

## Оглавление

<b><u>АННОТАЦИЯ.....</u></b>	<b><u>5</u></b>
<b><u>1 КЛАССЫ, ОБЪЕКТЫ И ДОСТУП К ОБЪЕКТАМ И ИХ КОМПОНЕНТАМ</u></b>	<b><u>6</u></b>
1.1 Описание класса	6
1.2 Создание объектов при отсутствии конструктора в классе. Обращение к общедоступным полям и методам класса. Неявный параметр this	9
1.3 Создание и уничтожение объектов с использованием конструкторов и деструкторов	15
1.4 Константные объекты. Перегрузка методов для константных объектов	27
1.5 Классы с динамическими полями	32
1.6 Создание, инициализация и уничтожение динамических объектов	35
1.7 Динамические массивы объектов и массивы указателей на объекты	38
Вопросы для самоконтроля	45
<b><u>2 ПОСТРОЕНИЕ ИЕРАРХИИ КЛАССОВ</u></b>	<b><u>46</u></b>
2.1 Наследование	46
2.2 Множественное наследование	54
2.3 Виртуальное наследование	58
2.4 Простой полиморфизм	61
2.5 Сложный полиморфизм	64
2.6 Абстрактные функции и классы	70
Вопросы для самоконтроля	74
<b><u>3 КОМПОЗИЦИЯ И НАПОЛНЕНИЕ</u></b>	<b><u>75</u></b>
3.1 Композиция. Объектные поля класса	75
3.2 Наполнение	80
3.3 Особенности работы с динамическими полиморфными объектами	83
3.4 Восходящее и нисходящее изменение типа объектов	85
3.5 Контейнерные классы	91
Вопросы для самоконтроля	99
<b><u>4 ОСОБЫЕ СЛУЧАИ ОРГАНИЗАЦИИ ДОСТУПА К ОБЪЕКТАМ И ИХ КОМПОНЕНТАМ</u></b>	<b><u>100</u></b>

4.1 Локальные и вложенные классы	100
4.2 Статические компоненты класса	103
4.3 Дружественные функции и классы	107
Вопросы для самоконтроля	112
<b>5 ПЕРЕОПРЕДЕЛЕНИЕ ОПЕРАЦИЙ</b>	<b>113</b>
5.1 Функции-операторы, их типы и ограничения на переопределение	113
5.2 Описание компонентных функций-операторов	115
5.3 Описание внешних функций-операторов	117
5.4 Особенности переопределении операции присваивания	121
5.5 Переопределение операций для объектов с динамическими полями. Контроль освобождения динамической памяти	124
Вопросы для самоконтроля	128
<b>6 ШАБЛОНЫ</b>	<b>129</b>
6.1 Параметризованные функции	129
6.2 Параметризованные классы	131
6.3 Контейнеры на основе шаблонов	141
Вопросы для самоконтроля	144
<b>7 ИСКЛЮЧЕНИЯ</b>	<b>145</b>
7.1 Механизм исключений C++	145
7.2 Механизм исключений C	152
7.3 Совместное использование различных механизмов обработки исключений	159
Вопросы для самоконтроля	162
<b>8 СОЗДАНИЕ И УНИЧТОЖЕНИЕ ДИНАМИЧЕСКИХ ОБЪЕКТОВ.</b>	
<b>«УМНЫЕ УКАЗАТЕЛИ»</b>	<b>163</b>
8.1 Проблемы работы с динамическими объектами	163
8.2 Шаблон auto_ptr	166
8.3 Шаблон shared_ptr	170
8.4 Шаблон weak_ptr	173
Вопросы для самоконтроля	176
<b>ЛИТЕРАТУРА</b>	<b>177</b>

МГТУ им. Н.Э. Баумана  
Факультет Информатика и Системы Управления  
Кафедра ИУ-6 Компьютерные системы и сети  
Иванова Галина Сергеевна, Ничушкина Татьяна Николаевна

**Объектно-ориентированное программирование  
на языке C++ в среде Visual Studio 2008.**

Электронное учебное пособие по дисциплине  
Объектно-ориентированное программирование

МОСКВА

2013 год МГТУ им. Баумана

## **Аннотация**

Учебное пособие посвящено описанию средств и механизмов объектно-ориентированного программирования на языке С++ в среде программирования Visual Studio 2008. Оно является продолжением аналогичного учебного пособия, посвященного описанию средств процедурного (структурного) программирования в той же среде, и предполагает знакомство с особенностями процедурного программирования на С++.

Объектная модель С++ среди моделей современных языков программирования является одной из самых развитых. В ней используется надежная схема сокрытия компонентов (полей и методов) класса, реализованы перегрузка операций, возможность применения статических компонентов, создания шаблонов функций и классов и др.

Основное внимание уделяется разъяснению различных аспектов объектной модели С++ и особенностей ее реализации в рассматриваемой среде. Пособие содержит большое количество примеров программ. При необходимости все примеры могут быть скопированы и запущены в программной среде.

Также пособие содержит контрольные вопросы, ответы на которые позволят учащимся в лучшем усвоении представленного материала.

Пособие предназначено для студентов 1 курса кафедр ИУ6 и АК5, обучающихся по учебному плану бакалаврской подготовки.

# 1 Классы, объекты и доступ к объектам и их компонентам

## 1.1 Описание класса

В C++ так же, как и в других языках программирования, класс – создаваемый программистом структурный тип данных, который используется для описания множества объектов предметной области, имеющих общие свойства и поведение.

Класс объявляется следующим образом:

```
class <Имя класса>
{
private:    <Внутренние (недоступные) компоненты класса>
protected: <Защищенные компоненты класса>
public:    <Общие (доступные) компоненты класса>
};
```

Описание предусматривает три секции. Компоненты класса, объявленные в секции `private`, называются *внутренними*. Они доступны только компонентным функциям того же класса и функциям, объявленным *дружественными* (см. раздел 4.3) описываемому классу.

Компоненты класса, объявленные в секции `protected`, называются *защищенными*. Они доступны компонентным функциям не только данного класса, но и его потомков. При отсутствии наследования – интерпретируются как внутренние.

Компоненты класса, объявленные в секции `public`, называются *общими*. Они доступны за пределами класса в любом месте программы. Именно в этой секции осуществляется объявление полей и методов *интерфейсной части* класса.

Если при описании класса тип доступа к компонентам не указан, то по умолчанию принимается тип `private`.

В качестве компонентов в описании класса фигурируют *поля*, применяемые для хранения параметров объектов, и *функции*, описывающие правила взаимодействия с этими полями. В соответствии со стандартной терминологией ООП функции – компоненты класса или *компонентные функции* можно называть *методами*.

Компонентные функции или методы могут быть описаны как внутри, так и вне определения класса. В последнем случае определение класса должно содержать прототипы этих функций, а заголовок описываемой функции должен включать *квалификатор видимости*, который состоит из имени класса и знака «::». Таким образом, компилятору сообщается, что определяемой функции доступны внутренние поля класса:

```
<Тип результата> <Имя класса> :: <Имя функции>(<Список параметров>)
{
    <Тело компонентной функции>
}
```

### Пример 1.1. Описание класса.

#### А. Описание компонентных функций внутри класса.

```
#include <stdio.h>
class First
{
public:
    char c;
    int x,y;
    /* компонентные функции, определенные внутри класса */
    void print(void)
    {
        printf ("%c %d %d ",c,x,y);
    }
    void set(char ach,int ax,int ay)
    {
        c=ach;    x=ax;    y=ay;
    }
};
```

#### Б. Описание компонентных функций вне класса.

```
#include <stdio.h>
class First
```

```

{
    public:    char c;
              int x,y;
              void print(void);
              void set(char ach,int ax,int ay);
};

    /* компонентные функции, описанные вне класса */
void First::print(void)
{    printf ("%c %d %d ",c,x,y);    }
void First::set (char ach,int ax,int ay)
{    c=ach;    x=ax;    y=ay;    }

```

Согласно стандарту C++, если тело компонентной функции размещено в описании класса, то эта функция по умолчанию считается *встраиваемой* (*inline*). Коды таких функций компилятор помещает непосредственно в место их вызова, что ускоряет работу программы, но увеличивает ее размер и накладывает некоторые ограничения на использование языковых средств. Так, встраиваемыми не могут быть функции, содержащие операторы цикла, операторы безусловного перехода, ассемблерные вставки, а также виртуальные компонентные функции и функции, реализующие рекурсивные алгоритмы.

Обычно при попытке встраивания таких функций компиляторы C++ выдают сообщение об ошибке или предупреждение. Однако *в старших версиях Visual C++, начиная с Visual Studio 2005, компилятор самостоятельно принимает решение о реализации конкретного метода как встраиваемого, а соответственно и не выдает никаких сообщений, игнорируя описание *inline**. Поэтому особенности встраивания компонентных функций далее рассматривать не будем.

## 1.2 Создание объектов при отсутствии конструктора в классе. Обращение к общедоступным полям и методам класса. Неявный параметр `this`

В программе, использующей классы, по мере необходимости объявляют объекты этих классов.

*Объекты* – переменные программы, соответственно на них распространяются общие правила длительности существования и области действия переменных, а именно:

<sup>35</sup><sub>17</sub> внешние, статические и внешние статические объекты создаются до вызова функции `main()` и уничтожаются по завершении программы;

<sup>35</sup><sub>17</sub> автоматические объекты создаются каждый раз при вызове функции, в которой они объявлены, и уничтожаются при выходе из нее;

<sup>35</sup><sub>17</sub> объекты, память под которые выделяется динамически, создаются оператором `new` и уничтожаются оператором `delete`.

При объявлении полей в описании класса не допускается их инициализация, поскольку в момент описания класса память для размещения его полей еще не выделена. Выделение памяти осуществляется не для класса, а для объектов этого класса, поэтому возможность инициализации полей появляется только во время или после объявления объекта конкретного класса.

Объявление объектов и способы инициализации их полей зависят от наличия или отсутствия в классе специального инициализирующего метода – *конструктора*, а также от того, в какой секции класса описано инициализируемое поле. Конструктор, являясь методом класса, может инициализировать любое поле объекта при его создании (см. раздел 1.3).

Если в классе отсутствует конструктор, но описаны защищенные `protected` или скрытые `private` поля, то возможно создание только неинициализированных объектов. Для этого используется стандартная конструкция объявления переменных или указателей на них. Например:

```
First a, // объект класса First
      *b, // указатель на объект класса First
      c[4]; // массив c из четырех объектов класса First
```

При объявлении указателя, как и для обычных переменных, память под объект не выделяется. Это необходимо сделать отдельно, используя операцию `new`, после работы с динамическим объектом память необходимо освободить:

```
b=new First; ... delete b;
```

Объект, созданный таким способом, называют *динамическим*.

Значения полей неинициализированных статических и динамических объектов или массивов объектов задают в процессе дальнейшей работы с объектами: защищенных и скрытых – только в методах класса, а общедоступных – в методах класса или непосредственным присваиванием в программе.

При отсутствии в классе конструктора и защищенных `protected` или скрытых `private` полей для объявления инициализированных объектов используют оператор инициализации, применяемый при создании инициализированных структур, например:

```
First a = {'A', 3, 4},
         c[4] = {'A', 1, 4}, {'B', 3, 5}, {'C', 2, 6}, {'D', 1, 3};
```

Инициализирующие значения при этом должны перечисляться в порядке следования полей в описании класса.

**Пример 1.2.** Создание инициализированных и неинициализированных объектов при отсутствии конструктора.

```
#include <conio.h>
class Num
{
public:
    int n;
};
void main(int argc, char* argv[])
{
    Num N = {56}; // инициализированный объект
    Num NN;      // неинициализированный объект
    _getch();
}
```

### Обращение к общедоступным полям и методам объекта из программы.

Обращение к общедоступным полям и методам объекта из программы может осуществляться с помощью полных имен, каждое из которых имеет вид

```
<Имя объекта> . <Имя класса> :: <Имя поля или функции> ;
```

Например:

```
a.First::set('A', 3, 4); // статический объект
b->First::set('B', 3, 4); // динамический объект
c[i].First::set('C', 3, 4); // массив объектов
```

Однако обычно доступ к компонентам объекта обеспечивается с помощью укороченного имени, в котором квалификатор доступа опущен, тогда принадлежность к классу определяется по типу объекта:

```
<Имя объекта>.<Имя поля или функции>
<Имя указателя на объект> -> <Имя поля или функции>
<Имя объекта>[<Индекс>].<Имя поля или функции>
```

Например:

```
a.x                b->x                c[i].x
a.set('A', 3, 4)   b->set('B', 3, 4)   c[i].set('C', 3, 4)
```

Первая строка демонстрирует обращение к общедоступным полям простого объекта, динамического объекта и элемента массива объектов. Вторая строка – обращение к общедоступным методам соответствующих объектов.

**Пример 1.3.** Различные способы инициализации общедоступных полей объекта.

```
#include <locale.h>
#include <string.h>
#include <stdio.h>
```

```

#include <conio.h>
class sstro
{
public:
    char str1[80];
    int x,y;
    void set_str(char *vs) // инициализация полей
    {
        strcpy(str1,vs);    x=0;    y=0;
    }
    void print(void)       // вывод содержимого полей
    {
        printf("x=%5d    y=%5d    str: ",x,y);
        puts(str1);
    }
};
void main()
{
    setlocale(0,"russian");
    sstro aa = {"Строка",200,400}; // инициализированный объект
    sstro bb,cc; // неинициализированные объекты
    bb.x=200; // инициализация посредством прямого обращения
    bb.y=150;
    strcpy(bb.str1,"Строка");
    cc.set_str("Строка"); // вызов инициализирующего метода
    aa.print(); bb.print(); cc.print();
    _getch();
}

```

Результат работы программы:

```

x= 200    y= 400    str: Строка
x= 200    y= 150    str: Строка
x=  0     y=  0     str: Строка

```

**Неявный параметр `this`.** При вызове компонентной функции для конкретного объекта, ей неявно передается в качестве параметра указатель на поля того объекта, для которого она вызывается. Этот указатель имеет специальное имя `this` и неявно определен как константный в каждой функции класса:

```
<Имя класса> *const this;
```

Поскольку указатель константный, его изменять нельзя, однако в каждой принадлежащей классу функции он указывает на поля именно того объекта, для которого вызывают функцию. Иными словами, указатель `this` является дополнительным (скрытым) параметром каждой нестатической (см. далее) компонентной функции.

Указатель `this` используется для доступа к полям конкретного объекта. Фактически обращение к тому или иному полю объекта или его методу выглядит следующим образом:

```
this->pole      this->str      this->fun().
```

Соответственно при объявлении некоторого объекта `A` выполняется операция `this=&A`, а при выделении памяти для размещения динамического объекта, адресуемого указателем `b`, – операция `this=b`.

При работе с компонентами класса этот указатель можно задавать явным образом. Но следует отметить, что в этом случае использование `this` обычно не дает никакого преимущества, так как данные большинства конкретных объектов уже доступны в принадлежащих классу функциях по именам.

Однако иногда явное применение указателя `this` полезно и даже необходимо. Например, без него нельзя обойтись, если в теле принадлежащей классу функции требуется явно задать адрес объекта, для которого она была вызвана.

**Пример 1.4.** Использование параметра `this`.

```
#include <stdio.h>
#include <conio.h>
class TA
{
    int x,y;
public:
```

```

void print(void)
{
    printf("x=%5d  y=%5d\n", x, y);
}
TA *fun1() // возвращает указатель на объект, для которого вызывается
{
    x=y=100;
    return this;
}
TA fun2() // возвращает объект, для которого вызывается
{
    x=y=200;
    return *this;
}
};
void main()
{
    TA aa;
    aa.fun1()->print(); // выводит: 100 100
    aa.fun2().print(); // выводит: 200 200
    _getch();
}

```

Кроме того, указатель `this` явно используют для формирования результата при переопределении операций (см. главу 5), так как операция переопределяется для конкретного вызывающего ее объекта.

### 1.3 Создание и уничтожение объектов с использованием конструкторов и деструкторов

*Конструктор* – метод класса, который *автоматически вызывается при выделении памяти под объект*.

По правилам C++ *конструктор* имеет то же имя, что и класс, не наследуется в производных классах, может иметь аргументы, но не возвращает значения, может быть параметрически перегружен.

Конструктор определяет операции, которые необходимо выполнить при создании объекта. Традиционно такими операциями являются инициализация полей класса и выделение памяти под динамические поля, если такие в классе объявлены. Явный вызов конструктора не возможен, что в некоторых случаях усложняет создание инициализированных объектов.

При освобождении объектом памяти автоматически вызывается другой специальный метод класса – *деструктор*. Имя деструктора по аналогии с именем конструктора, совпадает с именем класса, но перед ним стоит символ «~» («тильда»). Деструктор определяет операции, которые необходимо выполнить при уничтожении объекта. Обычно он используется для освобождения памяти, выделенной под динамические поля объекта данного класса конструктором, и при необходимости может быть объявлен виртуальным. Деструктор не возвращает значения, не имеет параметров и не наследуется производными классами. Класс может иметь только один деструктор или не иметь ни одного. В отличие от конструктора деструктор может вызываться явно.

Момент уничтожения объекта, а, следовательно, и автоматического вызова деструктора определяется типом памяти, выбранным для размещения объекта: локальная, глобальная, внешняя и т. д. Если программа завершается с использованием функции `exit`, то вызываются деструкторы только глобальных объектов. При аварийном завершении программы, использующей объекты некоторого класса, функцией `abort` деструкторы объектов не вызываются.

**Пример 1.5.** Создание, инициализация и уничтожение объекта при наличии в классе конструктора и деструктора.

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

```

using namespace std;
class Num
{
    int n;
public:
    Num(int an){ cout<<"Конструктор"<<endl; n=an; }
    ~Num()      { cout<<"Деструктор"<<endl; }
};
void main(int argc, char* argv[])
{
    setlocale(0,"russian");
    Num N(56);
    system("pause");
}

```

Результат выполнения программы:

Конструктор

Для продолжения нажмите любую клавишу . . .

Деструктор

В соответствии с полученными результатами конструктор класса был вызван при объявлении объекта, а деструктор – во время завершения программы в момент освобождения памяти, выделенной под объект.

**Переопределение конструкторов. Конструкторы с аргументами, заданными по умолчанию.** Как и любая другая функция с параметрами, конструктор может быть переопределен (паметрически перегружен). Поэтому класс может иметь несколько конструкторов, позволяющих использовать разные способы инициализации полей объектов.

**Пример 1.6.** Многократное переопределение конструктора.

```

#include <locale.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>

```

```

class sstr
{
private:    char str1[60];
public:    int x,y;

    sstr() // стандартная инициализация всех полей
    {
        strcpy(str1,"Строка по умолчанию.");
        x=0;  y=0;
    }
    sstr(char *vs) // инициализация поля str1
    {
        strcpy(str1,vs);
        x=0;  y=0;
    }
    sstr(char *vs,int vx) // инициализация поля str1 и x
    {
        strcpy(str1,vs);
        x=vx;  y=0;
    }
    sstr(char *vs,int vx,int vy)// инициализация всех полей
    {
        strcpy(str1,vs);
        x=vx;  y=vy;
    }
    void print(void)
    {
        puts("Значения полей объекта: ");
        printf("x=  %5d  y= %5d \n",x,y);
        printf("str1= ");  puts(str1);
    }
};

void main()
{
    setlocale(0,"russian");

```

```

sstr a0; // объект с полями по умолчанию
sstr a1("Строка"); // объект с заданной строкой
sstr a2("Строка",100); // объект с заданными строкой и x
sstr a3("Строка",200,150); // полностью инициализированный объект
a0.print(); a1.print(); a2.print(); a3.print();
_getch();
}

```

Иногда в конструкторах целесообразно использовать параметры, значение которых задаются в заголовке по умолчанию. В этом случае вместо нескольких переопределяемых конструкторов обычно можно описать один. Вернемся к предыдущему примеру и заменим все переопределяемые конструкторы одним.

**Пример 1.7.** Использование конструктора с аргументами, заданными по умолчанию.

```

#include <locale.h>
#include <string.h>
#include <iostream>
using namespace std;
class sstr
{
private:    char str1[50];
public:    int x,y;
    sstr(char *vs,int vx,int vy); /* прототип конструктора с
                                параметрами, заданными по умолчанию */
    void print(void)
    {
        cout<<"    Содержимое полей :"<< endl;
        cout<<" x= "<<x<<" y= "<<y<<endl;
        cout<<" str1="<<str1<<endl;
    }
};

sstr::sstr(char * vs="Строка по умолчанию",
            int vx=80,int vy=90) /* тело конструктора с
                                параметрами, заданными по умолчанию */

```

```

{
    strcpy(str1,vs);
    x=vx;    y=vy;
}
void main()
{
    setlocale(0,"russian");
    sstr a0; // объект с полями по умолчанию
    sstr a1("Строка"); // объект с заданной строкой
    sstr a2("Строка",100); // объект с заданными строкой и x
    sstr a3("Строка",200,150); // объект с заданными строкой, x и y
    a0.print();
    a1.print();
    a2.print();
    a3.print();
    system("pause");
}

```

**Список инициализации.** Список инициализации – специальная конструкция, включаемая в описание конструктора. Назначение этой конструкции – инициализация полей особых типов, например, константных и ссылочных. Кроме этого список инициализации применяют для явного вызова конструкторов объектных полей (см. раздел 3.1) и полей базового класса (см. раздел 2.1). Однако эта конструкция может применяться и для инициализации обычных полей класса.

Список инициализации отделяется от заголовка конструктора двоеточием и состоит из записей вида:

<Имя> (<Список выражений>),

где в качестве имени могут фигурировать: имя поля данного класса, имя объекта другого класса, включенного в данный класс, или имя базового класса. Список выражений определяет значения, используемые для инициализации указанных частей создаваемого объекта.

**Пример 1.8.** Использование конструктора со списком инициализации.

```

#include <iostream>
using namespace std;
class obinteg
{
public:
    const int x,y;          // константные поля
    int &c;                  // ссылочное поле
    void print(void)
    {
        cout<<" x="<<x<<" y= "<<y<<" color="<<c<<endl;
    }
    obinteg(int vx,int vy,int &vc):
        x(vx) , y(vy) , c(vc) {} /* инициализация полей фиксированного
        и ссылочного типа посредством списка инициализации */
};
void main()
{
    int k=13;
    obinteg bb(40,100,k);    // инициализируемый объект
    bb.print();
    system("pause");
}

```

**Конструкторы без параметров.** Если в классе не объявлены конструктор и деструктор, то компилятор C++ автоматически выполняет построение «пустых» (без параметров и операторов) конструктора и деструктора. Если же хотя бы один конструктор в классе объявлен, то автоматический пустой конструктор уже не создается. Это значит, что, если при наличии в классе только конструкторов, требующих задания параметров, попробовать создать объект, не задавая аргументов, то мы получим сообщение об отсутствии соответствующего конструктора. Так, если в пример 1.5 добавить строку

```
Num cc; // неинициализированный объект
```

то при компиляции будет получено сообщение:

```
error C2512: 'Num': no appropriate default constructor available
```

Таким образом создание объектов без указания аргументов требует, чтобы в классе был задан один из конструкторов, которые могут быть *вызваны без указания аргументов*, а именно:

<sup>35</sup>/<sub>17</sub> неинициализирующий конструктор без параметров («пустой»), например:

```
Num () { }
```

<sup>35</sup>/<sub>17</sub> инициализирующий конструктор без параметров, например:

```
Num () { n=0; } или со списком инициализации Num () : n (0) { }
```

<sup>35</sup>/<sub>17</sub> инициализирующий конструктор, предполагающий значения по умолчанию для всех полей, например:

```
Num (int an=0) { n=an; }
```

При объявлении объекта с использованием неинициализирующего конструктора значения полей объекта не будут заданы, хотя память под объект будет выделена. Инициализирующий конструктор без параметров или конструктор с параметрами, заданными по умолчанию, обеспечат запись в поля объекта значений, определенных этими конструкторами. Если объектов одного класса в программе несколько, то у всех объектов, которые создаются инициализирующим конструктором без параметров или с параметрами, заданными по умолчанию, соответствующие поля будут иметь одинаковые значения.

Следует иметь в виду, что класс может содержать *только один* конструктор, который может быть вызван без указания аргументов. Если таких конструкторов было бы несколько, то компилятор не смог бы решить, какой конструктор должен быть вызван при объявлении переменных без аргументов.

Особенно важно наличие конструктора, который можно вызывать без аргументов, при создании массива объектов.

**Создание и инициализация массивов объектов.** Если при объявлении массива объектов класса, включающего конструкторы, использовать стандартную форму, то для каждого объекта массива будет вызван неинициализирующий или инициализирующий конструктор без аргументов. Например:

```
Point masPoint[10]; // десять раз вызывается конструктор без аргументов
```

Если класс `Point` описан с конструкторами, но при этом в нем отсутствует конструктор, который можно вызывать без аргументов, то при выполнении программы на этапе создания массива будет получено сообщение об ошибке `error C2512`, которое зафиксирует отсутствие в классе указанного конструктора. Если же в классе вообще отсутствуют конструкторы, то ошибка зафиксирована не будет, поскольку автоматически будет создан неинициализирующий конструктор без параметров.

