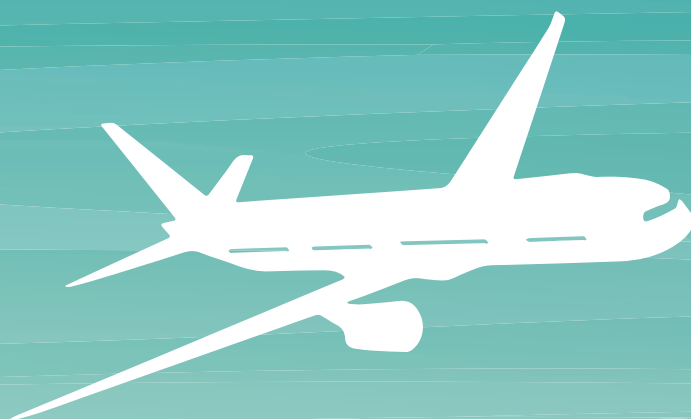




**МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ГРАЖДАНСКОЙ АВИАЦИИ**

Л.А. Надейкина

ПРОГРАММИРОВАНИЕ



**Москва
2017**

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ**

**«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ» (МГТУ ГА)**

Кафедра вычислительных машин, комплексов, систем и сетей

Л.А. Надейкина

ПРОГРАММИРОВАНИЕ

Утверждено Редакционно-
издательским советом МГТУ ГА в
качестве учебного пособия

Москва-2017

УДК 657.7:004(075.8)
ББК 6Ф7.3
Н17

Печатается по решению редакционно-издательского совета
Московского государственного технического университета ГА

Рецензенты: канд. техн. наук, доц. Л.А. Вайнейкис (МГТУ ГА);
канд. физ-мат. наук Н.Н. Остроухов (МАТИ)

Надейкина Л.А.
Н17 Программирование: учебное пособие. – М.: МГТУ ГА, 2017. – 84 с., лит.: 5 наим.,
28 рис., 9 табл.

ISBN 978-5-86311-994-6

В Учебном пособии рассматривается на базе языка C++ одна из основных парадигм современного программирования: объектно-ориентированное программирование (ООП), представленное таким понятием как класс и позволяющее разрабатывать библиотеки классов. Рассмотрены основные свойства классов, перегрузка операций, отношения классов, такие как включение и наследование, реализация виртуальных функций и абстрактных классов. Даны общие сведения об исключениях и методах их обработки.

Данное учебное пособие издается в соответствии с рабочей программой учебной дисциплины «Программирование» по Учебному плану для студентов I, II курса направления 09.03.01 очной формы обучения.

Рассмотрено и одобрено на заседании кафедры 19.05.2015 г. и методического совета 19.05.2015 г.

ББК 6Ф7.3
Св. тем. план 2017 г.
поз. 31

НАДЕЙКИНА Людмила Анатольевна

ПРОГРАММИРОВАНИЕ
Учебное пособие

Подписано в печать 01.02.2017 г.		
Печать офсетная	Формат 60х84/16	3,81 уч.-изд. л.
4,88 усл.печ.л.	Заказ № 1725/132	Тираж 30 экз.

Московский государственный технический университет ГА
125993 Москва, Кронштадтский бульвар, д. 20
ООО «ИПП «ИНСОФТ»
107140, г. Москва, 3-й Красносельский переулок д.21, стр. 1

© Московский государственный
технический университет ГА, 2017

Раздел 1 Структуры и объединения

1.1. Структура как совокупность данных

Из основных типов языка C++ пользователь может конструировать производные типы. Наиболее значимым из структурированных производных в языке C++ типов является **класс**. Прежде чем рассматривать множество понятий и разнообразие средств, относящихся к классам, остановимся сначала на частных случаях класса – на структурах и объединениях.

Рассмотрим структуры и объединения в том виде как они унаследованы от языка Си.

Структура - это объединенное в единое целое множество поименованных элементов (компонентов, полей) в общем случае разных типов.

Массивы также объединяют одним именем множество элементов, но все элементы должны быть одного типа.

Элементы структуры могут быть разных типов и все они должны иметь различные имена.

Перед определением **структуры** надо определить ее **структурный тип**, который и задает внутреннее строение структуры.

