;
};
class TChar:public TElement
{ public: char ch;
          TChar(char c):TElement(),ch(c) {}
          ~TChar() override;
          void Print() override;
};
void Show(TElement *e);
```

# Файл Num.cpp

```
#include "Num.h"

TNum::~TNum() {
   puts("Delete TNum.");
}
void TNum::Print() {
   printf("%d ",num);
}
TChar::~TChar() {
   puts("Delete TChar.");
}
void TChar::Print() {
   printf("%c ",ch);
}
void Show(TElement *e) {
   e->Print();
}
```

# Тестирующая программа

```
#include "Num.h"
#include <string.h>
#include <stdlib.h>
int main()
{  TSpisok N;
   char str[10];
   int k,i;
   TElement *p;
   while(printf("Input numbers, strings or <end>:"),
         scanf("%s",str),strcmp(str,"end"))
   { k=atoi(str);
     if (k||(strlen(str)==1 && str[0]=='0'))  p=new TNum(k);
     else p=new TChar(str[0]);
     N.Add(p);
   }
   puts("All list:");
   N.ForEach(Show);
```

# Тестирующая программа(2)

```
   p=N.First(); k=0;
   while (p!=nullptr)
    { if (TNum *q=dynamic_cast<TNum *>(p)) k+=q->num;
   // VS установить создание RTTI: /GR в Project\Settings…
      p=N.Next();
    }
   printf("Summa= %d\n",k);
   p=N.Last();
   i=0;
   while (p!= nullptr)
    { if (TChar *q=dynamic_cast<TChar *>(p)) str[i++]=q->ch;
      p=N.Previous();
    }
   str[i]='\0';
   printf("String= %s\n",str);
   return 0;
}
```