Если при создании массива использовался неинициализирующий конструктор, то все объекты – элементы массива будут неинициализированными. Применение инициализирующего конструктора без параметров или конструктора с параметрами, заданными по умолчанию, как указано выше, может:

<sup>35</sup><sub>17</sub> инициализировать все создаваемые объекты одинаково, как задано в инициализирующем конструкторе – редко требуется по условию задачи;

<sup>35</sup><sub>17</sub> инициализировать все создаваемые объекты случайным образом – еще реже требуется по условию задачи;

<sup>35</sup><sub>17</sub> инициализировать все создаваемые объекты вводимыми в процессе инициализации значениями – ввод в методе инициализации считается нетехнологичным.

Для правильной (соответствующей условиям задачи) инициализации объектов во всех трех описанных выше случаях скорее всего понадобится специальный инициализирующий метод, задающий значения полей, например:

```
void Point::setPoint(float af){ f=af; } // инициализирующий метод
```

Также для создания массивов объектов, инициализированных данными решаемой задачи, можно применить специальную конструкцию – *конструкцию, обеспечивающую явный вызов инициализирующего конструктора с указанием необходимых аргументов*, например:

```
Point A[3]={Point(3.5), Point(8.13), Point(10.7)};
```

В результате будет сформирован массив из трех элементов, у каждого из которых будет свое значение поля `f`.

**Пример 1.9.** Объявление и инициализация массивов объектов класса, включающего конструкторы.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
class Num
{
    int n;
public:
    Num(int an):n(an) {} // инициализирующий конструктор
    Num() {} // неинициализирующий конструктор
    void setNum(int an){n=an;} // инициализирующий метод
};
void main()
{
    setlocale(0, "russian");
    int nn;
    Num A[5]; // неинициализированный массив
    for(int i=0;i<5;i++) // инициализация элементов массива
    {
        printf("Введите значение поля %d-го объекта:", i);
        scanf("%d", &nn);
        A[i].setNum(nn);
    }
    // инициализированный массив
    Num B[3]={Num(3), Num(8), Num(10)};
    _getch();
}
```

**Копирующий конструктор.** В языке C++ при объявлении объектов допускается использовать операцию присваивания, в правой части которой указано имя ранее определенного объекта, например:

```
Point A(20, 6), C=A;
```

Аналогичная операция неявно выполняется при вызове подпрограммы, среди параметров которой есть объект, передаваемый в подпрограмму в качестве параметра-значения. По правилам C++ при этом создается копия объекта-параметра, что также предполагает копирование полей.

При выполнении операции присваивания объектов для инициализации полей объекта-копии вызывается специальный *копирующий конструктор*, который в списке параметров содержит параметр типа определяемого класса, например:

```
Class1(const Class1 &t); или Class1(const Class1 &t, int a=0);
```

Копирующие конструкторы могут определяться в классе явно, но могут использоваться и копирующие конструкторы, определенные по умолчанию. В последнем случае предполагается, что для каждого класса описан копирующий конструктор вида

```
<Имя класса> (const <Имя класса>&),
```

который в зависимости от структуры класса может содержать список инициализации, включающий копирующие конструкторы базового класса и объектных полей. Такой конструктор автоматически строится компилятором и обеспечивает *последовательное копирование всех полей объекта*. При этом если для какого-либо поля или базового класса явно определен копирующий конструктор без `const`, то и конструктор класса в целом неявно определяется без `const`.

**Пример 1.10.** Использование автоматически создаваемого копирующего конструктора.

```
#include <locale.h>
#include <string.h>
#include <iostream>
using namespace std;
class child
{
private:    char name[20];    int age;
```

```

public:
    void print(void)
    {
        cout<<" Имя: "<<name;
        cout<<" Возраст : "<<age<< endl;
    }
    child(char *Name,int Age):age(Age)
    {
        strcpy(name,Name);
    }
};
void main()
{
    setlocale(0,"russian");
    child aa("Мария",6);
    aa.print();    // выводит: Имя: Мария Возраст: 6
    child bb=aa;    // вызывает копирующий конструктор
    bb.print();    // выводит: Имя: Мария Возраст: 6
    system("pause");
}

```

В данном примере компилятор автоматически создает конструктор, согласно которому поля объекта aa без изменения копируются в поля объекта bb:

```
child(const child & obj):name(obj.name),age(obj.age){}
```

Следует помнить, что автоматически генерируемый конструктор не учитывает особенностей объекта, а потому не всегда применим. Известны два случая, когда копирующий конструктор *необходимо* (!) описывать в классе явно:

<sup>35</sup><sub>17</sub> если при копировании полей необходимо изменять содержимое хотя бы некоторых из них;

<sup>35</sup><sub>17</sub> если класс содержит хотя бы одно динамическое поле (см. раздел 1.4).

**Пример 1.11.** Явное определение копирующего конструктора, изменяющего содержимое поля при копировании.

```

#include <locale.h>
#include <iostream>
using namespace std;
class Num
{
    int n;
public:
    Num(int an):n(an) {} // инициализирующий конструктор
    Num(const Num& ob) // копирующий конструктор
    { n = ob.n*3; }
    void print(void)
    { cout<<" Поле n ="<<n<<" "<< endl; }
};
void main()
{
    setlocale(0,"russian");
    Num aa(10); // вызывается инициализирующий конструктор
    aa.print(); // выводит Поле n = 10
    Num bb=aa; // вызывается копирующий конструктор
    bb.print(); // выводит Поле n = 30
    system("pause");
}

```

Использованное в последнем примере применение копирующих конструкторов встречается достаточно редко, поскольку работа с программой при этом существенно усложняется вследствие необходимости помнить особенности копирования объектов. Гораздо чаще приходится описывать копирующие конструкторы для классов, содержащих динамические поля.

## 1.4 Константные объекты. Перегрузка методов для константных объектов

Для предотвращения ошибочного изменения объект, как и любая другая переменная, может быть объявлен неизменяемым (константным). В этом случае он должен описываться с указанием `const`:

```
const <Имя класса> <Имя объекта>;
```

В классе, для которого возможно создание константных объектов, можно описать специальные методы, объявив передаваемый по умолчанию объект `*this` неизменяемым. В этом случае служебное слово `const` указывается после заголовка и параметров метода, но до его тела, например:

```
class A
{
public:
    voidproc() const{...} // метод для константного объекта
};
```

При таком описании компилятор будет запрещать в методе изменение полей константного объекта, выдавая соответствующее сообщение, например:

```
class A
{
private:
    int x;
public:
```

```

void f(int a) const // метод определен для константного объекта
{
    x = a; // ошибка компиляции !!!
}
};

```

Методы константных объектов параметрически перегружают методы обычных объектов, поскольку компилятор считает, что обычные и константные объекты имеют различные типы. Определив специальные методы для константных объектов, можно предусмотреть для них отдельную, обычно упрощенную обработку.

**Пример 1.12.** Методы константных объектов.

```

#include <iostream>
#include <conio.h>
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(); // вызывается обычный метод
    A const a2(2);
}

```

```

a2.f ();    // вызывается метод константного объекта

_getch ();

return 0;

}

```

Если класс содержит только метод для константного объекта, то этот метод без проблем будет вызываться и для константных и для обычных объектов. Однако при вызове обычного метода для константного объекта компилятор будет фиксировать ошибку.

```

#include <conio.h>

class A
{
public:
    void f () {} // не const метод
};

int main ()
{
    const A a;

    a.f ();    // Ошибка: обычный метод вызван для константного объекта

    _getch ();

    return 0;

}

```

В результате компиляции получаем сообщение об ошибке:

```
error C2662: 'A::f' : cannot convert 'this' pointer from  
'const A' to 'A &'
```

которое означает, что не возможно преобразование константного объекта к обычному.

То же сообщение мы получим, если метод константного объекта попытается вызвать другой метод, который не объявлен, как `const` метод:

```
#include <conio.h>

class A
{
public:
    unsigned f1 () {} // не const метод
};

class B : public A
{
public:
    void f2 () const
    {
        f1 (); // нельзя использовать не const метод
    }
};

void main()
{
    const B b;

    b.f2 ();

    _getch ();
}
```

В этом случае компилятор выдаст сообщение:

```
error C2662: 'A::f1' : cannot convert 'this' pointer from 'const B' to 'A &'.
```

Чтобы исправить ошибку во втором случае достаточно описать метод `f1()`, как метод для константного объекта. Однако, если исходный класс, включающий этот метод – библиотечный, то доступ к его методам у нас отсутствует.

Для предотвращения ошибок компиляции в обоих случаях можно выполнить преобразование `const_cast`, которое отменяет атрибут `const` объекта.

В первом примере при вызове обычного метода для константного объекта в основной программе следует написать:

```
const_cast<A*>(&a)->f();
```

Во втором примере при вызове обычного метода из метода, описанного для константного объекта, необходимо написать:

```
const_cast<B*>(this)->f1();
```

К сожалению, в обоих случаях, используя преобразование `const_cast`, мы отменяем действие атрибута `const`, нарушая строгую защиту полей объекта от ошибочных изменений.

## 1.5 Классы с динамическими полями

Динамически выделять память под простые переменные, включаемые в объект, невыгодно, так как память расходуется не только для размещения самой переменной, но и под информацию о выделении памяти. Поэтому динамическое распределение памяти обычно используют при работе со сложными структурами данных, особенно, если размер будущего поля заранее не известен (например, зависит от исходных данных и определяется в процессе вычислений).

Наиболее часто динамическое распределение памяти применяют в классах, использующих в качестве полей массивы, строки, структуры и их комбинации. В этом случае поле содержит указатель на переменную соответствующего типа. Указатель получает свое значение при выделении памяти под структуру данных, для которой он определен. Освобождение этой памяти происходит по адресу в указателе в соответствии с типом переменной.

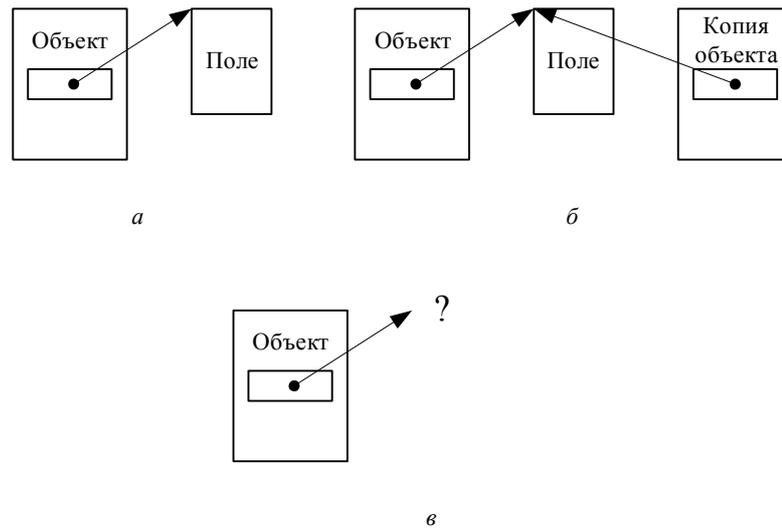
При создании классов с динамическими полями необходимо учитывать большое количество особенностей. Так конструктор такого класса обычно осуществляет выделение участков памяти требуемого размера под переменные, адреса которых присваиваются соответствующим указателям, и обеспечивает контроль наличия доступной памяти.

Деструктор класса соответственно должен при уничтожении объекта освобождать распределенную при конструировании память, поскольку автоматически эта память не освобождается.

При этом, если в классе предусмотрен неинициализирующий конструктор и в программе по какой-то причине динамические поля могут быть не созданы, то в деструкторе нужно проверять факт выделения памяти под поля прежде, чем ее освобождать, иначе при выполнении программы будет сгенерирована ошибка освобождения памяти. Кроме того, для инициализации полей объекта, созданного с помощью неинициализирующего конструктора, следует предусмотреть метод класса, который выделял бы память под поля такого объекта и инициализировал их.

Дополнительные проблемы могут возникнуть при необходимости создания копии объекта, например при передаче объекта в подпрограмму в качестве параметра-значения. В этом случае для реализации операции копирования класс с динамическими полями помимо обычного конструктора, в котором будет выделяться память под размещение динамического поля (например см. рис. 1.1, а), должен включать копирующий

конструктор для корректного создания копии объекта. Это связано с тем, что при использовании стандартного копирующего конструктора адрес динамического поля просто копируется, и после этого мы получаем два объекта, использующих одни и те же динамические поля (см. рис. 1.1, б). Соответственно при уничтожении копии объекта деструктором единое для двух объектов динамическое поле освобождается, оставляя основной объект без динамического поля (см. рис. 1.1, в). Теперь при любом обращении к этому полю мы получим сообщение об ошибке.



**Рис. 1.1. Объект с динамическим полем:**

*a* – после конструирования; *б* – после некорректного создания копии;

*в* – после удаления неверно созданной копии

**Пример 1.13.** Работа с объектом, включающим динамическое поле.

```
#include <stdio.h>
#include <conio.h>
class TNum
{
public: int *pn; // указатель для адреса динамического поля
    TNum(int n) // инициализирующий конструктор
    { pn=new int(n); }
    TNum(const TNum &Obj) // копирующий конструктор
    { pn=new int(*Obj.pn); }
    ~TNum() // деструктор
    { delete pn; }
};
```

```
void Print(TNum b)           // подпрограмма с параметром объектом
{ printf("%d ", *b.pn); }
void main()
{
    TNum A(1);
    Print(A);
    _getch();
}
```

## 1.6 Создание, инициализация и уничтожение динамических объектов

Как было указано в разделе 1.1, по правилам языка C++ объекту некоторого класса память может быть выделена статически – на этапе компиляции или динамически – во время выполнения программы. При этом выделение и освобождение участков осуществляется при выполнении операций `new` и `delete`.

При выделении памяти для отдельных объектов используют следующие формы обращения к операции `new`:

```
<Имя указателя на объект>=new <Имя класса>;
```

или

```
<Имя указателя на объект>=new <Имя класса>(<Список параметров>);
```

Вторую форму применяют при наличии у конструктора списка параметров.

Операция `delete` требует указания только имени объекта:

```
delete <Имя указателя на объект>;
```

Например:

```
Num *a;          // объявление указателя на объект
a = new Num;     // выделение памяти под неинициализированный объект
delete a;       // освобождение выделенной памяти
```

В этом случае класс, под объекты которого выделяется память, должен содержать конструктор, вызываемый без указания аргументов, и при наличии защищенных или скрытых полей – метод инициализации невидимых извне полей.

Для создания инициализированных динамических объектов класс должен содержать инициализирующий конструктор с соответствующими параметрами:

```
Num *b;          // объявление указателя на объект
b = new Num(5); // выделение памяти под инициализированный объект
```

```
delete b;
```

При освобождении памяти автоматически будет вызван деструктор класса, а при его отсутствии автоматически будет сгенерирован «пустой» деструктор вида:

```
Num::~~Num() {}
```

**Пример 1.14.** Использование динамических объектов со статическими полями.

```
#include <locale.h>
#include <iostream>
#include <iomanip>
using namespace std;
class TVector
{
private:    int x,y,z;
public:
    TVector() {} // неинициализирующий конструктор
    TVector(int ax,int ay,int az) // инициализирующий конструктор
    { x=ax;y=ay;z=az; }
    ~TVector() {} // деструктор
    void PrintVec();
    void SetVec(int ax,int ay,int az) // инициализирующий метод
    { x=ax;y=ay;z=az; }
};
void TVector::PrintVec()
{
    cout<<"Значение вектора: "<<setw(5)<<x<<" , ";
    cout<<setw(5)<<y<<" , "<<setw(5)<<z<<"\n";
}
void main()
{
    setlocale(0,"russian");
    TVector *a,*b; // два указателя на объекты класса
```

```

// выделяем память под динамические объекты класса
a=new TVector(12,34,23); // инициализированный объект
b=new TVector;          // неинициализированный объект
b->SetVec(10,45,56); // инициализация объекта
a->PrintVec();        // выводит: 12, 34, 23
b->PrintVec();        // выводит: 10, 45, 56

// освобождаем память, выделенную под динамические объекты класса
delete a;             // вызывает деструктор
delete b;             // вызывает деструктор
system("pause");
}

```

При работе с динамическими объектами следует помнить, что *присваивание одного объекта другому с помощью указателей сводится к копированию адреса*: после выполнения операции первый указатель содержит тот же адрес, что и второй. Старое значение в первом указателе стирается, и, если он содержал адрес некоторого объекта, то память, выделенная ранее под этот объект, остается занятой и более недоступной. Такая ошибка получила название «утечка памяти». К ошибке также приведет попытка освободить память по обоим указателям, так как один и тот же участок памяти, выделенный под объект, освобождается дважды. Во избежание этих ошибок необходимо учитывать и контролировать подобные ситуации.

На класс, используемый для создания динамических объектов, можно возложить задачу управления памятью, если переопределить операции `new` и `delete`. Особенно это полезно для классов, которые являются базовыми для многочисленных производных классов.

Кроме этого в конструкторе или методе, выполняющем распределение памяти, следует предусмотреть контроль ее наличия. При обнаружении недостатка памяти необходимо корректно выйти из создавшегося положения.

## 1.7 Динамические массивы объектов и массивы указателей на объекты

По требованию решаемой задачи из динамических объектов могут создаваться массивы. Это можно сделать тремя способами:

<sup>35</sup><sub>17</sub>создать динамический массив объектов – память под него выделяют одним непрерывным фрагментом равным объему всех объектов массива (рис. 1.2, *а*), например:

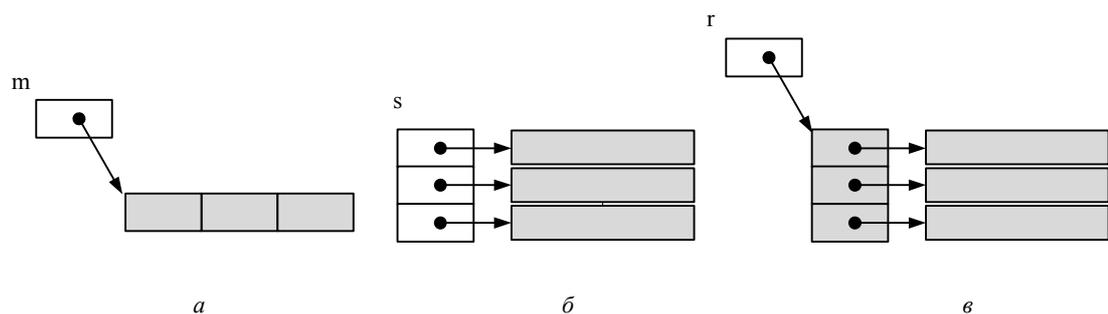
```
B mas[] = new B[n];
```

<sup>35</sup><sub>17</sub>создать статический массив указателей на объекты и затем динамически выделить память под элементы (рис. 1.2, *б*), например:

```
B *mas[n]; // память под массив указателей выделена статически
for (i=0; i<n; i++) mas[i]=new B; // выделение памяти под объекты
```

<sup>35</sup><sub>17</sub>создать динамический массив указателей и затем также динамически выделить память под элементы (рис. 1.2, *в*), например:

```
B **mas=new B *[n]; // память под массив указателей выделена динамически
for (i=0; i<n; i++) mas[i]=new B; // выделение памяти под объекты
```



**Рис. 1.2.** Три способа организации динамических массивов объектов (серым выделены элементы структуры, размещенные в динамической памяти):

*а* – динамический массив объектов; *б* – статический массив указателей на динамически размещаемые объекты; *в* – динамический массив указателей на динамически размещаемые объекты

Освобождать выделенную память нужно *так, как она была выделена*:

<sup>35</sup><sub>17</sub> одним фрагментом – для динамического массива объектов:

```
delete[] mas;
```

<sup>35</sup><sub>17</sub> поэлементно – для массива указателей на объекты:

```
for (i=0; i<n; i++) delete mas[i];
```

<sup>35</sup><sub>17</sub> одним фрагментом, если память под массив указателей выделялась динамически:

```
delete [] mas;
```

Различия между статическими и динамическими, инициализированными и неинициализированными объектами и массивами из них существенно влияют на синтаксис их описания и на работу с ними, поэтому эти различия важно хорошо понимать.

**Пример 1.15.** Определим класс `Point`, включающий два скрытых поля, инициализирующий и неинициализирующий конструкторы, инициализирующий метод и метод вывода содержимого полей на экран:

```
#include <iostream>
class Point
{
private: int x,y;
public: Point(){}
        Point(int ax,int ay): x(ax),y(ay){}
        void SetPoint(int ax,int ay){ x=ax; y=ay; }
        void Print(){ std::cout<<x<<" "<<y<<"\n"; }
};
```

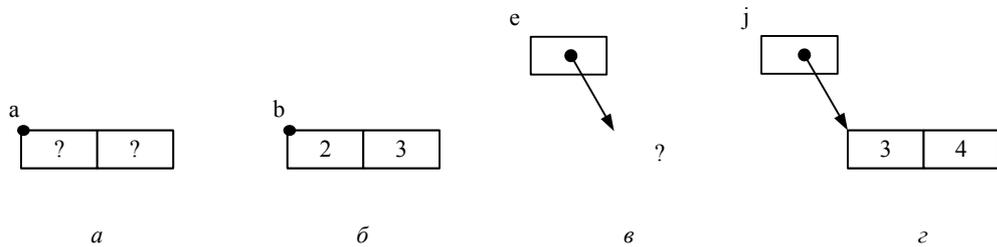
А теперь создадим объекты и массивы объектов различных типов.

### **1. Статические и динамические объекты.**

*А. Неинициализированный статический объект* (см. рис. 1.3, а):

```
Point a;           // объявление объекта
a.SetPoint(5,10); // инициализация полей
a.Print();        // вывод содержимого полей на экран
```

Поскольку объект – статический память специально выделять и освобождать не надо, эта операция будет выполнена автоматически.



**Рис. 1.3.** Результаты операций объявления статических ( $a$ – $b$ ) и динамических ( $e$ – $z$ ) объектов:

$a$  – статический неинициализированный объект;  $b$  – статический инициализированный объект;  $e$  – указатель на несозданный динамический объект;  $z$  – указатель на динамический инициализированный объект

**Б. Инициализированный статический объект** (см. рис. 1.3,  $b$ ):

```
Point b(2, 3); // создание объекта
b.Print(); // вывод содержимого полей на экран
```

Объект сразу создается инициализированным, проблем с памятью также нет.

**В. Неинициализированный динамический объект** (см. рис. 1.3,  $e$ ):

```
Point *e; // объявление неинициализированного указателя на объект
e=new Point(3, 4); // выделение памяти и инициализация полей
e->Print(); // вывод содержимого полей на экран
delete e; // освобождение памяти
```

Под объект необходимо отдельно выделить память, соответственно ее следует и освободить.

**Г. Инициализированный динамический объект** (см. рис. 1.3,  $z$ ):

```
Point *j=new Point(3, 4); /* объявление указателя на объект,
                           выделение памяти и инициализация полей объекта */
j->Print(); // вывод содержимого полей на экран
delete j; // освобождение памяти
```

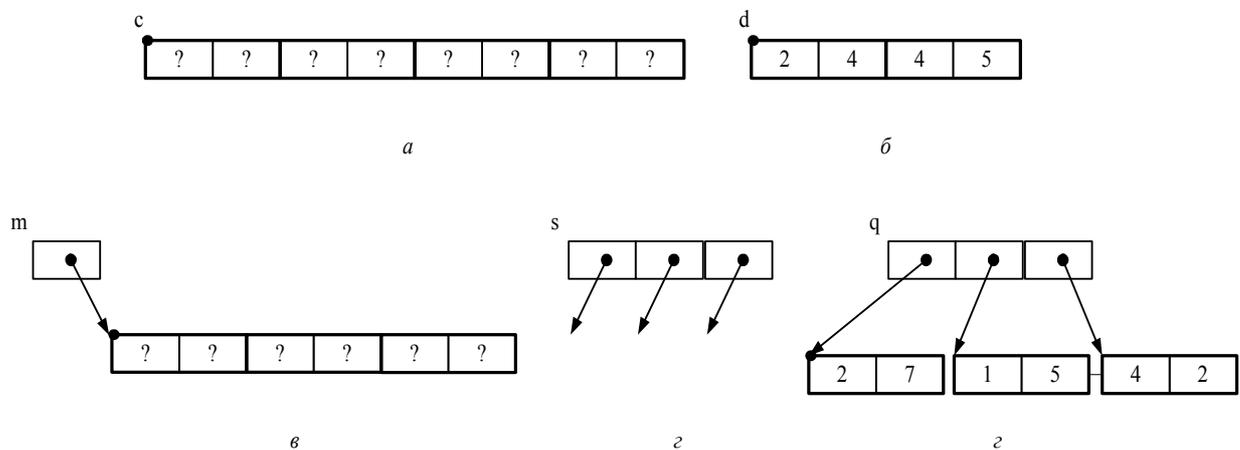
Обработка инициализированного объекта ничем не отличается от обработки неинициализированного объекта, просто операции объявления указателя, выделения памяти и инициализации полей объекта совмещены в одном операторе.

## 2. Статические и динамические массивы объектов.

А. Неинициализированный статический массив объектов (см. рис. 1.4, а):

```
Point c[4];          // объявление массива объектов
for(int i=0;i<4;i++)
{
    c[i].SetPoint(i*i,i-5); // инициализация полей
    c[i].Print();        // вывод содержимого полей на экран
}
```

Поскольку массив объявляется неинициализированным, его объекты приходится инициализировать, после чего содержимое полей можно выводить. Проблем с памятью нет.



**Рис. 1.4.** Результаты выполнения операции объявления массивов объектов:

а – неинициализированный статический массив объектов; б – инициализированный статический массив объектов; в – неинициализированный динамический массив объектов; z – неинициализированный статический массив указателей на объекты

Б. Инициализированный статический массив объектов (см. рис. 1.4, б):

```
Point d[2]= {Point(2,4),Point(4,5)}; /* создание массива
    объектов и инициализация их полей */
for(i=0;i<2;i++) d[i].Print(); // вывод содержимого полей на экран
```

Массив создается инициализированным, содержимое полей можно сразу выводить. Проблем с памятью нет.

В. Неинициализированный динамический массив объектов (рис. 1.4, в):

```
Point *m=new Point[3];
for(i=0;i<3;i++)
```

```

{
    m[i].SetPoint(i, i+1);
    m[i].Print();
}
delete [] m;

```

Память под динамический массив выделяется одним куском, следовательно и освобождать ее надо также.

Г. *Неинициализированный статический массив указателей на объекты* (рис. 1.4, з):

```

Point *s[3];
for(i=0; i<3; i++)
{
    s[i]=new Point(i, i+1);
    s[i]->Print();
}
for(i=0; i<3; i++) delete s[i];

```

В данном случае сам массив указателей – статический, т.е. память под него заказывать не надо. Память следует выделить под каждый из объектов, а в завершении работы выделенную память необходимо освободить.

Д. *Инициализированный статический массив указателей на объекты* (рис. 1.4, д):

```

Point *q[]={new Point(2, 7), new Point(1, 5), new Point(4, 2)};
for(i=0; i<3; i++)
{
    q[i]->Print();
}
for(i=0; i<3; i++) delete q[i];

```

В этом случае память под объекты также выделяется динамически, а их адреса заносятся в статический массив указателей. Освобождение памяти выполняется в цикле отдельными кусками так, как она выделялась.

**Пример 1.16.** Обработка массивов динамических объектов.

Рассмотрим программу, создающую массив из пяти динамических объектов, доступных через массив указателей на эти объекты, и массив из трех динамических объектов, доступных через указатель на первый объект массива.