Введем простейший формат определения **структурного типа**:

```
struct имя_типа  
{список_компонентов};
```

Определение структурного типа, всегда заканчивается **;'**.

Список компонентов – это определения и описания, типизированных данных (полей), разделенных **;'**.

Например, как определить структуру, описывающую характеристики человека:

- имя человека (строка символов);
- возраст человека (целое число).

Определим тип такой структуры, используя спецификатор типа – **struct**:

```
struct men {  
char name [30];  
int age;  
};
```

Определим структурный тип для определения структуры, описывающей данные о студенте:

- ФИО студента (**char***);
- номер зачетки (**int**);
- отметки сессии (**int mark[]**);
- средний балл по результатам сессии (**float**).


```

struct student {
char * FIO;
int nz;
int maks [3];
float ball;
};

```

После определения типа можно определять конкретные структуры этого типа, массивы структур, указатели на структуру:

```

men men1, men2, mens [10], * m;
student st, stm [30], * pst;

```

men1, men2 - две структуры тип **men**;
mens[10] - массив из 10 структур типа **men**;
m - указатель на структуру типа **men**;
st - структура типа **student**;
stm[30] - массив из 30 структур типа **student**;
pst - указатель на структуру типа **student**.

При определении **структурного типа** не происходит выделения памяти.

Выделение памяти происходит при определении переменных данного структурного типа, то есть при определении **структур**.

Элементы структуры располагаются в памяти подряд. Количество памяти на каждый элемент выделяется в соответствии с типом элемента. Количество памяти, выделенное под структуру, определяется суммой байт участков, выделенных под ее элементы.

Если нет необходимости объявлять в разных частях программы структуры одного типа, можно не вводить именованный тип, а сразу в строке определения типа объявить все необходимые объекты, например,

```

struct {char fio[14]; int nz; float st; } A, B , Stud [10] , * Ptr ;

```

Для **обращения к элементу структуры** чаще всего используется уточненное имя:

имя_структуры. имя_элемента_структуры

Например, допустимо использовать следующие операторы:

```

cin >>A. fio;           // ввод значения – строки символов в поле fio структуры A
cout<<A.fio;           // значение элемента выводится в выходной поток
cin >> Stud [5]. fio; // вводится значение компонента fio шестого элемента
                        //массива структур Stud[5]
B.st = 259.5;          // полю st структуры B присваивается значение
cout << B.st ;          // выводится значение поля

```

Символьные данные можно представлять двумя способами:

1) в виде символьного массива:

```

char fio [14];

```

В этом случае имени массива нельзя присваивать строку, так как имя массива – указатель константа и присвоить ему новый адрес нельзя. То есть ***A.fio = "Петров";*** – не допустимо!

Чтобы поместить в это поле строку с фамилией надо воспользоваться операцией копирования:

cstrcpy (A.fio, "Петров");

Функция

char* cstrcpy (char*s1 , char* s2);

описана в модуле ***cstring.h***, копирует символы строки ***s2*** в строку ***s1*** и возвращает строку ***s1***.

Допустимо также введения значения компонента с клавиатуры так как память под массив ***fio*** выделена:

cin>> A.fio; - допустимо.

2) Фамилию в структуре можно представлять, используя указатель:

char* FIO ;

и тогда допустима операция присваивания указателю адреса строки:

A.FIO ="Григорьев";

При объявлении указателя в качестве компонента структуры для представления массива символов (например, – фамилии студента) выделяется память только на указатель, а на элементы массива память не выделяется.

Если в данном случае надо ввести значение строки, например, с клавиатуры или из файла, следует выделить участок памяти и присвоить указателю ***FIO*** адрес участка. Затем произвести ввод строки:

A.FIO=new char [14];

cin>> A.FIO;

При определении структуры можно провести её ***инициализацию*** – задание начальных значений её элементов.

В этом случае после определения структуры ставится знак '=' и следует список инициализации, заключенный в фигурные скобки, в котором через запятую перечисляются начальные значения элементов структуры.

Определим структуру типа ***student*** с инициализацией:

student one =

{"Петров", 4123, {4, 3, 2}, (one.maks [0] +one.maks [1] +one.maks [2])/3.0};

Уточненные имена формируются с именем конкретной структуры, а не с именем типа!