```

#include <locale.h>
#include <string.h>
#include <iostream>
using namespace std;

```

```

class sstr
{
private:   char str[40];
public:   int x,y;
         void print(void)
         {
             cout<<"Содержимое полей : "<< endl;
             cout<<"x= "<<x<<" y= "<<y<<" str= "<<str<<endl;
         }
         sstr() {} // неинициализирующий конструктор
         sstr(int vx,int vy,char *vs="Строка по умолчанию")
         { setstr(vx,vy,vs); }
         ~sstr() {} // деструктор
         void setstr(int ax,int ay,char *vs); /* инициализация
                                                полей объекта */
};

void sstr::setstr(int ax,int ay,char *vs)
{
    if (strlen(vs)>=40)
    { strncpy(str,vs,40); str[40]='\0'; }
    else strcpy(str,vs);
    x=ax; y=ay;
}

void main()
{
    setlocale(0,"russian");
    sstr *a[5], // массив указателей на пять динамических объектов
        *c;    // указатель на массив динамических объектов
    int i,j;
    char *vs="Строка"; // выделить память и инициализировать объект
    for(int i=0;i<5;i++) // создать массив динамических объектов
        a[i]=new sstr(10+i,10+2*i,"aaaa"+i);
    cout<<" Массив объектов a"<<endl;
    for(i=0;i<5;i++) // вывести содержимое полей объектов

```

```
{
    cout<<"Элемент "<< i+1;
    a[i]->print();
}
for(i=0;i<5;i++) delete a[i];      // освободить память
c=new sstr[3];      // выделить память под три динамических объекта
for(i=0;i<3;i++)    // инициализировать поля динамических объектов
    (c+i)->setstr(15+i,12+i*2,vs+i);
cout<<" Массив объектов c"<<endl;
for(i=0;i<3;i++)    // вывести содержимое полей объектов
{
    cout<<"Элемент "<< i+1;
    (c+i)->print();
}
delete []c; // освободить память
system("pause");
}
```

### Вопросы для самоконтроля

1. Что такое класс в С++? Как выполнить описание класса?

[Ответ.](#)

2. Какие существуют способы ограничения доступа к компонентам класса? Как и где они используются? Чем отличается описание компонентных функций внутри и вне определения класса?

[Ответ.](#)

3. Сформулируйте особенности конструкторов и деструкторов классов С++. Что такое неинициализирующий конструктор и чем он отличается от конструктора без параметров? Когда использование конструктора, вызываемого без аргументов, необходимо?

[Ответ.](#)

4. Что такое копирующий конструктор? Назовите случаи, когда использование такого конструктора обязательно.

[Ответ.](#)

5. Какие сложности возникают при работе с динамическими объектами?

[Ответ.](#)

## 2 Построение иерархии классов

### 2.1 Наследование

*Наследованием* называют конструирование новых более сложных *производных* классов (классов-потомков) из уже имеющихся *базовых* классов (классов-родителей) посредством добавления полей и методов. Это – эффективное средство расширения функциональных возможностей существующих классов без их перепрограммирования и повторной компиляции существующих программ.

По определению компонентами производного класса являются:

<sup>35</sup><sub>17</sub> компоненты базового класса, за исключением конструктора, деструктора и компонентной функции, переопределяющей операцию «присваивания» (=) (см. раздел 5.4);

<sup>35</sup><sub>17</sub> компоненты, добавляемые в теле производного класса.

В функциональном смысле производные классы являются более мощными по отношению к базовым классам, так как, включая поля и методы базового класса, они обладают еще и своими компонентами.

Ограничение доступа к полям и функциям базового класса при наследовании осуществляется с помощью специальных описателей, определяющих вид наследования:

```
class <Имя производного класса >:
```

```
    <Вид наследования><Имя базового класса>{<Тело класса>;
```

где вид наследования определяется ключевыми словами: `private`, `protected`, `public`.

Видимость полей и функций базового класса из производного определяется секцией, в которой находится объявление компонента и видом наследования (см. табл. 2.1).

Таблица 2.1. Видимость компонентов базового класса в производном

Вид наследования	Объявление компонентов в базовом классе	Видимость компонентов в производном классе
private	private	не доступны
	protected	private
	public	private
protected	private	не доступны
	protected	protected
	public	protected
public	private	не доступны
	protected	protected
	public	public

Если вид наследования явно не указан, то по умолчанию принимается `private`. Однако хороший стиль программирования требует, чтобы в любом случае вид наследования был задан явно.

Согласно таблице 2.1 самым простым является наследование вида `public` (общедоступное), однако при его применении следует помнить, что скрытые в секции `private` компоненты базового класса в производном все равно недоступны. Соответственно для обращения к скрытым полям базового класса должны использоваться методы общедоступной или защищенных секций базового класса, например:

```
class A
{
    int x;
public:
    void set_a(int ax){x=ax;}
    print_a(){cout<<x;}
};
class B:public A
{
    int y;
public:
    void set_b(int ax,int bx)
    {
        set_a(ax); // доступ к скрытому полю x
        b=bx;
    }
    print_b()
```

```

    {
        print_a(); // доступ к скрытому полю x
        cout<<y;
    }
};

```

**Конструкторы и деструкторы производных классов.** Как уже упоминалось раньше в C++ *конструкторы и деструкторы базового класса в производных классах не наследуются*. Однако если базовый класс содержит хотя бы один конструктор и деструктор, то производный класс также должен включать собственные конструктор и деструктор. При этом C++ поддерживает определенные правила взаимодействия между этими компонентами базовых и производных классов.

При создании объектов производного класса предусмотрен *автоматический вызов конструктора базового класса* для инициализации его полей. Однако следует помнить, что по умолчанию осуществляется вызов конструктора базового класса *без параметров*. Если такой конструктор в базовом классе отсутствует, то компилятор выдает сообщение об ошибке `error C2512`.

Поскольку явный вызов конструктора базового класса в программе невозможен, чтобы передать этому конструктору аргументы для инициализации полей базового класса, следует:

<sup>35</sup><sub>17</sub> добавить соответствующие параметры к собственным параметрам конструктора производного класса;

<sup>35</sup><sub>17</sub> вызвать конструктор базового класса *в списке инициализации* конструктора производного класса, передав ему соответствующие аргументы.

```

class A
{
    int x;
public:
    A(int ax):x(ax) {} // конструктор базового класса
};
class B
{
    int y;

```

```
public:
    B(int ax,int ay):A(ax),y(ay) {} // конструктор производного класса
};
```

При этом соблюдается строгий порядок конструирования полей базового и производного классов: не зависимо от порядка указания в списке инициализации сначала вызывается конструктор базового класса, а затем – конструкторы полей, объявленных в производном классе.

**Пример 2.1.** Порядок работы конструкторов базового и производного классов.

```
#include <iostream>
using namespace std;
class A
{
public:    int a;
    A(int v):a(v) {}
    void printa() { cout<<a<<endl; }
};
class B: public A
{
public:    int b;
    B(int va,int vb):A(va),b(vb) {}
    void printb(void) { cout<<a<<" "<<b<<endl; }
};
class C: public B
{
public:    int c;
    C(int va,int vb,int vc):B(va,vb),c(vc) {}
    void printc(void)
    { cout<<a<<" "<<b<<" "<<c<<endl; }
};
void main()
{
    A aa(10);          // вызывается конструктор класса A
    B bb(10,100);     //вызывается конструктор класса A, а затем – B
```

```

C cc(10,100,1000); // вызываются конструкторы классов A, B и C
aa.printa();      bb.printb();    cc.printc();
system("pause");
}

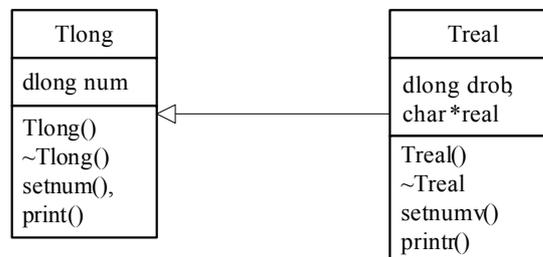
```

Деструкторы (описанные в программе или созданные автоматически) будут вызываться в порядке, обратном порядку вызова конструкторов.

**Пример 2.2.** Проектирование классов с использованием наследования (классы Целое число и Вещественное число).

Пусть требуется разработать классы для реализации объектов Целое число и Вещественное число. Объект Целое число должен хранить длинное целое в десятичной записи и уметь выводить его значение. Объект Вещественное число должен хранить вещественное число, задаваемое в виде `sssss.ddddd`, и его символьное представление. Он также должен уметь выводить свое значение на экран.

Для обоих объектов необходимо предусмотреть возможность инициализации как в момент объявления переменной, так и в процессе функционирования. Поскольку вещественное число включает длинное целое как целую часть, класс для его реализации можно наследовать от класса, реализующего длинное целое число (рис. 2.1).



**Рис. 2.1.** Наследование классов

```

#include <locale.h>
#include <stdlib.h>
#include <iostream>
#include <string.h>
using namespace std;
typedef unsigned long dlong;
class Tlong // Класс Целое число
{
public:

```

```

    dlong num;          // числовое поле класса
    Tlong() {}         // неинициализирующий конструктор
    Tlong(dlong an) : num(an) {} // конструктор
    ~Tlong() {}       // деструктор
    void print(void)   // вывод значения поля
    { cout<<" Целое число : "<<num<<endl; }
    void setnum(dlong an) // инициализации поля
    { num=an; }
};

class Treal: public Tlong // Класс Вещественное число
{
public:
    dlong drob;        // дробная часть числа
    char *real;        // запись вещественного числа
    Treal() {}         // неинициализирующий конструктор
    Treal(char *st) : Tlong() // инициализирующий конструктор
    { setnumv(st); }
    ~Treal()           // деструктор
    { delete real; }
    void printr();     // вывод вещественного числа
    void setnumv(char * st); // инициализация полей класса
};

void Treal::setnumv(char * st)
{
    int l=strlen(st); char *ptr;
    real=new char[l+1]; strcpy(real,st);
    ptr=strchr(real, '.'); *ptr='\0';
    drob=dlong(atol(ptr+1));
    num=dlong(atol(real));
    *ptr='.';
}

void Treal::printr()
{
    cout<<"Вещественное число: "<<real<<endl;
}

```

```

    cout<<"Целая часть: "; print();
    cout<<"Дробная часть: "<<drob<<endl;
}
void main ()
{
    setlocale(0,"russian");
    Treal a("456789.1234321"), // объект производного класса
        *pa=new Treal("456789.1234321"), // указатель
    mask[3]= // инициализированный массив объектов
        {Treal("1748.5932"),
         Treal("4567.34321"),
         Treal("18689.9421")};
    a.printr();
    pa->printr();    delete pa;
    for(int i=0;i<3;i++)
    {
        cout<<"Элемент массива "<<(i+1)<<": "<<endl;
        mask[i].printr();
    }
    system("pause");
}

```

### **Управление видимостью компонентов базового класса в производном.**

Производный класс может получить доступ к компонентам секций `protected` и `public` базового класса даже при самом жестком виде наследования `private`. Для этого используют квалификатор доступа `<Имя класса>::<Имя компонента>`. Квалификатор следует поместить в ту же секцию, в которой необходимый компонент был описан в базовом классе.

**Пример 2.3.** Получение производным классом доступа к защищенным и общедоступным компонентам базового класса при наследовании `private`.

```

#include <iostream>
using namespace std;
class A

```

```

{
private:      int x;
protected:  int y;
public:      int z;
    void print(void)
    {
        cout<<"x = "<<x<<endl;
        cout<<"y = "<<y<<endl;
        cout<<"z = "<<z<<endl;
    }
    A(){x=20; y=30; z=50;}
};

class B: private A    // все компоненты класса A не доступны в классе B
{
protected: A::y; /* защищенное поле класса A,
                 объявляется доступным в классе B */
public:     A::z; /* общее поле класса A,
                 объявляется доступным в классе B */
    B():A(){}
    void print(void)
    {
        cout<<"y = "<<y<<endl;
        cout<<"z = "<<z<<endl;
    }
};

void main()
{
    A aa; // создание объекта базового класса
    B bb; // создание объекта производного класса
    aa.print(); // выводит: x = 20 y = 30 z = 50
    bb.print(); // выводит: y = 30 z = 50
    system("pause");
}

```

## 2.2 Множественное наследование

Язык C++ позволяет осуществлять наследование не только от одного, но и одновременно от нескольких классов. Такое наследование получило название *множественного*. Описание производного класса при множественном наследовании выглядит следующим образом:

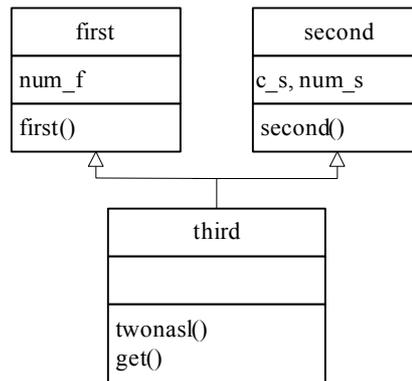
```
class <Имя производного класса>:
    <Вид наследования><Имя базового класса 1>,
    <Вид наследования><Имя базового класса 2>,
    ...
    <Вид наследования><Имя базового класса n> {...};
```

Вид наследования, как и в случае простого наследования, определяет режим доступа к компонентам соответствующего базового класса. Если конструкторы базовых классов не имеют аргументов, то производный класс может не иметь конструктора. При наличии у конструкторов базового класса аргументов производный класс обязан иметь конструктор со списком инициализации следующего вида:

```
<Имя конструктора производного класса>(<Список аргументов>):
    <Имя конструктора базового класса 1>(<Список аргументов 1>),
    .....
    <Имя конструктора базового класса n>(<Список аргументов n>)
    {<Тело конструктора производного класса>}
```

Последовательность активизации конструкторов такая же, как и для случая единственного базового класса: сначала активизируются конструкторы всех базовых классов в порядке их перечисления в объявлении производного класса, затем конструкторы объектных полей и в завершении – конструктор производного класса. Соответственно поля базовых классов создаются в том порядке, в котором эти поля перечислены при объявлении производного класса.

**Пример 2.4.** Наследование от двух базовых классов. В данном примере класс `third` наследуется от классов `first` и `second` (рис. 2.2).



**Рис. 2.2.** Множественное наследование

Объект класса `third` состоит из следующих полей:

<sup>35</sup><sub>17</sub> поля `num_f`, унаследованного от класса `first` (описанного `public`, наследованного `private`, следовательно доступного `private`) и инициализированного случайным числом в диапазоне от -50 до 49;

<sup>35</sup><sub>17</sub> полей `num_s` и `c_s`, унаследованных от класса `second` (описанных `public`, наследованных `public`, следовательно доступного `public`) и инициализированных числами 20 и символом «R», причем инициализация поля `c_s` в конструкторе класса `second` не предусмотрена, поэтому она выполняется в теле конструктора класса `third`.

```

#include <locale.h>
#include <iostream>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
using namespace std;
class first
{
public:    int num_f;
    first(int va):num_f(va)
    { cout<<"Конструктор first\n"; }
};
class second
{
public:    char c_s;    int num_s;
    second(int vn):num_s(vn)
    { cout<<"Конструктор second\n"; }
};
  
```

```

};
class third:private first, // сокрытие базового класса
             public second // открытый базовый класс
{
public:
    third(int nm,char vc,int nfx):
        first(nfx), second(nm)
    {
        cout<<"Конструктор third\n";
        c_s=vc; /* инициализация поля базового класса
                конструктором производного класса */
    }
    int get(void) // получение значения внутреннего поля
    { return num_f; }
};

void main()
{
    setlocale(0,"russian");
    srand((unsigned)time(NULL)); // инициализация датчика случайных чисел
    int r=rand()/1000-rand()/1000; // присвоение числу случайного значения
    third aa(r,'R',50);
    cout<<aa.get()<<" "<<aa.c_s<<" "<<aa.num_s<<endl;
    _getch();
}

```

В результате выполнения программы мы получаем следующую цепочку обращений к конструкторам:

```

Конструктор first
Конструктор second
Конструктор third

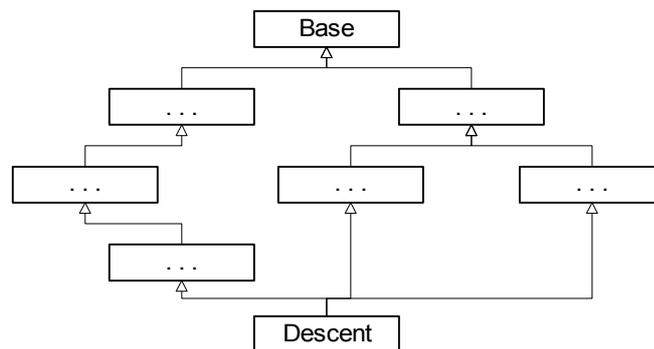
```

и содержимое полей num\_f, c\_s и num\_s, доступных в классе third:

50 R -4

### 2.3 Виртуальное наследование

При множественном наследовании базовый класс не может быть указан в производном классе более одного раза. Однако возможна ситуация, когда производный класс при наследовании от потомков одного базового класса многократно наследует одни и те же компоненты базового класса (рис. 2.3). Иными словами, производный класс будет содержать несколько копий полей одного базового класса.



**Рис. 2.3.** Многократное наследование полей базового класса

Чтобы избежать многократного включения в производный класс компонентов базового класса, используется *виртуальное наследование*. При виртуальном наследовании производный класс описывают следующим образом:

```
class <Имя производного класса>:
    virtual <Вид наследования><Имя базового класса>{...};
```

В этом случае включение в производный класс полей базового класса осуществляется один раз, а их инициализация выполняется в производном классе, который не является прямым потомком базового класса. Вызов конструкторов при этом происходит в следующем порядке: сначала конструктор виртуально наследуемого базового класса, затем конструкторы базовых классов в порядке их перечисления при объявлении производного класса, за ними – конструкторы объектных полей и конструктор производного класса. Деструкторы соответственно вызываются в обратном порядке.

Виртуально наследуемый класс *обязательно должен содержать конструктор без параметров*, который активизируется при выполнении конструкторов классов – прямых потомков виртуально наследуемого класса.

**Пример 2.5.** Виртуальное наследование. Реализуем иерархию классов, представленную на рис. 2.3. Класс `derived` наследуется от двух наследников класса `fixed`. Чтобы исключить удваивание полей, описанных в классе `fixed`, необходимо использовать виртуальное наследование.

```
#include <locale.h>
#include <iostream>
using namespace std;
class fixed
{
protected:  int Fix;
public:
    fixed() // конструктор без параметров
    { cout<<" Конструктор класса fixed\n"; }
    fixed(int  fix):Fix(fix) // конструктор с параметром
    { cout<<" Конструктор класса fixed int\n"; }
};
class derived_1: virtual public fixed // виртуальное наследование
{
public:      int One;
    derived_1(void)
    { cout<<" Конструктор класса derived 1\n"; }
};
class derived_2: virtual private fixed // виртуальное наследование
{
public:      int Two;
    derived_2(void)
    { cout<<" Конструктор класса derived 2\n"; }
};
class derived: public derived_1, public derived_2
    /* объявление производного класса – непрямого потомка */
{
public:
    derived(void)
```

```

    { cout<<" Конструктор класса derived() \n";}
    derived(int fix):fixed(fix)
    { cout<<" Конструктор класса derived (int) \n";}
    void Out( )
    { cout<<" Значение поля  Fix = "<< Fix<<endl;}
};
void main()
{
    setlocale(0,"russian");
    derived Var(10);
    Var.Out( );
    system("pause");
}

```

В результате работы программы получаем

```

Конструктор класса fixed  int
Конструктор класса 1
Конструктор класса 2
Конструктор класса derived (int)
Значение поля Fix=10

```

Если бы наследование не было виртуальным, поле Fix было бы включено в объект класса derived дважды:

```

derived_1::Fix      и      derived_2::Fix.

```

## 2.4 Простой полиморфизм

В языке C++ предусмотрен механизм полиморфизма, обеспечивающий возможность определения разных описаний некоторого единого по названию метода для классов различных уровней иерархии. При этом различают *простой полиморфизм*, базирующийся на механизме раннего связывания, и *сложный полиморфизм*, использующий механизм позднего связывания.

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

Подобный подход позволяет строить более гибкие и совершенные иерархии классов, переопределяя в производных классах методы в соответствии с требованиями разрабатываемой программы или системы, использующей эти классы.

**Пример 2.6.** Использование раннего связывания. Для демонстрации применения статического полиморфизма вернемся к примеру 2.2. В этом примере функции `print()` и `printr()` выполняют одинаковые по смыслу операции – выводят главные поля объекта на экран, следовательно, их можно назвать одним именем, например `print()`. Метод `print()`, таким образом, станет полиморфным – переопределяемым в производном классе. Отдельное определение метода в своем классе называют *аспектом* полиморфного метода.

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <string.h>
using namespace std;
typedef unsigned long dlong;
class Tlong          // Класс Целое число
{
```

```

public:
    dlong num;          // числовое поле класса
    Tlong() {}         // неинициализирующий конструктор
    Tlong(dlong an) : num(an) {} // конструктор
    ~Tlong() {}       // деструктор
    void print(void)   // вывод значения поля
    { cout<<" Целое число : "<<num<<endl; }
    void setnum(dlong an) // инициализации поля
    { num=an; }
};

class Treal: public Tlong // Класс Вещественное число
{
public:
    dlong drob;        // дробная часть числа
    char *real;        // запись вещественного числа
    Treal() {}        // неинициализирующий конструктор
    Treal(char *st) : Tlong() // конструктор
    { setnumv(st); }
    ~Treal()          // деструктор
    { delete real; }
    void print();    // вывод вещественного числа (переопределяется)
    void setnumv(char * st); // инициализация полей класса
};

void Treal::setnumv(char * st)
{
    int l; char *ptr;
    l=strlen(st); real=new char[l+1]; strcpy(real,st);
    ptr=strchr(real, '.'); *ptr='\0';
    drob=dlong(atol(ptr+1));
    num=dlong(atol(real));
    *ptr='.';
}

void Treal::print()
{

```

```
    cout<<"Вещественное число: "<<real<<endl;
    cout<<"Целая часть: "; Tlong::print();
    cout<<"Дробная часть: "<<drob<<endl;
}
void main ()
{
    setlocale(0,"russian");
    Tlong i(174832); // простой объект базового класса
    i.print(); // явный вызов переопределяемого метода
    Treal a("1748.5932"); // простой объект производного класса
    a.print(); // явный вызов переопределенного метода
    system("pause");
}
```

В рассмотренном примере требуемый аспект полиморфного метода определяется по типу объекта, для которого он вызывается.

## 2.5 Сложный полиморфизм

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

**Пример 2.7.** Для демонстрации ошибок, возникающих при некорректном использовании простого полиморфизма введем в описание базового класса предыдущего примера новый метод `show()`. Этот метод будет вызывать статический полиморфный метод `print()` и наследоваться в производных классах. Кроме этого добавим в программу внешнюю функцию `show_ext()` с параметром – ссылкой на базовый класс, чтобы показать особенности раннего связывания.

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <string.h>
using namespace std;
typedef unsigned long dlong;
class Tlong          // Класс Целое число
{
public:
    dlong num;        // числовое поле класса
    Tlong() {}        // неинициализирующий конструктор
    Tlong(dlong an) : num(an) {} // конструктор
    ~Tlong() {}       // деструктор
    void print(void)  // вывод значения поля
    { cout<<" Целое число : "<<num<<endl; }
    void setnum(dlong an) // инициализации поля
    { num=an; }
    void show()      // метод, вызывающий переопределяемый метод
```

```

        { print(); }
};
class Treal: public Tlong    // Класс Вещественное число
{
public:
    dlong drob;           // дробная часть числа
    char *real;          // запись вещественного числа
    Treal() {}           // неинициализирующий конструктор
    Treal(char *st) :Tlong() // конструктор
    { setnumv(st); }
    ~Treal()             // деструктор
    { delete real; }
    void print();       // вывод вещественного числа (переопределяется)
    void setnumv(char * st); // инициализация полей класса
};
void Treal::setnumv(char * st)
{
    int l; char *ptr;
    l=strlen(st); real=new char[l+1]; strcpy(real,st);
    ptr=strchr(real, '.');
    drob=dlong(atol(ptr+1)); *ptr='\0';
    num=dlong(atol(real));
    *ptr='.';
}
void Treal::print()
{
    cout<<"Вещественное число: "<<real<<endl;
    cout<<"Целая часть: "; Tlong::print();
    cout<<"Дробная часть: "<<drob<<endl;
}
// Внешняя функция с параметром - ссылкой на базовый класс Tlong
void show_ext(Tlong &par)
{ par.print(); }
void main ()

```

```

{
    setlocale(0, "russian");
    Tlong i(174832); // простой объект базового класса
    i.show(); // косвенный вызов переопределяемого метода класса
    Treal a("1748.5932"); // простой объект производного класса
    a.show(); // выводит только целую часть числа (ошибка!)
    Treal *pa=new Treal("456789.1234321"); /* указатель на объект
                                           производного класса */
    pa->print(); // явный вызов переопределенного метода
    pa->show(); // выводит только целую часть числа (ошибка!)
    delete pa; // вызывает деструктор производного класса
    Tlong *pb=new Treal("234567.34765");/* указатель
                                           базового класса, объект производного класса */
    pb->print(); // выводит только целую часть числа (ошибка!)
    delete pb; // неявно вызывается деструктор класса Tlong (ошибка!)
    show_ext(a); // выводит только целую часть числа (ошибка!)
    system("pause");
}

```

Сравнение примеров 2.6 и 2.7 показывает, что результаты явного и опосредованного вызовов метода `print()` различаются. При явном вызове для переменных базового и производного класса, а также, если указатели и объекты совпадают по типу, никаких проблем не возникает: вызывается аспект метода класса, к которому принадлежит объект. Опосредованный вызов может приводить к ошибкам. Это объясняется следующими причинами.

Во-первых, при раннем связывании метод `show()` жестко соединяется с методом `print()` базового класса на этапе компиляции. Следовательно, из метода `show()` всегда будет вызываться метод `print()` базового класса, для которого поля производных классов не доступны.

Во-вторых, при выполнении стандартного преобразования указателя производного класса в указатель на базовый, поля и методы производного класса становятся невидимыми для указателя на объекты базового класса, поэтому обращение к методу `print()` приведет к вызову одноименного метода базового класса.

В-третьих, внешняя функция `show_ext()` описана с формальным параметром – ссылкой на объект базового класса. По правилам синтаксиса в качестве фактического параметра при вызове такой функции ей можно передать имя объекта производного класса. При этом передача параметра осуществляется по адресу, который становится известен только на этапе выполнения, а увязка всех адресов определяется на этапе компиляции, поэтому реально в функции будет реализовано обращение к методу `print()` базового класса.

Во всех трех случаях объект, для которого вызывается метод, определяется адресом, хранящемся в указателе, который по типу может не совпадать с типом объекта:

**1-й случай** – *если наследуемый метод для объекта производного класса вызывает метод, переопределенный в производном классе* – в этом случае вызывающий метод определен в базовом классе, поэтому компилятор считает, что и аспект полиморфного метода необходим базового класса, что не корректно для случая, когда наследуемый метод вызван для объекта производного класса.

**2-й случай** – *если объект производного класса через указатель базового класса обращается к методу, переопределенному производным классом* – в этом случае тип объекта компилятор определяет по типу указателя, соответственно подключая аспект полиморфного метода базового класса, хотя реально объект может принадлежать производному классу.

**3-й случай** – *если процедура вызывает переопределенный метод для объекта производного класса, переданного в процедуру через параметр-переменную, описанный как объект базового класса («процедура с полиморфным объектом»)* – в этом случае требуемый аспект полиморфного метода определяется типом параметра переменной, что не корректно, если в качестве аргумента в процедуре передается объект производного класса.

Фиксация адреса метода на этапе компиляции является отличительной чертой раннего связывания. Поэтому в трех перечисленных случаях для получения правильного результата необходимо использование сложного полиморфизма, который реализуется механизмом позднего связывания, позволяющего выбирать требуемый аспект полиморфного метода уже на этапе выполнения, когда тип (класс) объекта, для которого вызывается метод точно известен.

Реализации сложного полиморфизма основана на применении виртуальных функций.

*Виртуальными* называют функции, которые объявлены с использованием ключевого слова `virtual` в базовом классе и переопределяются (замещаются) в одном или нескольких производных классах. При этом прототипы функций в разных классах должны совпадать не только по именам, но и по сигнатуре, хотя алгоритмы, реализуемые такими функциями, различны. Если прототипы функций не совпадают, то механизм виртуальности для них не включается.

Виртуальную функцию в C++ принято называть «полиморфной», что не совсем соответствует общепринятой терминологии, согласно которой она является «динамической полиморфной», в отличие от статических полиморфных функций, которые в C++ принято называть просто «переопределенными».

При использовании виртуальных функций нужный аспект полиморфной функции, вызываемой из метода базового класса или через указатель на объекты базового класса, определяется на этапе выполнения, когда известно, для какого объекта вызван метод: объекта базового класса или объекта производного.

Вызов виртуальной функции реализуется как косвенный вызов по таблице виртуальных методов (ТВМ), а потому требует больше времени. ТВМ автоматически создается во время компиляции, а затем используется во время выполнения программы.

Класс, который содержит хотя бы одну виртуальную функцию, называется *полиморфным*, и, соответственно, объект такого класса тоже является *полиморфным*.

**Пример 2.8.** Использование позднего связывания. Для исправления ошибок предыдущего примера необходимо объявить функцию `print()` и деструктор класса `Tlong` виртуальными. Для чего следует добавить в описание этих функций ключевое слово `virtual`:

```
virtual ~Tlong()
{
    cout<<"Виртуальный деструктор класса Tlong"<<endl;
}
virtual void print(void);
```

Тогда нужный аспект полиморфной функции будет определяться на этапе выполнения и программа будет работать верно:

```
void main ()
```

```

{
    setlocale(0, "russian");
    Treal a("1748.5932"); // простой объект производного класса
    a.show(); // вызывает метод Print() производного класса
    Treal *pa=new Treal("456789.1234321"); /* указатель и объект
                                           производного класса */
    pa->show(); // вызывает метод Print() производного класса
    delete pa; // вызывает деструктор производного класса
    Tlong *pb=new Treal("234567.34765"); /* указатель
                                           базового класса, объект производного класса */

    pb->print(); // вызывает метод Print() производного класса
    delete pb; // вызывает деструктор производного класса
    show_ext(a); // вызывает метод Print() производного класса
    system("pause");
}

```

Функция, объявленная виртуальной, остается таковой, сколько бы производных классов не строилось. Однако иногда в одном или нескольких производных классах переопределение виртуальной функции может отсутствовать. В этом случае механизм подключения виртуальной функции сохраняется, а, следовательно, вызывается функция базового класса, ближайшего к рассматриваемому. Отсутствие аспекта виртуальной функции не нарушает механизма позднего связывания и, если у наследников такого класса появится аспект виртуальной функции, он будет вызван без каких-либо проблем.

Виртуальная функция обязательно должна быть компонентом некоторого класса. Она может быть объявлена дружественной другому классу, но не может быть объявлена статической (см. главу 4).

## 2.6 Абстрактные функции и классы

Практика объектно-ориентированного программирования показывает, что иногда при разработке сложных иерархий классов целесообразно описывать классы, которые выражают некоторую общую концепцию, описывают основной интерфейс для использования в производных классах. Такой класс включает определение данных и методов, которые будут общими для различных производных классов, и составляет некоторую базу, от которой наследуются другие классы иерархии. Смыслового определения виртуальной функции в подобном базовом классе может не быть, но без ее объявления корректная реализация сложной иерархии затруднена. Такие функции в языке C++ описывают как абстрактные виртуальные.

*Абстрактной виртуальной* называется функция, объявленная в базовом классе как виртуальная, но не содержащая описания выполняемых действий. Для объявления абстрактной виртуальной функции используется следующая форма:

```
virtual <Тип><Имя_функции>(<Список параметров>) = 0;
```

Здесь присваивание нулю – признак абстрактной виртуальной функции. При этом производный класс должен определить свою собственную версию функции, так как в базовом классе не существует версии, которую можно было бы использовать в производном.

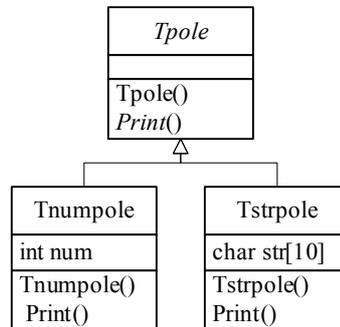
Класс, который содержит, по крайней мере, одну абстрактную виртуальную функцию, принято называть *абстрактным*. Этот класс может использоваться только как базовый для создания других классов, причем если класс, производный от абстрактного, не содержит аспекта виртуальной функции, то он также является абстрактным.

*Создание объектов абстрактного класса запрещено.* Однако можно создавать указатели на объект абстрактного базового класса и ссылки того же типа и применять их для реализации механизма полиморфизма.

**Пример 2.9.** Использование абстрактного класса при работе с полиморфными объектами.

Пусть нам необходимо организовать массив, в котором хранятся указатели на объекты двух классов: целые числа и строки. По условию задачи объекты этого массива необходимо выводить на экран.

Для того, чтобы собрать в одном массиве указатели на объекты разных классов, необходимо, чтобы эти классы имели общего предка, т.е. нам потребуется организовать иерархию классов, по типу представленной на рис. 2.4.



**Рис. 2.4.** Пример иерархии с абстрактным классом

Класс `Tpole` – общий базовый, его существование позволит нам объявить массив указателей на объекты этого класса. Сами объекты создавать не будем: в процессе работы в этот массив будут помещены указатели на объекты класса `Число` или на объекты класса `Строка`.

Для вывода на экран содержимого хранимых объектов в программе должен быть организован цикл, в котором для каждого объекта вызывается метод `Print()`. При этом вызов метода происходит через *указатель на объекты базового класса*. Если бы такой метод в базовом классе отсутствовал, то для вызова этого метода объектами производных классов тип указателя пришлось бы явно переопределять. Это связано с тем, что указатель на объекты базового класса «не видит» полей и методов, появившихся в переопределенном классе. При наличии абстрактного метода `Print()` в базовом классе этой проблемы не возникает.

Естественно в базовом классе метод `Print()` объявляется виртуальным абстрактным, поскольку в объектах базового класса на экран выводить нечего. Соответственно и класс `Tpole` получается абстрактным.

```

#include <locale.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

class Tpole // абстрактный класс Поле
{
public:

```

```

    Tpole() {}

    virtual void Print(void)=0; // абстрактная функция
};

class Tnumpole:public Tpole    // класс Число
{
    private: int num;
    public:
        Tnumpole(int n):num(n){}

        void Print(void) /* аспект виртуальной функции*/
        {   printf("Число = %5d\n",num);   }
};

class Tstrpole:public Tpole    // класс Строка
{
    private: char str[10];
    public :
        Tstrpole(char *st) {   strcpy(str,st);   }

        void Print(void) /* аспект виртуальной функции*/
        {   printf("Строка = %s\n",str);   }
};

void main()
{
    setlocale(0,"russian");
    int n,i; char st[80];
    Tpole *a[10]; // массив указателей на объекты класса Tpole
    for(i=0;i<10;i++)
    {
        printf("\nВведите целое число или строку: ");
        scanf_s("%s",st,80);
        if ((n=atoi(st))!=0||
            (strlen(st)==1 && st[0]=='0'))
            a[i]=new Tnumpole(n); // Число
        else
            a[i]=new Tstrpole(st); // Строка
    }
}

```

```
for(i=0;i<10;i++) a[i]->Print();  
for(i=0;i<10;i++) delete a[i];  
_getch();  
}
```

Программа создает 10 полиморфных объектов разных производных классов: если введено число, то объект класса Число, иначе – объект класса Строка. Затем, при выводе содержимого массива осуществляется вызов нужного аспекта виртуальной функции.

### Вопросы для самоконтроля

1. Как описывается производный класс?

[Ответ.](#)

2. Что такое множественное и виртуальное наследование?

[Ответ.](#)

3. Как определяется доступность компонентов базового класса в производном классе? Какова последовательность подключения конструкторов и деструкторов базового и производного классов?

[Ответ.](#)

4. Назовите виды полиморфизма в C++. Определите понятие виртуальных и абстрактных функций. Что такое абстрактный класс? Назовите особенности использования абстрактного класса.

[Ответ.](#)

## 3 Композиция и наполнение

### 3.1 Композиция. Объектные поля класса

Композицию и наполнение используют для включения одних объектов в другие в тех случаях, когда нецелесообразно или невозможно с той же целью применить наследование.

*Композицией* называют такое отношение между классами, при котором объекты одного являются неотъемлемой частью другого. В С++ композиция реализуется механизмом создания *объектных* полей, т.е. полей, которые являются объектами других классов. Количество таких полей может быть любым. Если объектов много и они однотипны, то включаемые объекты можно собирать в структуры, например массивы или списки.

Конструирование объектного поля как и полей базового класса по правилам С++ предполагает вызов конструктора класса создаваемого объектного поля. При этом, как и для базовых полей, автоматически будет вызван конструктор *без параметров*, отсутствие которого в классе приведет к получению сообщения об ошибке `error C2512`. Чтобы обеспечить вызов конструктора с требуемыми значениями параметров, необходимо использовать список инициализации.

**Пример 3.1.** Использование композиции для реализации включения объектов.

Для демонстрации композиции вернемся к примеру 2.2. Построим класс Вещественное число на базе класса Целое число, используя не наследование, а композицию. Это возможно, так как запись вещественного числа включает целую и дробную части, которые можно представить объектными полями класса Целое число `Tlong`. При описании класса Вещественное число считаем, что описание класса `Tlong` помещено в файл `Tlongclass.h`.

В классе предусмотрим три конструктора:

```

35
17    пустой без параметров – на случай создания неинициализированных
        объектов;
35
17    инициализирующий – создает объект по записи числа;
35
17    инициализирующий – создает объект, получая отдельно целую и дробную
        части числа.

```

В первом и втором конструкторах при создании объектных полей автоматически вызывается конструктор поля без параметров. В последнем конструкторе для конструирования инициализированных объектных полей использован вызов конструкторов в списке инициализации.

```
#include "stdafx.h"
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include "Tlongclass.h"
class Treal // Класс Вещественное число
{
private:
    Tlong celn; // объектное поле - целая часть вещественного числа
    Tlong drob; // объектное поле - дробная часть вещественного числа
    char *real; // запись вещественного числа
public:
    Treal() {} // неинициализирующий конструктор
    Treal(char *st) // инициализирующий конструктор
        { setnumv(st); }
    Treal(dlong c, dlong d): celn(c),drob(d) // конструктор
        { // со списком инициализации
            char *s1=new char[10];
            ltoa(celn.num,s1,10);
            char *s2=new char[10];
            ltoa(drob.num,s2,10);
            int len=strlen(s1)+strlen(s2);
            real=new char [len+1];
            strcpy(real,s1);
            strcat(real,s2);
        }
    ~Treal() { delete real; } // деструктор
    void printr(); // вывод числа на экран
    void setnumv(char * st); // инициализация полей класса
}
```

```

};
void Treal::setnumv(char * st)
{
    char *ptr;
    real=new char[strlen(st)+1];
    strcpy(real,st);
    ptr=strchr(real, '.');
    drob.setnum(dlong(atol(ptr+1)));    *ptr='\0';
    celn.setnum(dlong(atol(real)));    *ptr='.';
}
void Treal::printr()
{
    cout<<"Вещественное число :"<<real<<endl;
    cout<<"Целая часть :";    celn.print();
    cout<<"Дробная часть :";    drob.print();
    cout<<endl;
}
void main ()
{
    setlocale(0,"russian");
    Treal a,b("345678.45567"),d("345678.45567");
    a.setnumv("345678.45567");
    a.printr();
    b.printr();
    d.printr();
    _getch();
}

```

Нетрудно видеть, что реализация класса `Treal` с объектными полями проще, чем с использованием наследования в примере 2.3, так как в ней отсутствуют методы работы с дробной частью числа. Но по сравнению с наследованием у нее есть недостаток: *классы `Treal` и `Tlong` не входят в одну иерархию и соответственно с их объектами нельзя работать через один типизированный указатель, а потому для них не реализуется механизмы простого или сложного полиморфизма.* В соответствии с этим решение о конкретной реализации принимается с учетом особенностей использования

разрабатываемых классов. Причем если объектных полей больше двух, но ограниченное и точно указанное количество, то использование композиции просто необходимо.

**Пример 3.2.** Включение в класс заданного количества однотипных объектов. Рассмотрим реализацию класса Массив целых чисел. В качестве элементов массива будем также использовать объекты класса Целое число примера 2.2. Аналогично предыдущему примеру будем считать, что описание класса Tlong находится в файле Tlongclass.h.

```
#include "Tlongclass.h"
#include <conio.h>
class    mastlong    // Класс Массив целых чисел
{
private:    dlong    size;    // размер массива
    Tlong    mas[10];    // массив объектов класса Tlong
public:
    mastlong(){}    // неинициализирующий конструктор
    mastlong(dlong af,dlong m1[]) // конструктор с параметрами
    { setmas(af,m1); }
    ~mastlong(){}    // деструктор
    void printm(); // вывод элементов массива на экран
    void setmas(dlong af,dlong m1[]); // инициализация полей
};
void mastlong::setmas(dlong af,dlong m1[])
{
    int i;
    if (af <= 10) size=af; else size=10;
    for(i=0;i<size;i++)    mas[i].setnum(m1[i]);
}
void mastlong::printm()
{
    int i;
    printf("Содержимое объекта МАССИВ \n");
    for(i=0;i<size;i++)    mas[i].print();
}
void main()
```

```
{
    setlocale(0, "russian");
    int i, n; dlong mn[4]={456, 5678, 64328, 45234};
    mastlong a(4, mn);
    a.printm();
    _getch();
}
```

В данном примере, независимо от размера реального массива, в объекте создается поле – массив на 10 элементов. Но если количество объектов сильно меняется в зависимости от применения объекта или заранее непредсказуемо, то лучше использовать поле – динамический массив, реализуя при этом механизм наполнения.

## 3.2 Наполнение

*Наполнением* называют такое отношение классов, при котором количество объектов некоторого класса, включаемых в другой класс, не ограничено и может меняться от нуля до достаточно больших значений. В С++, как и в других языках программирования, наполнение реализуется с применением указателей. Использование указателей позволяет управлять включаемыми объектами, которые, как правило, собраны в массив или списковую структуру. Наполнение может использоваться и в случае небольшого количества объектных полей, не связанных в структуры, но это приводит к получению более сложной программы, чем при применении композиции из-за проблем реализации динамических полей.

**Пример 3.3** Использование наполнения для реализации включения объектов.

Демонстрацию наполнения опять выполним на базе классов Целое число и Вещественное число из примеров 2.2 и 3.1. Построим класс Вещественное число, используя указатели на объекты класса Целое число. Поскольку описание класса Целое число не меняется, для сокращения текста опять воспользуемся уже имеющимся файлом "Tlongclass.h".

```
#include "Tlongclass.h"
#include <string.h>
#include <iostream>
using namespace std;
class Treal // Класс Вещественное число
{
public:    Tlong *celn; // поле целая часть - указатель на объект класса Tlong
        Tlong *drob; // поле дробная часть - указатель на объект класса Tlong
        char *real; // поле вещественное число - указатель на строку
        Treal() // конструктор с инициализацией указателей
        { celn=drob=NULL; real=NULL; }
        Treal::Treal(char * st) // инициализирующий конструктор
        { setnumv(st); }
        ~Treal() // деструктор
        {
```

```

        delete real;
        if (celn!=NULL) delete celn;
        if (drob!=NULL) delete drob;
    }
    void printr();    // ВЫВОД ЗАПИСИ ВЕЩЕСТВЕННОГО ЧИСЛА
    void setnumv(char * st); // инициализация полей
};

void Treal::setnumv(char * st)
{
    int l=strlen(st); char *ptr;  dlong t;
    real=new char[l+1];  strcpy(real,st);
    ptr=strchr(real, '.');
    t=dlong(atol(ptr+1));  *ptr='\0';
    if (t!=0) drob=new Tlong(t);
    t=dlong(atol(real));  *ptr='.';
    if (t!=0) celn=new Tlong(t);
}

void Treal::printr()
{
    cout<<"Вещественное число " <<real<<endl;
    if (celn!=NULL){cout<<"Целая часть   ":";  celn->print();}
    if (drob!=NULL){cout<<"Дробная часть ":";  drob->print();}
    cout<<endl;
}

void main ()
{
    setlocale(0,"russian");
    Treal a("78457.23065");
    a.printr();
    system("pause");
}

```

Из примера видно, что объем программы при использовании наполнения практически такой же, как при применении композиции. Он увеличивается только благодаря включению в конструктор операций по выделению памяти под поля объекта и в

деструктор – по освобождению выделенной конструктором памяти. Но у приведенного варианта есть одно достоинство. Если у представляемого вещественного числа отсутствует дробная или целая часть, то память под это поле не выделяется, в отличие от предыдущего примера, где память под поля выделяется всегда.

Таким образом, мы вновь констатируем, что при проектировании сложных классов выбор конкретной реализации зависит от целей и задач, для которых они создаются. Во многих случаях выполняют комбинирование в одном классе различных отношений.

### 3.3 Особенности работы с динамическими полиморфными объектами

Как уже отмечалось, C++ позволяет присваивать указателям на базовый класс адреса объектов любого из производных классов. В этом случае возникают проблемы с доступом к полям объекта, описанным в производном классе.

Во-первых, указатель на объект базового класса связан с описанием его полей, и поля, описанные в производном классе, для него невидимы. Поэтому при обращении через указатель на базовый класс к полям объекта производного класса необходимо средствами языка явно переопределить («привести») тип указателя (см. раздел 3.4).

Во-вторых, если при определении указателя на базовый класс создается динамический объект производного класса, то во время уничтожения такого объекта вызывается деструктор лишь базового класса и память освобождается некорректно, так как деструктор не может правильно определить размеры освобождаемой памяти. Эта проблема решается применением виртуального деструктора. При объявлении деструктора базового класса виртуальными деструкторы производных классов также становятся виртуальными. Тогда при уничтожении объекта с помощью оператора delete вызов деструктора будет происходить через ТВМ и, следовательно, деструкторы производных классов будут вызваны корректно.

Обе указанные особенности имеют место и при работе с полиморфными объектами.

**Пример 3.4.** Использование указателей на базовый класс и виртуального деструктора. В программе определены два класса: класс, содержащий поле целого типа, и производный от него класс, содержащий в качестве поля динамический массив, размер которого определяется значением поля целого типа, унаследованного от родителя.

```
#include <locale.h>
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
class integ
{
protected: int n;
public:
```

```

    integ(int vn) { n=vn; } // конструктор
    virtual ~integ() {}      // деструктор
    virtual void print(void) { cout<<" " <<n<<endl;}
};
class masinteg: public integ
{
    int *mas;
public:
    masinteg(int vn); // конструктор
    ~masinteg(){ delete [] mas; } // деструктор
    void print(void);
};
masinteg::masinteg(int vn):integ(vn)
{
    mas=new int[n];
    for(int i=0;i<n;i++) mas[i]=rand()/1000;
}
void masinteg::print()
{
    for(int i=0;i<n;i++)
    { cout<<" " <<mas[i]; cout<< endl; }
}
void main()
{
    setlocale(0,"russian");
    srand( (unsigned)time( NULL ) );
    integ *pa; // указатель на базовый класс
    pa=new integ(5); // создание объекта базового класса
    pa->print();
    delete pa; // автоматический вызов деструктора integ
    pa=new masinteg(6); // создание объекта производного класса
    pa->print();
    delete pa; // автоматический вызов деструктора masinteg
    system("pause");
}

```

### 3.4 Восходящее и нисходящее изменение типа объектов

В ООП при работе с объектами возможно временное изменение их типа. При этом различают:

<sup>35</sup><sub>17</sub> нисходящее приведение типа;

<sup>35</sup><sub>17</sub> восходящее приведение типа.

Приведение типа объекта называют *нисходящим*, если в его результате указатель или ссылка на объекты базового класса преобразуется в указатель или ссылку на объекты производного, и *восходящим*, если указатель или ссылка на объекты производного класса преобразуется в указатель или ссылку на объекты базового класса.

**Восходящее приведение** типа возможно всегда, поскольку все поля и методы предка унаследованы потомком, а, следовательно, ошибок при обращении к компонентам при восходящем приведении возникать не будет. Данный тип приведения типа объекта используют при необходимости вызвать переопределенный в потомке метод базового класса, например:

```
class A {
    public:
        void func(char ch);
};
class B : public A {
    public:    void func(char *str);
};
...
B b;
b.func("c");    // вызвать B::func()
(A)b.func('c'); // вызвать A::func(); (A)b - восходящее приведение типа
```

**Нисходящее приведение** приведение допустимо, только для случаев, когда объект производного класса доступен через указатель или ссылку на объект базового, т.е. когда при приведении восстанавливается соответствие объекта своему классу (или при сложной иерархии – возможно его родителю, если указатель или ссылка принадлежали более «древним» предкам).

Нисходящее приведение выполняют, чтобы получить через указатель или ссылку базового класса доступ к компонентам, добавленным в производном классе. Такая ситуация встречается сравнительно часто при работе с полиморфными объектами.

При выполнении нисходящего приведения типов необходима проверка, так как никакой гарантии, что указатель ссылается на адрес объекта именно данного производного класса, нет. Используется же это преобразование при работе с полиморфными объектами постоянно, в связи с тем, что это единственный способ обеспечить видимость полей производного класса при работе с объектом через указатель на базовый класс.

**Операторы приведения типов.** В последних версиях C++ приведение типов выполняется, как традиционно, так и с использованием специальных операторов. Рассмотрим целесообразность использования всех возможных вариантов для нисходящего приведения типов:

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++ только для полиморфных классов, требует указания опции компилятора /GR (Project/Settings...) в Visual Studio, если приведение невозможно, то возвращает NULL.

Следовательно для нисходящего приведения типов полиморфных объектов целесообразно применять именно `dynamic_cast`.

Рассмотрим эффект использования различных вариантов приведения типов более подробно.

Динамическое приведение типа: `dynamic_cast <T>(t)`.

Операнды:

T – указатель или ссылка на класс или `void*`,

t – выражения типа указателя, причем оба операнда либо указатели, либо ссылки.

Приведение типа осуществляется во время выполнения программы. Предусмотрена проверка возможности преобразования, использующая RTTI (информацию о типе

переменных, используемую во времени выполнения программы), которая строится в C++ только для полиморфных объектов.

Применяется для нисходящего приведения типов полиморфных объектов, например:

```
#include <iostream>
class A
{
public:   virtual ~A() {} /* класс обязательно должен включать
                        виртуальный метод, так как для выполнения
                        приведения требуется RTTI */
};
class B: public A
{
public:   virtual ~B() {}
};
void func(A& a) /* функция, работающая с полиморфным объектом */
{
    B& b=dynamic_cast<B&>(a); // нисходящее приведение типов
}
void main()
{
    B b;
    func(b); // вызов функции с полиморфным объектом
    system("pause");
}
```

Если вызов `dynamic_cast` осуществляется в условной конструкции, то ошибка преобразования, обнаруженная на этапе выполнения программы, приводит к установке значения указателя равным `NULL`, в результате чего активизируется ветвь «иначе».

Например:

```
if (Derived* q=dynamic_cast<Derived*> (p)
    {<если преобразование успешно, то ...>}
else {<если преобразование неуспешно, то ...>}
```

В этом примере осуществляется преобразование указателя на объекты базового класса в указатель на объекты производного класса с проверкой правильности по RTTI на этапе выполнения программы. Если преобразование некорректно, то оператор возвращает NULL, и устанавливается `q=NULL`, в результате чего управление передается на ветвь `else`.

Если вызов осуществляется в операторе присваивания, то при неудаче генерируется исключение `bad_cast`. Например:

```
Derived* q=dynamic_cast<Derived*> (p);
```

Статическое приведение типа: `static_cast<T>(t)`.

Операнды: `T` – указатель, ссылка, арифметический тип или перечисление; `t` – аргумент типа, соответствующего `T`. Оба операнда должны быть определены на этапе компиляции. Операция выполняется на этапе компиляции без проверки правильности преобразования.

Может преобразовывать:

1) целое число в целое другого типа или в вещественное и обратно:

```
int i;
float f=static_cast<float>(i); /* осуществляет преобразование
                               без проверки на этапе компиляции программы */
```

2) указатели различных типов, например:

```
int *q=static_cast<int>(malloc(100)); /* осуществляет преобразование
                                       без проверки на этапе компиляции программы */
```

3) указатели и ссылки на объекты иерархии в указатели и ссылки на другие объекты той же иерархии, если выполняемое приведение однозначно. Например, восходящее приведение типов или нисходящее приведение типов непалиморфных объектов, иерархии классов которых не используют виртуального наследования:

```
class A {...}; // класс не включает виртуальных функций
class B:public A{}; // не используется виртуальное наследование
```

```

void somefunc()
{
    A    a;  B    b;
    B& ab=static_cast<B&>(a); // нисходящее приведение
    A& ba=static_cast<A&>(b); // восходящее приведение
}

```

Остальные два оператора приведения типа напрямую с объектами обычно не используются. Это оператор `const_cast<T>(t)`, который применяют для отмены действия модификаторов `const` или `volatile`, и оператор `reinterpret<T>(t)`, который осуществляет преобразования, ответственность за которые полностью ложится на программиста.

**Пример 3.5.** Восходящее и нисходящее приведение типов объектов с использованием различных средств языка.

```

#include <iostream>
#include <string.h>
using namespace std;
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;}
};
void main()
{
    TA *pA=new TA('A'), *pC=new TB("AB");
    TB *pB=new TB("AC");
}

```

```

// восходящее приведение типов
    ((TA *)pB)->func();
    reinterpret_cast<TA *>(pB)->func();
    static_cast<TA *>(pB)->func();
    dynamic_cast<TA *>(pB)->func();

// нисходящее приведение типов - корректно
    ((TB *)pC)->func();
    reinterpret_cast<TB *>(pC)->func();
    static_cast<TB *>(pC)->func();
    dynamic_cast<TB *>(pC)->func();

// нисходящее приведение типов - некорректно
    ((TB *)pA)->func();
    reinterpret_cast<TB *>(pA)->func();
    static_cast<TB *>(pA)->func();
    // dynamic_cast<TB *>(pA)->func(); // фиксирует ошибку!

// проверка корректности нисходящего преобразования
    if (TB *pD=dynamic_cast<TB *>(pA)) pD->func();
    else cout<<"Cast Error"<<endl;
    system("pause");
}

```

### 3.5 Контейнерные классы

Решение многих задач подразумевает создание наборов объектов и обработку таких наборов. Объект, назначением которого является хранение объектов других типов и управление ими, принято называть *контейнером*. Соответственно класс такого объекта называют *контейнерным*.

При реализации контейнерного класса используют механизмы композиции или наполнения. Если контейнерный класс реализует механизм композиции, то тип и количество управляемых объектов жестко определены типом и количеством объектных полей. Если он применяет механизм наполнения, то подключение реализуется через указатели, следовательно, контейнер может управлять как объектами некоторого базового, так и объектами всех потомков этого класса.

Обычно контейнерные классы реализуют типовые структуры, такие, как массив, стек или список, и включают методы, выполняющие основные операции над хранимыми данными.

Обязательная операция, присутствующая в любом контейнерном классе – последовательная обработка объектов. Такая обработка может обеспечиваться двумя способами.

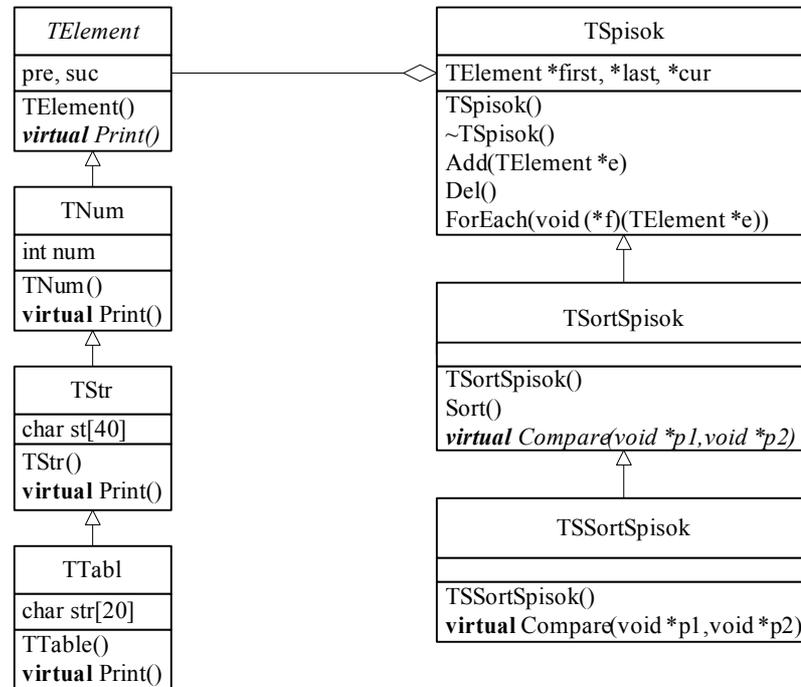
**Первый способ** базируется на создании специальной процедуры просмотра всех элементов контейнера. В эту процедуру в качестве параметра передается имя функции или процедуры, реализующей алгоритм требуемой обработки элемента контейнера.

**Второй способ** реализуется через определение итератора или класса итераторов, подходящего для данного вида контейнера. При помощи итераторов программист может управлять контейнером, не зная фактических типов элементов. Несколько ключевых компонентных функций позволяют программисту найти концы последовательности элементов.

**Пример 3.6.** Контейнерный класс с процедурой поэлементной обработки. Пусть требуется разработать контейнер на базе сортированного списка элементов, принадлежащих иерархии классов Число – Строка – Таблица.

Структуру классов будем разрабатывать поэтапно. В основу иерархии классов положим классы Список – Элемент. Эти два класса образуют контейнерный класс, управляемые объекты которого должны наследоваться от класса Элемент (рис. 3.1). Класс

Список будет содержать три поля – указатели на объекты класса Элемент: указатель на первый элемент, указатель на последний элемент и указатель на текущий элемент. Эти указатели используются для организации двусвязного списка. Класс Элемент содержит только два поля – указатели на элементы того же класса, которые будут хранить адрес следующего и адрес предыдущего элементов списка.



**Рис. 3.1.** Диаграмма классов, реализуемая в примере 3.6

Затем на базе этих классов разработаем абстрактный контейнерный класс Сортированный список, предусматривающий метод сортировки `Sort` с внутренним вызовом метода сравнения элементов `Compare`. Наличие внутреннего метода сравнения позволит в дальнейшем на базе этого класса создавать другие классы, использующие различные законы сортировки.

От класса Элемент наследуем классы Число, Строка, Таблица, добавляя новые поля и перекрывая метод вывода содержимого элемента `Print()`.

Теперь можно описать класс Пользовательский сортированный список, который переопределит метод сравнения элементов при сортировке, задав сравнение реальных полей классов предметной области задачи.

```

#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>

```

```

class TElement          // абстрактный класс Элемент списка
{
public: TElement *pre,*suc;
    TElement(void) { pre=suc=NULL;} // конструктор
    virtual void Print(void)=0; // абстрактный метод вывода
};

class TSpisok          // класс Список
{
protected: TElement *first,*last,*cur; /*указатели на первый,
                                         последний и текущий элементы списка */
public: TSpisok(void){ first=last=cur=NULL;}
    ~TSpisok(); // деструктор
    void Add(TElement *e); // добавление элемента в список
    TElement *Del(void); // удаление элемента из списка
    void ForEach(void (*f)(TElement *e)); // поэлементная обработка
};

TSpisok::~TSpisok()
{
    while ((cur=Del())!=NULL)
        { cur->Print(); delete(cur); }
}

void TSpisok::Add(TElement *e)
{
    if (first==NULL) first=last=e;
    else { e->suc=first; first=first->pre=e;}
}

TElement *TSpisok::Del(void)
{
    TElement *temp=last;
    if (last!=NULL)
    {
        last=last->pre;
        if (last!=NULL) last->suc=NULL;
    }
}

```

```

        if (last==NULL) first=NULL;
        return temp;
    }
void TSpisok::ForEach(void (*f)(TElement *e))
{
    cur=first;
    while (cur!=NULL) { (*f)(cur); cur=cur->suc;}
}
class TSortSpisok:public TSpisok // класс Сортированный список
{
protected:
    virtual int Compare(void *op1,void *op2)=0; /* абстрактный метод
                                                сравнения для процедуры сортировки */
public:
    void Sort(void); // метод сортировки
    TSortSpisok(void):TSpisok(){} // конструктор
    ~TSortSpisok(){} // деструктор
};
void TSortSpisok::Sort(void)
{
    int swap=1;    TElement *temp;
    while (swap)
    {
        swap=0; cur=first;
        while (cur->suc!=NULL)
        {
            if (Compare(cur,cur->suc))
            {
                temp=cur->suc;
                cur->suc=temp->suc;
                temp->suc=cur;
                if (cur->pre!=NULL)
                    cur->pre->suc=temp;
                else first=temp;
                temp->pre=cur->pre;
            }
        }
    }
}

```

```

        cur->pre=temp;
        if (cur->suc!=NULL)
            cur->suc->pre=cur;
        else last=cur;
        cur=temp; swap=1;
    }
    else cur=cur->suc;
}
}
}
class TNum: public TElement // класс Число
{
public: int num; // числовое поле целого типа
    void Print(void) { printf("%d ",num); }
    TNum(){} // конструктор по умолчанию
    TNum(int n):num(n) {} // конструктор
};
class TStr: public TNum // класс Строка (num - длина строки)
{
public: char st[40]; // поле символьная строка
    TStr(){} // конструктор по умолчанию
    TStr(char *s):TNum(strlen(s))
    {
        strcpy(st,s);
        if (num>=40)st[40]='\0'; else st[num+1]='\0';
    }
    void Print(void)
    { TNum::Print(); printf("%s\n",st); }
};
class TTabl: public TStr // класс Таблица - добавляет строку
{
public: char str[20];
    void Print(void)
    { TStr::Print(); printf("%s\n ",str); }
};

```

```

TTabl(){} // конструктор по умолчанию
TTabl(char *s,char *s2):TStr(s)
{
    strcpy(str,s2);
    if (strlen(s2)>=20) str[20]='\0';
    else str[strlen(s2)+1]='\0';
}
};
class TSSpisok:public TSortSpisok /* класс Пользовательский
                               сортированный список*/
{
protected:
    int Compare(void *op1,void *op2) /* метод сравнения
    для процедуры сортировки с явным преобразованием типа */
    { return (((TTabl *)op1)->num)<(((TTabl *)op2)->num); }
public:
    TSSpisok(void):TSortSpisok(){}
    ~TSSpisok(void){}
};
void Show(TElement *e) // процедура для передачи в метод ForEach
{ e->Print(); }
TSSpisok N; // объект класса Сортированный список
void main()
{
    setlocale(0,"russian");
    int k; char str[40]; char str2[20];
    TElement *p; // указатель на базовый класс TElement
    // цикл формирования списка из объектов классов TNum, TStr, TTabl
    while ( printf("Введите число:"), scanf("%d",&k) !=EOF)
    {
        p=new TNum(k); N.Add(p);
        printf("Введите строку:"); scanf("%s",str);
        printf("Введите строку 2:"); scanf("%s",str2);
        p=new TTabl(str,str2); N.Add(p);
    }
}

```

```

        printf("Введите строку:");  scanf("%s",str);
        p=new TStr(str);             N.Add(p);
    }
    puts("\nВведены элементы:");
    N.ForEach(Show);    // вывод элементов списка
    N.Sort();           // сортировка
    puts("\nПосле сортировки:");
    N.ForEach(Show);
    _getch();
}

```

**Пример 3.7.** Контейнерный класс с итераторами. Для демонстрации работы с итераторами воспользуемся текстом примера 3.6, изменив его следующим образом: из класса TSpisok удалим метод ForEach() и добавим два метода-итератора ifirst() и inext(). Описание остальных классов оставим без изменения.

В результате класс TSpisok будет иметь вид

```

class TSpisok           // класс Список
{
protected: TElement *first,*last,*cur;
public:
    TSpisok(void){ first=last=cur=NULL;} // конструктор
    ~TSpisok();    // деструктор
    void Add(TElement *e); // добавление элемента в список
    TElement *Del(void); // удаление элемента из списка
    // функции-итераторы
    TElement * ifirst() // указатель на первый элемент
    { return cur=first; }
    TElement * inext() // указатель на следующий элемент
    { return cur=cur->suc; }
};

```

Фрагмент программы, демонстрирующий применение итераторов для такого описания класса, будет выглядеть следующим образом:

```

    TSpisok N; // объявление объекта класса Список
void main(void)
{
    int k; char str[40]; char str2[20];
    TElement *p;
    // цикл формирования списка из элементов классов TNum, TStr, TTabl
    while (printf("Введите число: "), scanf("%d", &k) != EOF)
    {
        p=new TNum(k);      N.Add(p);
        printf("Введите строку:"); scanf("%s", str);
        printf("Введите строку 2:"); scanf("%s", str2);
        p=new TTabl(str, str2);      N.Add(p);
        printf("Введите строку:"); scanf("%s", str);
        p=new TStr(str);      N.Add(p);
    }
    puts("\nВведены элементы:");
    // цикл обработки каждого элемента списка с помощью итераторов
    p=N.ifirst();
    while (p!=NULL)
    {    p->Print();    p=N.inext();    }
    _getch();
}

```

**Использование шаблонов классов для проектирования контейнеров.** Очень часто для реализации контейнера используются параметризованные классы, в том числе и стандартные, описанные в библиотеке CLASSLIB. Они также могут работать с объектами одного класса или иерархии классов. В последнем случае в качестве параметра в шаблон передается имя класса-родителя или указатель на класс-родитель. Пример контейнера, построенного на основе шаблона, приведен в разделе 6.3.

### Вопросы для самоконтроля

1. Какое отношение между классами называется композицией? Для чего она используется?

[Ответ.](#)

2. Чем композиция отличается от наполнения? Как наполнение реализуется?

[Ответ.](#)

3. Что такое виртуальный деструктор и каковы особенности его использования?

[Ответ.](#)

4. Что такое восходящее и нисходящее приведение типов? Где и как они используются?

[Ответ.](#)

5. Что называют "контейнером" в программировании? Для чего используют контейнеры?

[Ответ.](#)

## 4 Особые случаи организации доступа к объектам и их компонентам

### 4.1 Локальные и вложенные классы

Класс может быть объявлен внутри некоторой функции. Такой класс в C++ принято называть *локальным*. Функция, в которой объявлен локальный класс, не имеет непосредственного доступа к компонентам локального класса, доступ осуществляется с указанием имени объекта встроеного класса. Локальный класс не может иметь статических полей. Объект локального класса может быть создан только внутри функции, в области действия объявления класса. Все компонентные функции локального класса должны быть встраиваемыми.

Иногда возникает необходимость объявления одного класса внутри другого. Такой класс называется *вложенным*. Вложенный класс расположен в области доступа класса, внутри которого он объявлен. Соответственно, объекты этого класса могут использоваться как компоненты внешнего класса. Компонентные функции и статические компоненты вложенного класса могут быть описаны вне глобального класса.

**Пример 4.1.** Вложенные классы. Ниже описан класс TGroup (Группа студентов), составной частью которого является класс TStudent (Студент). Фактически между этими классами реализовано наполнение, но при описании класс Студент вложен в класс Группа.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
class TGroup
{
    class TStudent
    {
        private: char * Famyli,*Name;    float old;
        public:
            char * getFam() {    return Famyli;    }
            char * getName () {    return Name;    }
    }
}
```

```

float getold() { return old; }
void setstudent()
{
    char s[30];
    puts("input famyli"); gets(s);
    Famyli= new char [strlen(s)+1];
    strcpy(Famyli,s);
    puts("input Name"); gets(s);
    Name= new char [strlen(s)+1];
    strcpy(Name,s);
    puts("Input Old"); gets(s);
    old=atof(s);
}
void printst()
{
    printf("%15s %10s %6.2f\n",
           getFam(), getName(), getold());
}
};

int size; TStudent *Masgroup;
public:
void TGroup::setgroup(int asize)
{
    size=asize; Masgroup=new TStudent[size];
    for(int i=0;i<size;i++)
        Masgroup[i].setstudent();
}
void printGr()
{
    puts("Information of Group");
    for(int i=0;i<size;i++)
        Masgroup[i].printst();
}
};

```

```
void main()
{
    TGroup AK;    int n;
    puts("input n<=30"); scanf("%d\n",&n);
    AK.setgroup(n);
    AK.printGr();
    _getch();
}
```

## 4.2 Статические компоненты класса

Класс – это тип, а объект – конкретный представитель этого класса в программе. Для каждого объекта существует своя копия полей класса. Если все объекты одного типа используют некоторые данные совместно, то возникает проблема размещения этих данных и обеспечения их доступности из всех объектов класса. Эту проблему можно решить использованием внешних `extern` или внешних статических `extern static` переменных, но это нарушит принцип инкапсуляции, поскольку переменные будут доступны не только объектам класса. Более грамотным решением является применение статических компонентов класса, что позволит снизить потребность в глобальных переменных программы.

*Статическими* называются компоненты класса, объявленные с модификатором памяти `static`. Такие компоненты (поля и методы) являются частью класса, но не связаны с его объектами.

Имеется только одна копия *статических полей класса* – общая для всех объектов данного класса, которая существует даже при их отсутствии.

Инициализацию статических полей класса осуществляют обязательно вне определения класса, но с указанием квалификатора видимости `<Имя класса>::`. Например:

```
class point
{
    int x,y;
    static int obj_count; /* статическое поле (счетчик обращений),
                           инициализация в этом месте не возможна */
public:
    point () { x=0; y=0; obj_count++; } // обращение к статическому полю
};
int point::obj_count=0; // инициализация статического поля
```

Любой метод может обратиться к статическому полю класса и изменить его значение. Существует также возможность обращения к статическим полям класса при отсутствии объектов данного класса. Такой доступ осуществляют с помощью

*статических компонентных функций* – компонентных функций, объявленных со спецификатором `static`.

Статические функции не ассоциируются с каким-либо объектом и не получают параметра `this`. Следовательно, *они не могут без указания объекта обращаться к нестатическим полям класса*. При необходимости ссылка на конкретный объект может быть передана в списке параметров, и тогда статическая функция может обратиться к нестатическим полям объекта следующим образом:

<Имя объекта>.<Имя нестатического поля класса>.

При обращении к статическим полям класса такой проблемы не возникает:

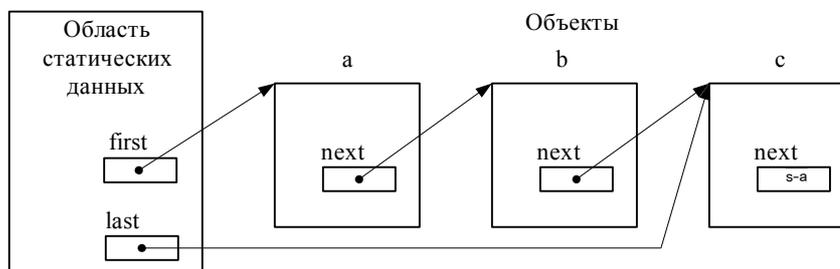
```
class point
{
    int x,y,color;          // нестатические поля
    static int obj_count;  // статическое поле - счетчик обращений
public:
    point () { x=0; y=0; color=5; }
    static void draw_point(point &p); // статическая функция
};
int point::obj_count=0; // инициализация статического поля
void point::draw_point(point &p) // имя объекта передано через параметр
{
    putpixel(p.x,p.y,p.color ); // обращение к нестатическому полю
    obj_count++;                // обращение к статическому полю
}
```

Обращаться к статическим компонентам класса, являющимся принадлежностью всех объектов данного класса, можно, указав вместо имени объекта имя класса:

<Класс>::<Компонент>.

### Пример 4.2. Класс со статическими компонентами.

В качестве примера, иллюстрирующего полезные свойства статических компонентов, рассмотрим класс, использующий статические компоненты для построения списка своих объектов (рис. 4.1).



**Рис. 4.1.** Структура списка объектов, использующего статические компоненты класса

В поле `first` хранится адрес первого элемента списка, в поле `last` – адрес последнего элемента списка. Нестатическое поле `next` хранит адрес следующего объекта. Сформированный в примере список использован для вывода всех экземпляров класса с помощью статической функции `displayAll()`. Статическая функция `displayAll()`, как не получающая неявно адреса полей объекта посредством указателя `this`, обращается к нестатическому полю `next` с указанием конкретного объекта через явно заданный параметр:

```
#include <locale.h>
#include <string.h>
#include <conio.h>
#include <stdio.h>
class String
{
public:    char str[40];
        static String *first; // статическое поле - указатель на начало списка
        static String *last; // статическое поле - указатель на конец списка
        String *next;
        String(char *s)
        {
            strcpy(str,s);
            next=NULL;
            if (first==NULL) first=this;
            else last->next=this;
        }
};
```

```

        last=this;
    }
    void display()    { puts(str); }
    static void displayAll();    // объявление статической функции
};
String *String::first=NULL; // инициализация статических компонентов
String *String::last=NULL;
void String::displayAll()    // описание статической функции
{
    String *p=first;
    if (p==NULL) return;
    do { p->display(); p=p->next; }
    while (p!=NULL);
}
void main(void)
{
    setlocale(0,"russian");
    String a(" Пример "), b(" использования статических "),
           c(" компонентов"); // объявление-создание трех объектов класса
    if (String::first!=NULL) // обращение к общему статическому полю
        String::displayAll(); // обращение к статической функции
    _getch();
}

```

### 4.3 Дружественные функции и классы

Как было отмечено ранее, механизм управления доступом позволяет выделить внутренние `private`, защищенные `protected` и общедоступные `public` компоненты классов. Причем внутренние компоненты локализованы в классе и не доступны извне, а защищенные – доступны только компонентным функциям класса и его наследникам. Такое ограничение доступа к внутренним и защищенным компонентам класса может оказаться неоправданно строгим. Оно может существенно сужать возможности наследования других классов от данного и сокращать количество вариантов его использования.

Кроме того, бывают случаи, когда функции, не являющиеся компонентными, должны иметь возможность обращаться к внутренним компонентам класса. В такой ситуации класс может предоставить особые привилегии определенным внешним функциям или компонентным функциям другого класса. Эти функции получили название дружественных.

По определению, *дружественной функцией* класса называется функция, которая, не являясь компонентом некоторого класса, имеет доступ ко всем его компонентам, в том числе внутренним. Функция не может стать другом класса «без его согласия». Для получения «прав друга» функция должна быть описана в теле класса со спецификатором `friend`. Именно при наличии такого описания класс предоставляет функции права доступа к защищенным и внутренним компонентам.

**Пример 4.3.** Внешняя дружественная функция.

```
#include <locale.h>
#include <iostream>
using namespace std;
class FIXED
{
private:    int a;
public:    FIXED(int v):a(v){}
    friend void show(FIXED);
};
void show(FIXED Par)
```

```

{
    cout<<Par.a<<endl; // функция имеет доступ к внутреннему полю a
    Par.a+=10;
    cout<<Par.a<<endl;
}
void main()
{
    setlocale(0,"russian");
    FIXED aa(25);
    cout << "Результаты работы: " << endl;
    show(aa); // выводит: 25 35
    system("pause");
}

```

Друзьями класса могут являться и компонентные функции другого класса. Однако следует заметить, что такую привилегию методу другого класса может предоставить лишь автор данного класса, а не автор метода.

**Пример 4.4.** Дружественная функция – компонент другого класса.

```

#include <locale.h>
#include <iostream>
using namespace std;
class FLOAT; // объявление класса без его определения
class FIXED
{
private:    int a;
public:    FIXED(int v):a(v) {}
           double Add(FLOAT);
           void print(){cout <<a << endl;}
};
class FLOAT
{
private:    double b;
public:    FLOAT(double val) { b=val; }
           void print(){ cout <<b << endl; }
}

```

```

friend double FIXED::Add(FLOAT); /* компонентная функция
      класса FIXED объявляется дружественной классу FLOAT */
};
double FIXED::Add(FLOAT Par)
{   return a+Par.b;   } /* функция получает доступ к внутреннему
                          полю класса FLOAT */

void main()
{
    setlocale(0,"russian");
    FIXED aa(25); FLOAT bb(45);
    cout << "Результаты работы: " << endl;
    cout << aa.Add(bb) << endl; // выводит: 70
    aa.print(); // выводит: 25
    bb.print(); // выводит: 45
    system("pause");
}

```

Следует отметить некоторые особенности дружественных функций. Дружественная функция при вызове не получает указатель `this`, поскольку не является компонентной. Объекты класса должны передаваться дружественной функции явно (через параметры).

Так как дружественная функция не является компонентом класса, на нее не распространяется действие спецификаторов доступа `public`, `protected`, `private`. Поэтому место размещения прототипа дружественной функции внутри определения класса безразлично. Права доступа дружественной функции не изменяются и не зависят от спецификаторов доступа.

Использование механизма дружественных функций позволяет упростить интерфейс между классами. Например, дружественная функция может получить доступ к внутренним и защищенным компонентам сразу нескольких классов. Тогда из этих классов можно убрать компонентные функции, предназначенные только для обеспечения доступа к «скрытым» компонентам.

Поскольку ограничение доступа к дружественной функции не относится, работать она будет достаточно быстро, и написать ее не труднее, чем функцию доступа к обычной структуре C++.

Если необходимо, чтобы все функции некоторого класса имели доступ к внутренним полям другого класса, то весь класс может быть объявлен дружественным:

```
friend class <Имя класса>.
```

#### Пример 4.5. Объявление дружественного класса.

```
#include <locale.h>
#include <iostream>
using namespace std;
class SHOW; // объявление класса без его определения
class PAIR
{
private:    char *Head,*Tail;
public:
    PAIR(char *one,char *two):Head(one),Tail(two){}
    friend class SHOW; // объявление дружественного класса
};
class SHOW /* всем функциям класса SHOW доступны внутренние
                                           поля класса PAIR*/
{
private:    PAIR Twins;
public:    SHOW(char *one,char *two):Twins(one,two){}
    void Head() { cout <<Twins.PAIR::Head << endl; }
    void Tail() { cout <<Twins.PAIR::Tail << endl; }
};
void main()
{
    setlocale(0,"russian");
    SHOW aa("ПРИВЕТ","ДРУЗЬЯ");
    cout << "Результаты работы:"<<endl;
    aa.Head(); // выводит: ПРИВЕТ
    aa.Tail(); // выводит: ДРУЗЬЯ
    system("pause");
}
```

}

### Вопросы для самоконтроля

1. Что собой представляют локальные классы? Для чего они используются?

[Ответ.](#)

2. Что такое статические компоненты класса? Чем они отличаются от обычных компонентов? Зачем используются?

[Ответ.](#)

3. Что такое дружественные функции и дружественные классы? Как определить дружественные функции? Где и как они используются?

[Ответ.](#)

## 5 Переопределение операций

### 5.1 Функции-операторы, их типы и ограничения на переопределение

В языке C++ операции над стандартно определенными типами данных являются встроенными, т. е. программист не может повлиять на выполнение этих операций. Однако язык предоставляет возможность переопределения любой из существующих операций для объектов вновь созданных классов.

При переопределении операций ни одно из ее исходных значений не теряется. Просто вводится операция с похожим смыслом для объектов нового класса.

Переопределяемые операции реализуются как особый вид функции со специальным именем `operator@`, где @ – символ переопределяемой операции. Такие функции обычно называются *функциями-операторами*. Функция-оператор может быть определена как компонент класса или как внешняя независимая от класса (свободная) функция. Соответственно различают:

<sup>35</sup><sub>17</sub>*простую*, т. е. определенную вне класса функцию-оператор;

<sup>35</sup><sub>17</sub>*компонентную*, т. е. определенную в классе функцию-оператор.

И простые, и компонентные функции-операторы могут быть одноместными и двуместными. В первом случае у них один параметр, а во втором – два. Единственную существующую в C++ операцию с тремя параметрами (условную операцию) переопределять запрещено.

Одноместные и двуместные функции-операторы в зависимости от места переопределения (в классе или вне его) описываются по-разному. Это связано с тем, что первый аргумент компонентной функции – всегда объект класса, задаваемый неявно, а потому одноместная компонентная функция объявляется без списка параметров, двуместная – с одним параметром.

Общая форма определения функции-оператора представлена в табл. 5.1. В приведенных в таблице описаниях символ @ – любая допустимая операция, <Тип результата> – тип возвращаемого значения (чаще всего того же типа, что и класс, хотя возможен и другой тип этого значения).

Таблица 5.1. Формы описания функции-оператора

Простая функция	Компонентная функция
-----------------	----------------------

одноместная	одноместная
<Тип результата> operator @ (Аргумент)	<Тип результата> operator @ ()
двуместная	двуместная
<Тип результата> operator @ (Арг1,Арг2)	<Тип результата>operator @ (Арг2.)

При переопределении операций следует помнить, что

<sup>35</sup><sub>17</sub> нельзя переопределять операции \*, sizeof, ?:, #, ##, ::, class::;

<sup>35</sup><sub>17</sub> операции =, [], () можно переопределять только в составе класса;

<sup>35</sup><sub>17</sub> переопределенная операция = не наследуется в производных классах;

<sup>35</sup><sub>17</sub> нельзя изменять приоритет и ассоциативность операции (порядок выполнения операций одного приоритета).

Функция-оператор допускает две формы вызова: стандартную и операторную, описанные в табл. 5.2.

**Таблица 5.2** Формы вызова функции – оператора

Стандартная форма	Операторная форма
Для простой функции operator @ (<Аргумент>) operator @ (<Аргумент1>,<аргумент2>)	Для простой функции @<Аргумент> <Аргумент1>@<Аргумент2>
для компонентной функции <Аргумент>.operator@ () <Аргумент1>.operator@(<Аргумент2>)	для компонентной функции @<Аргумент> <Аргумент1>@<Аргумент2>

## 5.2 Описание компонентных функций-операторов

Как было сказано выше при описании компонентной функции-оператора первый аргумент всегда объект класса, второй при необходимости передается через параметры.

При переопределении операций следует различать ситуации, когда возвращается адрес (ссылка или указатель) уже существовавшего объекта или новый объект. В первом случае возвращаемое значение описывается как `<Имя класса> &` или `<Имя класса> *`, а функция-оператор должна вернуть `*this` или `this` соответственно. Во втором – возвращаемое значение указывается как `<Имя класса>`, в классе создается новый объект и функция-оператор должна вернуть этот объект.

**Пример 5.1.** Описание компонентной функции-оператора (класс Точка).

```
#include <iostream>
using namespace std;
class TPoint
{
private:    float x,y;
public:
    TPoint(float ax,float ay):x(ax),y(ay){}
    TPoint(){}
    void Out(void)
    {   cout<<"\n{"<<x<<" "<<y<<"\n"; }
    TPoint& operator-()           // -a – одноместная операция
    {   x=-x; y=-y; return *this; }
    TPoint& operator+=(TPoint &p) // a+=b – двуместная операция
    {   x+=p.x; y+=p.y; return *this; }
    TPoint operator+(TPoint &p)   // a+b – двуместная операция
    {   TPoint pp(x,y); return pp+=p; }
};
void main()
{
    TPoint p(2,3),q(4,5),r(7,8);
    -q;    q.Out();
```

```
p+=r;    p.Out();  
q=p+r;   q.Out();  
system("pause");  
}
```

Фактически при выполнении операций  $p+=r$ ,  $q=p+r$  выполняется еще одна операция – операция присваивания. Такая операция предусмотрена для всех типов по умолчанию. При выполнении этой операции происходит копирование полей методом «поле за полем». В рассмотренном классе отсутствуют динамические поля, поэтому стандартное определение операции присваивания переопределять не надо. При наличии динамических полей переопределение этой операции обязательно (см. раздел 5.4).

### 5.3 Описание внешних функций-операторов

Функции-операторы программируют как внешние, если необходимо, чтобы первый аргумент операции не был объектом класса. Внешней функции-оператору доступны только общие компоненты класса. Если необходимо, чтобы такая функция-оператор имела возможность обращаться к внутренним или скрытым компонентам класса, то ее следует описать в классе со спецификатором `friend`, определив ее дружественной этому классу.

Так при переопределении некоторых операций для обеспечения их коммутативности необходимо описывать несколько вариантов функций-операторов. Поскольку у компонентной функции-оператора первый параметр всегда объект, коммутативные операции обычно переопределяют как внешние.

**Пример 5.2.** Переопределение коммутативной операции «умножение на скаляр» (класс Целое число). Переопределение операций продемонстрируем на примере описания класса Целое число (см. пример 2.2), для которого переопределим операцию умножения целого числа на скаляр «\*».

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
typedef unsigned long dlong;
class Tlong
{
public:
    dlong    num;
    Tlong() {}
    Tlong(dlong an) { setnum(an); }
    friend Tlong & operator *(Tlong&,int k);
    friend Tlong & operator *(int k,Tlong&);
    void print(void);
    void setnum(dlong an) { num=an; }
};
```

```

Tlong & operator *(Tlong & ob1,int k)
{
    Tlong *temp=new Tlong;
    temp->num=ob1.num*k;
    return *temp;
}
Tlong & operator *(int k,Tlong &ob2)
{
    Tlong *temp=new Tlong;
    temp->num=ob2.num*k;
    return *temp;
}
void Tlong::print()
{ cout<<"Значение числа : "<<num<<endl;}
Tlong a(424567),c,d;      long int n;
void main()
{
    setlocale(0,"russian");
    d=a*2;
    cout<<"d=a*2:"; d.print();
    c=operator *(2,a);
    cout<<"c=2*a:"; c.print();
    system("pause");
}

```

Двуместная операция умножения для обеспечения коммутативности описана дважды: для случая, когда второй операнд – скаляр, и для случая, когда первый операнд – скаляр.

При работе с классами библиотеки ввода-вывода C++ следует помнить, что эти классы поддерживают операции «<<<» (включение в поток) и «>>>» (извлечение из потока) только для стандартных типов данных. Поэтому как для определенных пользователем типов полей классов, так и для классов целиком при выводе объектов этих классов операции «<<<» и «>>>» следует переопределять для каждого класса.

При переопределении операций «включение в поток» и «извлечение из потока» используется следующая форма описания операций:

```

ostream & operator<<(ostream & out, <Новый тип> <Имя>)
{
    <Тело функции-оператора> }
istream & operator >>(istream & in, <Новый тип> &<Имя>)
{
    <Тело функции-оператора> }

```

Причем при вовлечении в эти операции полей, описанных `private` и `protected`, операции «<<» и «>>» приходится описывать как дружественные.

**Пример 5.3.** Переопределение операций ввода-вывода (класс Целое число).

```

#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
typedef unsigned long dlong;
class Tlong
{
public:
    dlong    num;
    Tlong() {}
    Tlong(dlong an) { setnum(an); }
    friend ostream & operator <<(ostream &out, Tlong obj);
    friend istream & operator >>(istream & in, Tlong &obj);
    void print(void);
    void setnum(dlong an) { num=an; }
};
ostream & operator <<(ostream & out, Tlong obj)
{
    out<<" Значение числа :    "<<obj.num<<endl;
    return out;
}
istream & operator >>(istream & in, Tlong &obj)
{
    cout<<" Введите число    "<<endl;

```

```
        in>>obj.num;
        return in;
    }
void Tlong::print()
{   cout<<" значение числа : "<<num<<endl; }
void main()
{
    setlocale(0,"russian");
    Tlong a,b;
    cout<<"Ввод значений полей объектов"<<endl;
    cin>>a>>b;
    cout<<"ПРОСТОЙ ОБЪЕКТ  a"<<a<<endl;
    cout<<"ПРОСТОЙ ОБЪЕКТ  b"<<b<<endl;
    system("pause");
}
```

## 5.4 Особенности переопределения операции присваивания

Операция присваивания по умолчанию определена для любого класса и обеспечивает копирование полей объектов, аналогичное тому, которое выполняется по умолчанию в копирующем конструкторе. При необходимости эту операцию также можно переопределить, однако выполнение операции присваивания имеет свои особенности, которые следует учитывать.

**Пример 5.4.** Конструирование и уничтожение временных объектов при переопределении операции присваивания.

Чтобы отследить процессы конструирования и уничтожения объектов в том числе временных, переопределим также копирующий конструктор и деструктор. Непосредственной необходимости в их переопределении не существует, но их переопределение позволит уточнить последовательность выполняемых операций.

```
#include <iostream>
using namespace std;
class TP
{
private:    float x,y;
public:
    TP(float ax,float ay):x(ax),y(ay)
    {   cout<<"Constructor\n"; }
    TP()
    {   cout<<"Constructor without parameters\n"; }
    TP(TP &p)
    {   cout<<"Copy\n";  x=p.x; y=p.y;   }
    ~TP() {cout<<"Destructor\n"; }
    void Out(void)
    {   cout<<"\n{ "<<x<<" "<<y<<" }\n"; }
    TP& operator+=(TP &p) // a+=b
    {
        x+=p.x; y+=p.y;  cout<<"operator+=\n";
        return *this;
    }
};
```

```

    }
    TP operator+(TP &p)    // a+b
    {
        TP pp(x,y);  cout<<"operator+\n";
        return pp+=p;
    }
    TP& operator=(TP &p)    // a=b
    {
        x=p.x; y=p.y;    cout<<"operator=\n";
        return *this;
    }
};

void main()
{
    TP p(2,3),q(4,5),r(7,8);
    p+=r;    p.Out();
    q=p+r;    q.Out();
    system("pause");
}

```

Полученный результат – трассу конструирования/уничтожения объектов и выполнения функций операторов сопоставим текстам программ (см. табл. 5.3).

**Таблица 5.3.** Последовательность конструирования и уничтожения объектов при выполнении операций сложения и операции присваивания

Операция	Текст функции 1	Текст функции 2	Вывод
p+=r;	TP& operator+=(TP &p) { x+=p.x; y+=p.y; cout<<"+=\n"; return *this; }	-	Operator +=
q=p+r;	TP operator+(TP &p) { TP pp(x,y); cout<<"operator+\n";		Constructor Operator +

	<pre>pp+=p;  } TP&amp; operator=(TP &amp;p) { x=p.x; y=p.y; cout&lt;&lt;"operator=\n"; return *this; }</pre>	<pre>TP&amp; operator+=(TP &amp;p) { x+=p.x; y+=p.y; cout&lt;&lt;"+=\n"; return *this; }</pre>	<pre>Operator += <b>Copy</b> Destructor  Operator = <b>Destructor</b></pre>
--	--	--	---

При выполнении операции сложения создается новый объект. Этот объект, являясь автоматическим, уничтожается в момент завершения работы функции. Но он содержит результат. Анализ трассы показывает, что для передачи результата в основную программу компилятор C++ создает внутренний временный объект – копию результата. Эта копия передается функции, реализующей операцию присваивания, а затем – уничтожается. Операции создания объекта копии и его уничтожения в таблице выделены полужирным шрифтом.

## 5.5 Переопределение операций для объектов с динамическими полями. Контроль освобождения динамической памяти

Особо аккуратно следует переопределять операции для объектов с динамическими полями, поскольку при этом велика вероятность допустить «утечку памяти». В разделе 1.4 указано, что класс, описывающий объекты с динамическими полями, должен включать копирующий конструктор. По тем же причинам он должен переопределять операцию присваивания.

**Пример 5.5.** Переопределение операций для объектов с динамическими полями.

```
#include <string.h>
#include <stdio.h>
#include <iostream>
using namespace std;
class String
{
private:    char *str,name;        int  len;
public:    String(int Len,char Name); // пустая строка с памятью
          String(char *vs,char Name); // инициализированная строка
          String(String &S);         // копия строки
          ~String();                 // освобождение памяти
          int Length(){return len;}  // определение длины
          char operator[](int n)     // чтение символа из строки
          { return ((n>=0)&&(n<len))?str[n]:'\0'; }
          void print()               // вывод строки и ее длины
          {
              cout<<"Str: "<<name<<": ";
              cout<<str<<" Length: "<<len<<endl;
          }
          String operator+(String &A); // слияние строк
          String operator+(char c);    // добавление символа
          String& operator=(String &S); // присваивание строк
};
```

```

String::String(int Len,char Name)
{
    len=Len; str=new char[len+1];
    str[0]='\0'; name=Name;
    cout<<"Constructor length "<<name<<"\n";
}
String::String(char *vs,char Name)
{
    len=strlen(vs); str=new char[len+1];
    strcpy(str,vs); name=Name;
    cout<<"Constructor "<<name<<"\n";
}
String::String(String &S)
{
    len=S.Length(); str=new char[len+1];
    strcpy(str,S.str); name='K';
    cout<<"Copy from "<<S.name<<" to "<<name<<"\n";
}
String::~~String()
{
    delete [] str; cout<<"Destructor "<<name<<"\n";
}
String String::operator+(String &A)
{
    cout<<"Operation + "<<"\n"; int j=len+A.Length();
    String S(j,'S'); strcpy(S.str,str);
    strcat(S.str,A.str); cout<<"Operation + "<<"\n";
    return S;
}
String String::operator+(char c)
{
    cout<<"Operation +c"<<"\n"; int j=len+1;
    String S(j,'Q'); strcpy(S.str,str);
    S.str[len]=c; S.str[len+1]='\0';
    cout<<"Operation +c"<<"\n";
}

```

```

    return S;
}
String& String::operator=(String &S)
{
    cout<<"Operation = "<<"\n"; len=S.Length();
    if (str!=NULL) delete[]str;
    str=new char[len+1];
    strcpy(str,S.str);
    cout<<"Operation = "<<"\n";
    return *this;
}
void main()
{
    String A("ABC", 'A'), B("DEF", 'B'), C(6, 'C');
    C.print();
    C=A+B;
    C.print();
    C=C+'a';
    C.print();
    system("pause");
}

```

Проанализируем трассу выполнения операции сложения (см. табл. 5.4).

**Таблица 5.4.** Результат выполнения операции сцепления строк

Операция	Выполняемые операторы	Результаты
----------	-----------------------	------------

<pre>C=A+B;</pre>	<pre>String operator+(String &amp;A) { cout&lt;&lt;"Operation +"&lt;&lt;"\n";   int j=len+A.Length();   String S(j,'S');   strcpy(S.str,str);   strcat(S.str,A.str);   cout&lt;&lt;"Operation +"&lt;&lt;"\n";   return S; } String&amp; operator=(String &amp;S) {cout&lt;&lt;"Operation =" &lt;&lt;"\n";   len=S.Length();   if (str!=NULL) delete[]str;   str=new char[len+1];   strcpy(str,S.str);   cout&lt;&lt;"Operation ="&lt;&lt;"\n";   return *this; }</pre>	<pre>Operation + Constructor length S Operation + Copy from S to K Destructor S Operation = Operation = Destructor K</pre>
-------------------	--	--

Аналогично предыдущему примеру передача результата операции присваивания осуществляется через временный объект – копию переменной, хранящей результирующее значение.

### Вопросы для самоконтроля

1. Что такое переопределение операций?

[Ответ.](#)

2. Какие операции можно переопределять?

[Ответ.](#)

3. Чем отличаются компонентные и внешние функции-операторы?

[Ответ.](#)

## 6 Шаблоны

### 6.1 Параметризованные функции

*Шаблон функции* – это глобальная функция, определенная за пределами классов. В отличие от перегрузки функции, при которой для каждой сигнатуры создается своя функция, шаблон семейства функций определяется один раз, но это описание содержит параметры. Для задания параметров используется список:

```
template <Список параметров шаблона> <Описание функции>
```

Каждый формальный параметр шаблона обозначается ключевым словом `class`, за которым следует имя параметра. Например, описание шаблона функции, возвращающей значение максимальной из двух переменных, выглядит следующим образом:

```
template <class type>
type maxx(type x, type y) { return (x>y)?x:y; }
```

где `type` – параметр – тип сравниваемых значений.

Использование шаблона функции позволяет передать в нее в качестве параметра тип используемых данных, а далее выполнять операции, предусмотренные алгоритмом над объектами заданных типов. Если для некоторых типов объектов операции, используемые в функции, не определены, следует ввести явное описание функции для этого типа. Например, при использовании шаблона из предыдущего описания, если в качестве аргумента будут использованы строки, то, так как операция «>» для строк не определена, функция выдаст неправильный результат. Для того чтобы в качестве параметра шаблона можно было использовать строки, следует добавить явное описание функции-оператора «>» для строк.

**Пример 6.1.** Использование шаблонов функций при создании шаблонов классов.

```
#include <locale.h>
#include <conio.h>
```

```

#include <string.h>
#include <iostream>
using namespace std;
template <class T> T maxx(T x, T y)
{ return(x>y)?x:y; }
char * maxx(char * x, char * y) // функция для строк
{ return strcmp(x,y)>0?x:y; }

void main ()
{
    setlocale(0, "russian");
    int a=1,b=2;
    char c='a',d='m';
    float e=123.67,f=456.67;
    char str[]="abcd", str2[]="abnd";
    // вызов функции для объектов различного типа
    cout << "Integer max= " << maxx(a,b) << endl;
    cout << "Character max= " << maxx(c,d) << endl;
    cout << "Float max= " << maxx(e,f) << endl;
    cout << "String max= " << maxx(str,str2) << endl;
    system("pause");
}

```

Каждый аргумент шаблона функции должен определять тип как минимум одного аргумента описываемой функции. Это гарантирует, что нужный вариант функции будет выбран на основе анализа типов ее аргументов. Отмеченное правило не распространяется на шаблоны классов.

## 6.2 Параметризованные классы

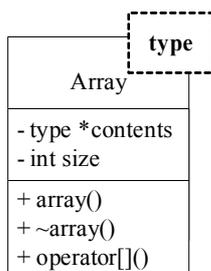
В языке C++ предусмотрена реализация еще одного типа полиморфизма – параметризованных классов и параметризованных функций.

Параметризованный класс – это некий шаблон (template), на основе которого можно строить другие классы. Этот шаблон следует рассматривать как описание множества классов, моделирующих абстрактную структуру данных и отличающихся типами полей, включенных в эту структуру. Хорошим примером параметризованных классов могут служить шаблоны реализации списков, массивов, множеств и т. п. Шаблон классов определяет правила построения каждого отдельного класса из множества разрешенных (допустимых) классов. Описание шаблона выглядит так:

```
template <Список параметров шаблона> <Описание класса>
```

В списке параметров шаблона могут присутствовать как параметры, определяющие тип, так и параметры, для которых этот тип фиксирован. Каждый формальный параметр, определяющий тип, обозначается ключевым словом `class`.

**Пример 6.2.** Шаблон, позволяющий формировать одномерные динамические массивы из элементов различных типов (рис. 6.1).



**Рис. 6.1.** Условное обозначение шаблона `array`

```
#include <locale.h>
#include <stdio.h>
#include <iostream>
using namespace std;
template <class type> // объявление шаблона с аргументом "type"
class array
{
```

```

type * contents; // указатель на динамический массив типа type
int size;        // размер массива
public:
array(int number)
{ contents = new type [size=number];}
~array () { delete [] contents;}
type & operator [] (int x) // переопределение операции []
{
    if ((x<0) || (x>=size))
    { cerr << "ошибочный индекс"; x=0;}
    return contents[x];
}
};

```

Используя шаблон, можно определить объекты класса `array` с аргументом любого допустимого типа. Допустимым являются как встроенные типы языка, так и типы, определенные пользователем.

Объявление объекта одного из классов, порождаемых шаблоном, выглядит следующим образом:

Имя параметризованного класса <Список параметров шаблона>

Имя объекта (Параметры конструктора)

Например, для шаблона классов, описанного в примере 6.2, можно объявить следующие объекты:

```

array <int> int_a(50); // массив элементов целого типа
array <char> char_a(100); // массив элементов типа char
array <float> float_a(30); // массив элементов типа float

```

Описание шаблона можно записать в файл с расширением «`h`» и, используя его стандартным образом, определять объекты заданного множества классов, а также выполнять операции над этими объектами.

**Пример 6.2 (продолжение).** Использование шаблона для формирования массивов и вывода их элементов.

```
#include "array.h"
#include <stdlib.h>
#include <time.h>
void main()
{
    setlocale(0, "russian");
    int i;
    srand( (unsigned)time( NULL ) );
    array<int> int_a(5);
    array<char> char_a(5);
    for (i=0;i<5;i++) // определение компонент массивов
    { int_a[i]=rand()/1000+10; char_a[i]='A'+i;}
    puts("Компоненты массивов");
    for (i=0;i<5;i++)
    { cout << " " << int_a[i]<< " " <<char_a[i]<<endl;}
    system("pause");
}
```

Функции параметризованного класса можно определить и вне описания класса, но следует учитывать, что такие функции являются параметризованными и их надо описывать как функции-шаблоны (параметризованные компонентные функции), например:

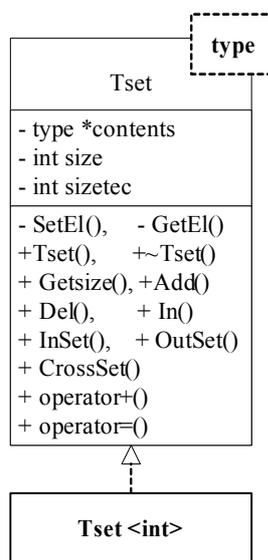
```
template <class type>
type & array <type>::operator[](int x) /* заголовок
                                     функции-шаблона*/
{
    if ((x<0)|| (x>=size))
    { cerr << "Ошибочный индекс"; x=0;}
    return contents[x];
}
```

**Пример 6.3.** Использование шаблона классов (шаблон классов «Множество»). Пусть необходимо разработать шаблон классов для реализации объектов типа «множество». Шаблон должен обеспечивать возможность выполнения следующих операций: построения множества из заданных элементов, добавления элементов, удаления элементов, пересечения множеств, объединения множеств, присваивания множеств и проверки вхождения элемента во множество.

Память под элементы множества целесообразно распределять в процессе работы (объект со скрытым динамическим полем типа «массив»), используя для этого конструктор. Соответственно деструктор должен освобождать память целиком (как она выделяется).

Для корректной работы с множеством доступ к его элементам осуществляется с помощью внутренних функций. Чтобы присвоение одного объекта другому происходило с выделением памяти под массив элементов объекта приемника, предусмотрена переопределенная операция присваивания. Типовые операции над множествами реализованы несколькими способами: через внешние операторы-функции, компонентные операции и в виде обычных компонентных функций. Так как возможна инициализация объекта другим объектом-множеством, предусмотрен копирующий конструктор.

Шаблон описывает правила выполнения всех указанных выше операций относительно параметра `type` (рис. 6.2).



**Рис. 6.2.** Структура классов для примера 6.3

Тестирующая программа объявляет объект класса Множество с элементами типа `char` и проверяет выполнение всех операций.

```

#include <locale.h>
#include <iostream>
#include <string.h>
using namespace std;
template <class type>      // объявление шаблона с аргументом type
class Tset
{
private:
    type *contents;      // указатель на динамический массив типа type
    int size;           // максимальный размер множества
    int SetEl(int ind, type m) // функция записи элемента в массив
    {
        if (ind<size)
        {   contents[ind]=m; sizetek+=1; return 1;   }
        else
        {
            cout<<"Превышен размер множества "<<endl;
            return 0;
        }
    }
public:
    int sizetek;        // текущий размер множества
    type GetEl(int ind) // функция чтения элемента из массива
    {   return contents[ind]; }
    Tset(){ contents=NULL; }
    Tset(Tset &A);      // прототип копирующего конструктора
    Tset(int number)    // конструктор класса Tset
    {   contents = new type [size=number];sizetek=0; }
    ~Tset(){ if (contents!=NULL) delete [] contents; }
    int Getsize()       // максимальный размер множества
    {   return size;   }
    void Add(type m);    // добавление элемента к множеству
    void Del(type m);    // удаление элемента из множества
    int In(type m);     // проверка вхождения элемента во множество

```

```

int   InSet(int Nn);      // заполнение множества
void  OutSet();          // вывод элементов множества
Tset& CrossSet(Tset &B); // пересечение множеств
Tset &operator +(Tset &B); // объединение множеств
Tset &operator =(Tset &B); // присваивание множеств
};

template <class type> Tset<type>::Tset(Tset<type>&A)
{
    contents = new type [size=A.GetSize()];
    sizetek=A.sizetek;
    for(int i=0;i<sizetek;i++)
        contents[i]=A.contents[i];
}

template <class type> int Tset<type>::InSet(int Nn)
{
    type c; int k=0;
    if (Nn>GetSize()) Nn=GetSize();
    cout<<"Введите "<<Nn<<" элементов множества: "<<endl;
    for (int i=0;i<Nn;i++)
    {
        cin>>c;
        if (!In(c))
            { if(!SetEl(k,c)) return 0; k+=1;}
    }
    return 1;
}

template <class type> void Tset<type>::OutSet()
{
    if (sizetek!=0)
    {
        for(int i=0;i<sizetek;i++)
            cout<<" "<<GetEl(i);
    }
    else cout<<" Пустое множество";
}

```

```

        cout<<endl;
    }
    template <class type> int Tset<type>::In(type m)
    {
        for (int i=0;i<sizetek;i++)
            if (m==GetEl(i)) return 1;
        return 0;
    }
    template <class type> void Tset<type>::Add(type m)
    {
        if (!In(m))
            if (sizetek<size) SetEl(sizetek,m);
    }
    template <class type> void Tset<type>::Del(type m)
    {
        int h;
        if (In(m))
        {
            h=0;
            for(int i=0;i<sizetek;i++)
                if(h) contents[i-1]=contents[i];
                else if (m==GetEl(i)) h=1;
            sizetek-=1;
        }
    }
    template <class type>
    Tset<type>& Tset<type>::operator =(Tset<type> &B)
    {
        if (this==&B) return *this;
        if (contents!=NULL) delete [] contents;
        contents = new type [size=B.GetSize()];
        sizetek=B.sizetek;
        for(int i=0;i<sizetek;i++)
            { contents[i]=B.contents[i];}
        return *this; }

```

```

template <class type>
Tset<type> & Tset<type>::operator +(Tset<type> &B)
{
    for(int i=0;i<B.sizetek;i++)
        Add(B.GetEl(i));    return *this;
}

template <class type>
Tset<type> &Tset<type>::CrossSet(Tset<type> &B)
{
    int i=0;
    do
    {
        if( !B.In(GetEl(i)))
            Del(GetEl(i));
        else i++;
    } while (i<sizetek);
    return *this;
}

template <class type>
Tset<type>& operator -(Tset<type> &A,Tset<type> &B)
{
    Tset<char> *C=new Tset<char>(A.GetSize());
    for(int i=0;i<A.sizetek;i++)
        if(!B.In(A.GetEl(i)))    C->Add(A.GetEl(i));
    return *C;
}

template <class type>
Tset<type>& operator*(Tset<type>&A,Tset<type> &B)
{
    int l;
    if(A.GetSize() > B.GetSize()) l=A.GetSize();
    else l=B.GetSize();
    Tset<char> *C=new Tset<char>(l);
    for(int i=0;i<A.sizetek;i++)
    {

```

```

        if( B.In(A.GetEl(i)) ) C->Add(A.GetEl(i));
    }
    return *C;
}
void main()
{
    setlocale(0,"russian");
    int n;
    Tset<char> aa(15),bb(10),dd(10),cc;
    cout<<"Введите мощность множества n<= ";
    cout<<aa.GetSize()<<endl;    cin>>n;
    aa.InSet(n);
    cout<<"Введите мощность множества n<= ";
    cout<<bb.GetSize()<<endl;    cin>>n;
    bb.InSet(n);
    cout<<"Введите множества множества n<= ";
    cout<<dd.GetSize()<<endl;    cin>>n;
    dd.InSet(n);
    cout<<" Множество aa: "; aa.OutSet();
    cout<<" Множество bb: "; bb.OutSet();
    cout<<" Множество dd: "; dd.OutSet();
    Tset<char> ee,pp=aa; // использование копирующего конструктора
    cout<<" Множество pp=aa: "; pp.OutSet();
    ee=aa*bb;
    cout<<"ee=aa*bb="; ee.OutSet();
    aa.CrossSet(bb);
    cout<<"aa.CrossSet(bb)="; aa.OutSet();
    aa=aa+dd;
    cout<<"aa=aa+dd="; aa.OutSet();
    cc=dd-bb;
    cout<<"cc=dd-bb"; cc.OutSet();
    system("pause");
}

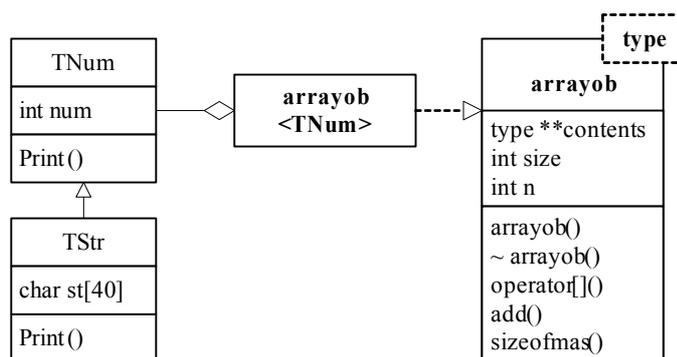
```

Тестирующая программа осуществляет формирование трех множеств букв и операции над ними, а также фиксирует последовательность и место вызова конструкторов различного назначения и деструкторов.

### 6.3 Контейнеры на основе шаблонов

Построение контейнеров на основе шаблонов – интересная задача. Такой контейнер позволяет организовать хранение элементов самых разных типов данных.

**Пример 6.4.** Контейнер на основе шаблона. Реализуем программу, использующую шаблон Динамический массив для организации динамического массива, хранящего объекты классов Число и Строка (рис. 6. 3).



**Рис. 6.3.** Иерархия классов для примера 6.4

```

#include <locale.h>
#include <string.h>
#include <iostream>
#include <stdlib.h>
using namespace std;
template <class type> // объявление шаблона класса
class arrayob // класс Динамический массив
{
    type **contents; // массив указателей на объекты типа type
    int size; // максимальное количество объектов в массиве
    int n; // реальное количество объектов в массиве
public:
    arrayob(int number)
    { contents=new type * [size=number];}
    ~arrayob ();
    void add(type *p)
    {

```

```

        if(n==size)cerr<<"Выход за пределы";
        else {contents[n]=p; n++;}
    }
    type & operator [] (int x)    // итератор массива объектов
    {
        if ((x<0)|| (x>=size))
        { cerr <<"Ошибка индекс " <<x<<endl;x=0;}
        return *contents[x];
    }
    int sizeofmas(){return n;} //реальный размер массива
};
template <class type> arrayob<type>::~~arrayob ()
{
    for(int i=0;i<size;i++) delete contents[i];
    delete []contents;
}
class TNum                // класс Число
{
public:    int num;
    virtual void Print(void) { cout<<num<<" "; }
    TNum(){cout<<"Введите число"<<endl; cin>>num;}
    TNum(int n):num(n) {}
    virtual ~TNum(void){}
};
class TStr:public TNum    // класс Строка
{
public:    char *st;
    virtual void Print(void) { TNum::Print(); cout<<st<<" ";}
    TStr();    // конструктор по умолчанию
    TStr(char *s):TNum(strlen(s))// конструктор с параметрами
    {
        st=new char[num+1];    strcpy(st,s);
        st[num]='\0';
    }
};

```

```

        virtual ~TStr(void){ delete [] st;}
};
TStr::TStr():TNum(40)
{
    cout<<"введите строку "<<endl;
    st=new char[num+1];    cin>>st;
    num=strlen(st);    st[num+1]='\0';
}
arrayob<TNum>  ob_a(5); //массив из 5 указателей на объекты
void main()
{
    setlocale(0,"russian");
    int i;
    for(i=0;i<5;i++)
        if (i/2*2==i)
            ob_a.add(new TNum);    // добавить Число
        else ob_a.add(new TStr);    // добавить Строку
    cout<<"Содержимое контейнера "<<"\n";
    for (i=0;i<ob_a.sizeofmas();i++)
        ob_a[i].Print();
    system("pause");
}

```

Шаблон оперирует указателями на объекты иерархии. Для вызова метода Print() и деструкторов классов используется механизм сложного полиморфизма.

### Вопросы для самоконтроля

1. Что такое шаблон? Определите понятие шаблона функции и шаблона класса. Приведите примеры применения шаблонов классов.

[Ответ.](#)

2. Сопоставьте понятия «параметризованные» и «контейнерные» классы. Чем определяется выбор того или иного типа классов в конкретном случае?

[Ответ.](#)

3. Определите, чем контейнеры, основанные на иерархии классов, отличаются от шаблонов, базирующихся на шаблонах.

[Ответ.](#)

## 7 Исключения

Практика разработки сложных программных систем позволяет сделать вывод, что достаточно большая часть любой программы приходится на перехват и обработку ситуаций, при возникновении которых по каким-либо причинам нормальный процесс обработки нарушается (ввод некорректной информации, попытка читать из несуществующего файла, обнаружение ситуации «деление на нуль» и т. п.).

Использование для проектирования программы технологии ООП приводит к тому, что обычно возникновение некорректной ситуации фиксируется в одном месте (объекте) программы, а ее исправление необходимо осуществлять в другом. Для обработки таких ситуаций применяют механизм исключений.

К сожалению, традиционно в С и С++ используются разные стандарты обработки исключений.

### 7.1 Механизм исключений С++

В языке С++ для работы с исключениями существуют специальные операторы `throw`, `try` и `catch`. Первый – для генерации исключения, а два других – для организации его перехвата.

Генерация исключений выполняется в том месте программы, где обнаруживается исключительная ситуация.

Оператор `throw` имеет следующий синтаксис:

```
throw [<Тип>](<Аргументы>);
```

где `<Тип>` – тип (чаще класс) генерируемого значения; если тип не указан, то компилятор определяет его исходя из типа аргумента (обычно это один из встроенных типов);

`<Аргументы>` – одно или несколько выражений, значения которых будут использованы для инициализации генерируемого объекта.

Например:

```
1) throw ("Неверный параметр"); /* генерирует исключение типа
```

```

const char * с указанным в кавычках значением */
2) throw (221);          /* генерирует исключение типа const int
                           с указанным значением */
3) class E { //класс исключения
    public: int num;    // номер исключения
        E(int n): num(n) {} // конструктор класса
    }
...
throw E(5); // генерирует исключение в виде объекта класса E

```

Перехват и обработка исключений осуществляются с помощью конструкции `try ... catch ... (catch...)`:

```

try {<Защищенный код>}
catch (<Ссылка на тип>)
{
    <Обработка исключений>
}

```

Блок операторов `try` содержит операторы, при выполнении которых могут возникнуть исключительные ситуации. Блоков `catch` может быть несколько. Каждый блок `catch` включает операторы, которые должны быть активизированы, если при выполнении операторов блока `try` было зафиксировано исключение типа, совместимого с указанным в `catch`. При этом:

<sup>35</sup><sub>17</sub>исключение типа `T` будет перехватываться обработчиками типов `T`, `const T`, `T&` или `const T&`;

<sup>35</sup><sub>17</sub>обработчики типа общедоступного базового класса перехватывают исключения типа производных классов;

<sup>35</sup><sub>17</sub>обработчики типа `void*` перехватывают все исключения типа указателя.

Блок `catch`, для которого в качестве типа указано «...» обрабатывает исключения всех типов.

Например:

```

try {<Операторы>} // выполняемый фрагмент программы
catch (EConvert& A) {<Операторы>} /* перехватывает исключения
                                     типа EConvert */
catch (char* Mes) {<Операторы>} // перехватывает исключения типа char*
catch (...) {<Операторы>} // перехватывает остальные исключения

```

Таким образом, обработчик может перехватывать исключения нескольких типов. При этом существенным оказывается порядок объявления обработчиков исключений.

Так, обработчик типа `void*` не должен указываться перед обработчиком типа `char*`, потому что при таком порядке все исключения типа `char*` будут обрабатываться обработчиком `void*`. То же самое касается недопустимости указания обработчика исключений базового класса перед обработчиками производных классов.

Например:

```

class E{};
class EA:public E{};    ...
try {...}
catch (E& e) {...} // этот обработчик перехватит все исключения
catch (EA& e) {...} // этот обработчик никогда не будет вызван

```

Иногда оказывается, что перехваченное исключение не может быть обработано в данном месте программы. В этом случае в обработчик включается оператор `throw` без параметра, в результате исключение генерируется повторно с тем же объектом, что и первый раз. При обработке этого исключения возобновляется просмотр стека вызовов в целях обнаружения другого обработчика данного или совместимого типа.

**Пример 7.1.** Создание класса исключения. Многократный перехват исключений.

```

#include <stdlib.h>
#include <iostream>
using namespace std;
class E{}; // класс исключения
void somefunc(bool ccc) // функция, генерирующая исключение
{
    if(ccc) throw E();

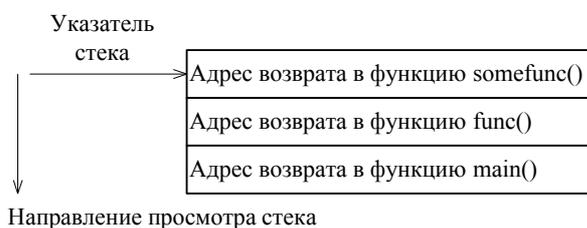
```

```

}
void func()
{
    try
    {
        somefunc(true);
    }
    catch(E& e) // первый обработчик исключения
    {
        /*если здесь исключение обработать нельзя, то возобновляем его*/
        if (true) throw;
    }
}
void main()
{
    try { func(); }
    catch(E& e) // второй обработчик исключений
    {
        // здесь обрабатываем исключение
        cout<<"Exception"<<endl;
    }
    system("pause");
}

```

Стек вызовов для данного примера показан на рис. 7.1.



**Рис. 7.1.** Содержимое стека вызовов в момент генерации исключения

Функция `somefunc()` генерирует исключение. Для его обработки осуществляется обратный просмотр стека вызовов, т. е. по очереди проверяются все функции, выполнение которых не завершено. При этом обнаруживается, что вызов `somefunc()` осуществлен в защищенной конструкции `try` блоке функции `func()`, и, следовательно, проверяется соответствие типа исключения типу имеющегося обработчика. Тип соответствует,

следовательно, исключение перехвачено, но, если оно не может быть обработано в данном обработчике, то исключение будет инициировано вновь. Теперь в поисках обработчика исключения будет проверяться следующая незавершенная функция – `main()`. В этой функции обнаруживается, что вызов `func()` выполнялся в защищенном блоке. При проверке связанного с ним блока `catch` выясняется, что данное исключение перехватывается и обрабатывается.

Использование имени переменной в качестве параметра оператора `catch` позволяет операторам обработки получить доступ к аргументам исключения через указанное имя. Например:

```
class E                //класс исключения
{
    public: int num;    // номер исключения
        E(int n) : num(n) {} // конструктор
}
...
throw E(5);          // генерируемое исключение
...
catch (E& e)         // перехват исключения E
{
    if (e.num==5) // получен доступ к полю
        {...}
}
```

Полностью последовательность обработки исключения выглядит следующим образом:

- 1) при генерации исключения происходит конструирование временного объекта исключения;
- 2) выполняется поиск обработчика исключения;
- 3) при нахождении обработчика создается копия объекта исключения с указанным именем;
- 4) уничтожается временный объект исключения;
- 5) выполняется обработка исключения;
- 6) уничтожается копия исключения с указанным именем.

Поскольку обработчику передается копия объекта исключения, желательно в классе исключения со сложной структурой предусмотреть копирующий конструктор и деструктор.

Иногда бывает удобно указать при объявлении функции, какие исключения она может генерировать. Именно для этих исключений программист будет предусматривать обработчики при вызове функции. Указание генерируемых исключений осуществляется в спецификации исключений:

```
throw (<Тип>, <Тип>...)
```

Например:

```
void func () throw(char*,int) {...} /*данная функция может генерировать
                                     исключения типов char* и int */
```

При организации перекрытия виртуальных методов следует учитывать, что спецификация исключений не считается частью типа функции и, следовательно, ее можно изменить:

```
class ALPHA
{
    public:
        struct ALPHA_ERR {};
        virtual void vfunc()
            throw (ALPHA_ERR) {} // спецификация исключения
};
class BETA : public ALPHA
{
    public:
        void vfunc() throw(char *) {} // изменение спецификации
};
```

Если в процессе выполнения программы будет сгенерировано исключение не предусмотренного спецификацией типа, то управление будет передано специальному

обработчику *непредусмотренных исключений*. Для определения этой функции в программе используется функция `set_unexpected()`:

```
void my_unexpected()
{ <Обработка исключений> }
...
set_unexpected(my_unexpected);
```

Функция `set_unexpected()` возвращает ранее используемый адрес функции – обработчика непредусмотренных исключений, что позволяет организовать достаточно сложную обработку.

Если нужный обработчик при обратном просмотре стека вызовов не найден, а обработчик непредусмотренных исключений отсутствует, то вызывается функция автоматически создаваемая функция `terminate()`. По умолчанию эта функция вызывает функцию `abort()`, которая аварийно завершает текущий процесс.

Автоматически создаваемую функцию `terminate()` можно заменить собственной. Для этого используют функцию `set_terminate()`:

```
void my_terminate() {<Собственная обработка завершения>}
...
set_terminate(my_terminate);
```

Функция `set_terminate()` возвращает адрес предыдущей программы обработки завершения, что позволяет при необходимости вернуть ей управление.

## 7.2 Механизм исключений C

В языке C используется так называемое *структурное управление исключениями*. Именно данный способ обработки исключений используется при необходимости перехватить исключения от операционной системы Win32.

В основе структурного управления исключениями лежат конструкции `__try...__except` и `__try...__finally`. Первая обеспечивает обычную обработку исключения, вторая – завершающую.

Обычная обработка программируется следующим образом:

```
__try {<Защищенный код>}
__except (<Фильтрующее выражение>)
    {<Обработка исключений>}
```

Фильтрующее выражение может принимать следующие значения:

<sup>35</sup><sub>17</sub> `EXCEPTION_EXECUTE_HANDLER` – управление должно быть передано на следующий за ним обработчик исключения. В этом случае по умолчанию при обратном просмотре стека вызовов активизируются деструкторы всех локальных объектов, созданных между местом генерации исключения и найденным обработчиком;

<sup>35</sup><sub>17</sub> `EXCEPTION_CONTINUE_SEARCH` – производится поиск другого обработчика;

<sup>35</sup><sub>17</sub> `EXCEPTION_CONTINUE_EXECUTION` – управление возвращается в то место, где было обнаружено исключение без обработки исключения (отмена исключения).

Как правило, в качестве фильтрующего выражения используется функция, которая возвращает одно из трех указанных выше значений.

Библиотека `except.h` включает также функции, позволяющие получить некоторую информацию об исключении:

`GetExceptionCode()` – возвращает код исключения.

`GetExceptionInformation()` – возвращает указатель на структуру, содержащую описание исключения.

Существует ограничение на вызов этих функций: они могут вызываться только непосредственно из блока `__except()`, например:

```
#include <except.h>
int filter_func(EXCEPTION_POINTERS *);
...
EXCEPTION_POINTERS *xp = 0;
    __try {          foo();      }
    __except (filter_func(xp = GetExceptionInformation()))
    { /* получение информации об исключении */ }
```

или с использованием составного оператора:

```
__except((xp = GetExceptionInformation()), filter_func(xp))
```

Фильтрующая функция не может вызывать функцию

`GetExceptionInformation()`, но результат этой функции можно передать в качестве параметра.

Функция `GetExceptionInformation()` возвращает указатель на структуру `EXCEPTION_POINTERS`:

```
struct EXCEPTION_POINTERS
{
    EXCEPTION_RECORD *ExceptionRecord;
    CONTEXT *Context;
};
```

Структура `EXCEPTION_RECORD` в свою очередь определяется следующим образом:

```
struct EXCEPTION_RECORD
{
    DWORD ExceptionCode; // код завершения
    DWORD ExceptionFlags; // флаг возобновления
    struct EXCEPTION_RECORD *ExceptionRecord;
```

```

void *ExceptionAddress;    // адрес исключения
DWORD NumberParameters; // количество аргументов
DWORD ExceptionInformation
    [EXCEPTION_MAXIMUM_PARAMETERS]; // адрес массива параметров
};

```

Обычно фильтрующая функция обращается к информации `ExceptionRecord`, чтобы определить, следует ли обрабатывать исключение данным обработчиком. Но иногда этой информации обработчику исключения оказывается недостаточно, и тогда используют поля, собранные в структуру `CONTEXT`.

Например, если исключение не обрабатывается, а управление возвращается обратно в точку генерации исключения (значение фильтрующего выражения равно `EXCEPTION_CONTINUE_EXECUTION`), то при возврате вновь возникнет то же исключение. Изменив соответствующие поля структуры `CONTEXT`, мы избегаем замкнутого круга, например:

```

static int xfilter(EXCEPTION_POINTERS *xp)
{
    int rc;
    EXCEPTION_RECORD *xr = xp->ExceptionRecord;
    CONTEXT *xc = xp->Context;
    switch (xr->ExceptionCode)
    {
        case EXCEPTION_BREAKPOINT:
            ++xc->Eip; /* в коде программы остались встроенные
                       точки останова, перешагнем через них,
                       изменив адрес команды на 1байт */
            rc = EXCEPTION_CONTINUE_EXECUTION;
            break;
        case EXCEPTION_ACCESS_VIOLATION:
            rc = EXCEPTION_EXECUTE_HANDLER;
            break;
        default: // продолжить поиск обработчика
            rc = EXCEPTION_CONTINUE_SEARCH;
    }
}

```

```

        break;
    };
    return rc;
}
...
EXCEPTION_POINTERS *xp;
__try
{
    func();
}
__except(xfilter(xp = GetExceptionInformation()))
{
    abort();
}

```

Для генерации структурного исключения используется специальная функция:

```

void RaiseException(DWORD <Код исключения>,
    DWORD <Флаг>,
    DWORD <количество аргументов>,
    const DWORD *<адрес массива 32-разрядных аргументов>);

```

где <флаг> может принимать значения:

0 – исключение возобновимо;

EXCEPTION\_NONCONTINUABLE – исключение не возобновимо.

**Пример 7.2.** Обработка структурного исключения.

```

#include <string.h>
#include <stdlib.h>
#include <windows.h>
#include <excpt.h>
#include <iostream>
using namespace std;
#define MY_EXCEPTION 0x0000FACE
void func()

```

```

{
    RaiseException(MY_EXCEPTION, 0, 0, 0); // исключение
}
DWORD ExceptionFilter(DWORD dwCode)
{
    if (dwCode==MY_EXCEPTION) // фильтрация исключения
        return EXCEPTION_EXECUTE_HANDLER;
    else return EXCEPTION_CONTINUE_SEARCH;
}
void somefunc()
{
    __try { func(); }
    __except(ExceptionFilter(GetExceptionCode()))
    {
        cout<<"MyException"<<endl;
    }
}
void main()
{
    somefunc();
    system("pause");
}

```

Структурное управление исключениями поддерживает также завершающую конструкцию, которая выполняется независимо от того, было ли обнаружено исключение при выполнении защищенного блока.

**Пример 7.3.** Совместное применение простой и завершающей обработки исключений.

```

#include <stdio.h>
#include <windows.h>

void main()
{
    puts("hello");
}

```

```

__try
{
    puts("in try");
    __try
    {
        puts("in try");
        RaiseException(0,0,0,0); // генерация исключения
    }
    __finally
    {
        puts("in finally");
    }
}
__except( puts("in filter"), EXCEPTION_EXECUTE_HANDLER )
{
    puts("in except");
}
puts("world");
system("pause");
}

```

Результат работы программы:

```

hello
in try
in try
in filter
in finally
in except
world
Для продолжения нажмите любую клавишу . . .

```

При выполнении завершающей обработки, так же как и при обычной обработке, вызываются деструкторы созданных локальных объектов. То же самое происходит, если

исключения остаются необработанными. Локальные объекты не уничтожаются только в том случае, если необработанным остается исключение Win32.

### 7.3 Совместное использование различных механизмов обработки исключений

При программировании обработки исключений операционной системы Win32 следует иметь в виду, что:

1) исключения Win32 можно обрабатывать только `__try...__except` или, соответственно, `__try...__finally`; оператор `catch` эти исключения игнорирует;

2) неперехваченные исключения Win32 не обрабатываются функцией обработки неперехваченных исключений и функцией `terminate()`, а передаются операционной системе, что обычно приводит к аварийному завершению приложения.

В свою очередь, исключения C++ не видимы для `__except`.

Если возникает необходимость перехвата структурных исключений и исключений C++ для одной последовательности операторов, то соответствующие конструкции вкладываются одна в другую.

**Пример 7.4.** Совместная обработка исключений различных типов. Рассмотрим организацию совместной обработки исключения Win32 «Деление на ноль в вещественной арифметике» и исключения C++.

В файле `Exception.h` определим класс исключения и функцию, использующую это исключение:

```
#ifndef ExceptionH
#define ExceptionH
class MyException // класс Мое исключение
{
private: char* what; // поле сообщения
public:
    MyException(char* s);
    MyException(const MyException& e);
    ~MyException();
    char* msg()const;
};
void func();
#endif
```

Тела методов опишем в файле Exception.cpp:

```
#include <string.h>
#include "Exception.h"
MyException::MyException(char* s = "Unknown")
{ what = strdup(s); }
MyException::MyException(const MyException& e )
{ what = strdup(e.what); }
MyException::~MyException()
{ delete[] what; }
char* MyException::msg() const
{ return what; }
```

Вызов защищенной функции осуществляем в основной программе:

```
#include <stdlib.h>
#include <windows.h>
#include <iostream>
#include "exception.h"
using namespace std;
void func1()
{
    int *p; // неинициализированный адрес
    __try
    {
        *p = 13; // ошибка!!
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        // перевод исключения Win32 в исключение C++
        throw(MyException("Wrong adresing!"));
    }
}
void func2()
{
```

```
try
{
    func1();
}
catch(const MyException& e)
{
    cout<<e.msg()<<endl;
}
void func()
{
    __try
    {
        func2();
    }
    __finally { cout<<"Finally execution"<<endl; }
}
void main()
{
    func();
    system("pause");
}
```

### Вопросы для самоконтроля

1. Какие средства обработки ошибок были включены в базовую объектную модель C++? Как их можно использовать?

[Ответ.](#)

2. Почему в C и C++ сосуществуют два различных механизма обработки исключений? Расскажите о каждом из них. В каких случаях они используются? Возможно ли их совместное применение?

[Ответ.](#)

## 8 Создание и уничтожение динамических объектов.

### «Умные указатели»

#### 8.1 Проблемы работы с динамическими объектами

Современное программирование предполагает интенсивную работу с динамической памятью. Однако работая с указателями программисты, особенно начинающие, допускают значительное количество ошибок, приводящих к нарушениям защиты и «утечкам» памяти. Причины этого кроются в особенностях языковых средств, используемых при работе с динамическими объектами. Так:

- при использовании указателей ответственность за выделение памяти под объект и ее освобождение целиком лежит на программисте;
- на один и тот же объект может ссылаться несколько не связанных между собой указателей, следовательно возможно наличие указателей, ссылающихся на освобождённые или перемещённые объекты;
- нет никакой возможности проверить, указывает ли ненулевой указатель на корректные данные, либо «в никуда»;
- указатель на единственный объект и указатель на массив объектов в языке C++ никак не отличаются друг от друга.

Разработать код, который корректно освобождает выделенную динамическую память, непросто. При этом надо предусмотреть освобождение памяти не только при нормальном вычислительном процессе, но и при аварийном завершении.

Так простейшая процедура, в которой создается динамический объект и вызывается его метод:

```
void proc ()
{
    B *temp = new B;    // выделение памяти под объект
    B->func ();        // вызов метода для объекта
    delete temp;       // освобождение памяти
}
```

с учетом необходимости освобождения памяти при аварийном завершении существенно удлиняется:

```
void proc ()
{
    B *temp = NULL;
    try
    {
        temp = new B; // выделение памяти под объект
        B->func ();    // вызов метода для объекта
    }
    catch (...)      // перехват всех исключений
    {
        delete temp; // освобождение памяти при аварийном
        // завершении процедуры
        throw;       // возобновление исключения
    }
    delete temp;     // освобождение памяти при нормальном
    // завершении процедуры
}
```

Еще сложнее обеспечить освобождение памяти при возникновении ошибки в конструкторе. Язык C++ обеспечивает корректное освобождение памяти, выделенной под объект, при аварийном завершении программы только, если этот объект полностью сконструирован. Если же конструирование объекта не завершено, т. е. исключение зафиксировано в конструкторе, то автоматического освобождения памяти, уже выделенной под часть объекта, выполнено не будет, и, следовательно, произойдет «утечка» памяти. Например, для класса, описанного ниже, при возникновении исключения в конструкторе часть памяти останется недоступной.

```
class D
{
private:
    A * a;    B * b;

public:
    D() : a(new A), b(new B) { }
    ~D() throw()
    { delete a; delete b; }
};
```

Обеспечить корректное освобождение памяти для данного класса существенно сложнее.

В соответствии с этим естественным является желание упростить работу с динамическими объектами. Очевидное решение – вместо обычного указателя использовать объект-указатель, хранящий адрес и освобождающий память в своём деструкторе. Данная технология называется *Resource Acquisition Is Initialization* (RAII) – «Захват ресурса есть инициализация». Её смысл заключается в том, что захват ресурса совмещается с инициализацией объекта, а освобождение – с уничтожением объекта. Именно таким поведением обладает шаблон `auto_ptr` из стандартной библиотеки C++.

## 8.2 Шаблон `auto_ptr`

Шаблон `auto_ptr` включен в стандартную библиотеку `<memory>` языка C++ и определен в адресном пространстве `std`. Это единственный «умный» указатель, включенный в нынешний стандарт C++.

Шаблон предусматривает следующие методы:

- конструкторы простой и копирующий;
- деструктор;
- `get()` – возвращает хранимый адрес;
- `release()` – возвращает хранимый адрес и записывает вместо него `NULL` в объект-указатель;
- `reset()` – освобождает память, адрес которой хранится в указателе-объекте, и, если параметр – новый динамический объект указан, то записывает в него новый адрес, если параметр не указан, то адрес устанавливается в `NULL`.

Шаблон также переопределяет следующие операции:

- `operator=()` – операцию присваивания;
- `operator*()` – операцию доступа к адресуемому объекту по адресу;
- `operator->()` – операцию доступа к адресуемому объекту по адресу;
- `operator auto_ptr<Other>` – операцию преобразования указателя `auto_ptr` одного типа к указателю `auto_ptr` другого типа;
- `operator auto_ptr_ref<Other>` – операцию преобразования указателя `auto_ptr` одного типа к указателю `auto_ptr_ref` другого типа.

Чтобы построить класс, используя шаблон, необходимо задать тип динамического объекта, который будет храниться в объекте указателя.

**Пример 8.1.** Создание объекта-указателя, его использование и перезапись.

```

#include <memory>

#include <iostream>

#include <conio.h>

using namespace std;

class A
{
public:    int x;

    A(int X):x(X){}

    A(){}

    ~A(){cout<<"destructor"<<endl;}

};

void main()
{

    auto_ptr<A> temp1; // неинициализированный объект-указатель

    auto_ptr<A> temp(new A(1)); // инициализированный объект-указатель

    A &a= *temp; // ссылка на хранимый объект

    cout<<"A.x="<<a.x<<endl;

    A *ptr = temp.get(); // указатель на хранимый объект

    cout<<"ptr="<<ptr<<endl;

    temp.reset(); // освобождение указателя

    A *ptr1 = temp.get(); // указатель на хранимый объект

    cout<<"ptr1="<<ptr1<<endl;

    _getch();

}

```

Результат выполнения программы:

```
A.x=1
ptr=00396520
destructor
ptr1=00000000
```

Следует помнить, что шаблон определяет объект-указатель, реализующий семантику «владения», при которой всегда существует только один объект-указатель, хранящий адрес выделенного фрагмента динамической памяти. Для этого реализация шаблона обеспечивает «разрушающее» копирование, при котором копируемый адрес уничтожается после переписи в новый объект-указатель. Таким образом становится невозможным наличие двух и более объектов-указателей, адресующих один объект, и исключаются ошибки, связанные с повторным освобождением памяти объекта.

**Пример 8.2.** Невозможность создания копии объекта-указателя `auto_ptr`.

```
#include <conio.h>
#include <memory>
class A
{
public:    void f () {}
};
int main ()
{
    std::auto_ptr<A> temp1 (new A) ; // объявление и инициализация
                                   // объекта-указателя
    temp1->f () ; // вызов метода - выполняется нормально
    std::auto_ptr<A> temp2 (temp1) ; // объявление и инициализация
                                   // второго объекта-указателя
    temp1->f () ; // вызов метода – не возможен, так как адрес
```

```
        // разрушен при копировании
    _getch();
}
```

Невозможность создания копий адреса не позволяет использовать указатели, построенные по шаблону `auto_ptr`, для создания списковых структур, поскольку в процессе обработки список будет разрушаться. Выходом из этой ситуации является использование умного указателя с подсчётом ссылок – `shared_ptr`.

### 8.3 Шаблон `shared_ptr`

Шаблон `shared_ptr` позволяет создавать объекты-указатели, отличающиеся от объектов-указателей `auto_ptr` в основном методикой копирования. Хотя данный шаблон в настоящее время не включен в стандарт, он реализован в библиотеке `boost`, которая может быть интегрирована в Visual Studio 2008.

Шаблон определен в адресном пространстве `boost`. Он реализует разделяемое «владение» с подсчетом ссылок, при котором очередное копирование адреса приводит к увеличению специального счетчика на единицу. В свою очередь при уменьшении количества указателей-объектов, связанных с динамическим объектом, значение счетчика уменьшается. Все объекты-указатели при этом равноценно владеют объектом. Согласно используемой методике адресуемый объект освобождает занимаемую им память при отключении последнего адресующего его объекта-указателя.

Шаблон также позволяет при инициализации задавать процедуру удаления `deleter()`, что может быть полезно для классов, требующих выполнения нестандартных действий при удалении динамического объекта.

Кроме того, шаблон предусматривает оператор неявного преобразования объекта-указателя в `bool`, что позволяет выполнять проверку указателя по правилам C++, например:

```
boost::shared_ptr<A> p = ptr;
if (p) { // если указатель содержит действующий адрес
}
```

При работе с объектами-указателями, построенными по шаблону `shared_ptr` программисты часто допускают одну и ту же *ошибку*, создавая несколько разделяемых указателей на один объект без их копирования, например:

```
A ptr_obj = new A(); // динамический объект
{ // новый блок
    shared_ptr<A> ptr1(ptr_obj); // разделяемый указатель на объект
```

```

shared_ptr<A> ptr2(ptr_obj); // второй разделяемый
                                // указатель на тот же объект
...
} // в этом месте программа выдает ошибку обращения к
  // несуществующему объекту

```

Дело в том, что инициализация разделяемых указателей обычным указателем на объект приводит к созданию двух счетчиков ссылок, независящих друг от друга. Каждый из этих счетчиков обнуляется в конце блока, поскольку оба объекта-указателя уничтожаются. Это приводит к двум попыткам удаления одного динамического объекта. Первая срабатывает, вторая – приводит к ошибке (!).

Правильным решением было бы создавать оба объекта-указателя копированием исходного объекта-указателя, тогда будет использован копирующий конструктор шаблона `shared_ptr`, и, соответственно, счетчик будет один. Это приведет к единственному удалению объекта при обнулении счетчика.

**Пример 8.3.** Работа с объектом с использованием разделяемых объектов-указателей.

```

#include <iostream>
#include <stdlib.h>
#include <conio.h>
#include <boost/shared_ptr.hpp>
using namespace std;
using namespace boost;
class A
{
private:  int num;
public:
    A(int Num) : num(Num) {cout<< "Object created..."<<endl;}
    ~A() {cout<< "Object deleted..."<<endl;}
    void Print() {cout<<"num="<<num<<endl;}
};
void main()
{

```

```
shared_ptr<A> ptr_obj(new A(5)); // создание объекта и указателя
{ // новый блок
    shared_ptr<A> ptr1 = ptr_obj; // разделяемый указатель
                                // на объект

    ptr1->Print();

    shared_ptr<A> ptr2 = ptr_obj; // второй разделяемый
    // указатель на тот же объект
    ptr2->Print();
    ptr_obj.reset(); // освобождаем исходный указатель
} // здесь переменные ptr1 и ptr2 уничтожаются и объект корректно удаляется
getch();
}
```

## 8.4 Шаблон `weak_ptr`

Шаблон `weak_ptr` также пока не определен в стандарте, но реализован в библиотеке `boost` в адресном пространстве `boost`. Его применяют, если необходим объект-указатель, *не владеющий* динамическим объектом. Необходимость такого указателя, например, может быть вызвана необходимостью создания двух объекта, ссылающиеся друг на друга. При реализации такой взаимосвязи с использованием разделяемых указателей, удалить объекты не удастся.

Шаблон `weak_ptr` позволяет объявлять объекты-указатели, которые так же как и `shared_ptr` связаны со счетчиком ссылок, однако *не увеличивают его значения*. Если одну из связей между взаимоссылающимися объектами реализовать через `weak_ptr`, то объекты будут корректно удалены.

Поскольку объекты-указатели шаблона `weak_ptr` не увеличивают счетчика ссылок, возможно удаление объекта при существующем «слабом» указателе на него, как при использовании обычных указателей. Чтобы не допустить ошибки в этом случае, объект указателя `weak_ptr` не действует как обычный указатель, т. е. не предоставляет возможности оперирования над объектом. Вместо этого шаблон предусматривает метод `lock()`, который возвращает новый объект-указатель `shared_ptr` на динамический объект, и, соответственно, на время жизни этого объекта `shared_ptr` увеличивает счетчик ссылок объекта.

Проверить существование объекта можно и создавая объект-указатель `shared_ptr` на него. Для этого используют метод класса `weak_ptr` `expired()`, который возвращает `true`, если объект, для которого создавался указатель перестал существовать, и `false` – в противном случае.

**Пример 8.4.** Демонстрация различия `shared_ptr` и `weak_ptr`.

```
#include <iostream>
#include <stdlib.h>
#include <conio.h>
#include <boost/weak_ptr.hpp>
#include <boost/shared_ptr.hpp>
using namespace std;
```

```

using namespace boost;
int main()
{
    shared_ptr<int> s1( new int(5) );    // создаем объект ->
                                     // счетчик ссылок = 1
    shared_ptr<int> s2 = s1; // создаем сильный указатель ->
                               // счетчик ссылок = 2
    weak_ptr<int> w = s2;    // создаем слабый указатель ->
                               // счетчик ссылок не меняется

    s2.reset();    // освобождаем s2 ->
                               // счетчик ссылок = 1 и объект существует

    s2 = w.lock(); // связываем сильный указатель с объектом ->
                               // счетчик ссылок = 2

    s2.reset();    // освобождаем s2 -> счетчик ссылок = 1

    s1.reset();    // освобождаем s1 ->
                               // счетчик ссылок = 0 -> объект уничтожен

// проверка существования объекта (способ 1)
    if (w.expired()) cout<<"Object is not exists..."<<endl;
    else cout<<"Object exists..."<<endl;

// проверка существования объекта (способ 2)
    s2 = w.lock(); // пытаемся создать сильный указатель из слабого ->
                               // s2 равен 0, т.к. объект не существует

    if (s2.get() == 0) // получаем адрес объекта и сравниваем с нулем
        cout<<"Object is not exists..."<<endl;
    else cout<<"Object exists..."<<endl;

    _getch();
}

```

Таким образом шаблон `weak_ptr` гарантирует, что, если динамический объект уже уничтожен, то это можно безопасно проверить.

### Вопросы для самоконтроля

1. Какие проблемы существуют при работе с указателями в C++?

[Ответ.](#)

2. Какие возможности предоставляет шаблон `auto_ptr`? Какие проблемы при его использовании остаются нерешенными?

[Ответ.](#)

3. Какие возможности предоставляет шаблон `shared_ptr`?

[Ответ.](#)

4. Как взаимодействуют между собой объекты шаблонов `shared_ptr` и `weak_ptr`?

[Ответ.](#)

## ЛИТЕРАТУРА

1. Березин Б.И., Березин С.Б. Начальный курс С и С++. – М.: «Диалог-МИФИ», 1997.
2. Вайнер Р., Пинсон Л. С++ изнутри: Пер. с англ. – Киев: «ДиаСофт», 1993.
3. Лукас П. С++ под рукой: Пер. с англ. – Киев: «ДиаСофт», 1993.
4. Подбельский В.В. Язык Си++. – М.: Финансы и статистика, 1995.
5. Скляр В.А. Язык С++ и объектно-ориентированное программирование. – М.: Высшая школа., 1997.
6. Страуструп Б. Язык программирования С++: Пер. с англ. – СПб.; М.: «Невский проспект» – «Издательство БИНОМ», 1999.
7. Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К. Объектно-ориентированное программирование: Учебник для ВУЗов. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2007.