Определение типа дает шаблон (формат, внутреннее строение) для определения конкретных структур - объектов. Так:

student.FIO = "Денисов"; - ошибка!

st.FIO= "Денисов"; - допустимо.

Так как **структурный тип** обладает всеми правами типов, можно определять **указатели на структуры**:

имя_структурного_типа * имя_указателя_на_структуру
иницилирующее_выражение

Например,

student * ptr = &one; //указатель иницирован адресом структуры **one**

После того как указатель получил в качестве значения адрес структуры появляются возможности обращаться к элементам структуры, используя указатель:

1) **имя_указателя -> имя элемента структуры**

cout << ptr -> FIO ;

2) разыменование указателя и формирование уточненного имени:

(*имя_указателя). имя_элемента_структуры

cout << (*ptr). FIO ;

Можно также определять **ссылки на структуры**. Ссылка на структуру – это другое имя (синоним) уже имеющейся структуры.

Объявление ссылки:

имя_структурного_типа & имя_ссылки на структуру инициализатор;

Наличие инициатора в объявлении ссылки обязательно.

student & refo = one; // **refo** – синоним **one**

men & refm (men1); // **refm** – синоним **men1**

Обращения:

one.FIO эквивалентно **refo.FIO**

men1.age эквивалентно **refm.age**

Остановимся на вопросе **представления структуры в памяти ЭВМ**.

Все элементы хранятся в памяти подряд, как показано на рис. 1.1. Количество памяти, выделяемое структуре, определяется суммой байтов участков, выделенных под элементы.

struct { long L ; int i1, i2 ; char c [4]; } str;

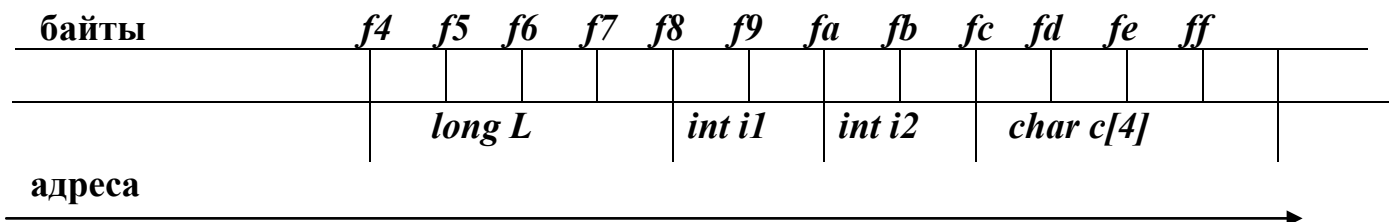


Рис.1.1. Размещение в памяти структуры **str**

Предполагается, что результат операций **sizeof**, например, следующий:

sizeof(str) = 12, если считать, что на **int** выделяется 2 байта

sizeof(str.L) = 0xa95f4

```
sizeof(str.i1) = 0xa95f8
sizeof(str.i2) = 0xa95fa
sizeof(str.c) = 0xa95fc
```

Компонентами (членами) структура могут быть объекты любых типов - скалярных типов, так и типов, определенных пользователем.

Единственное существенное ограничение - элементом структуры не может быть структура или массив структур того же типа.

В то же время элементом определяемой структуры может быть указатель на структуру определяемого типа:

```
struct def { def A, int B; }; //ошибка!
struct good { good* B; float C; }; //правильно!
```

Элементом структуры может быть другая структура, тип которой уже определен.

Если элементом структуры является указатель на структуру другого типа, в которой в свою очередь, используется указатель на первую структуру, то возможна следующая последовательность определений:

```
struct A; // неполное определение структурного типа.
struct B { struct A* pa; };
struct A { struct B* pb; };
```

Неполное определение структурного типа допустимо, так как определение указателя на структуру *A* не требует сведений о размере и строении структуры типа *A*. Последующее определение в той же программе структуры *A* обязательно.

Структуры и функции.

Функции могут возвращать структурированные данные как результат с помощью оператора **return**:

1) Функция может возвращать структуру как результат:

```
struct person { char*name; int age; }
person func1 (); //прототип функции
```

Результат вызова такой функции может, например, быть присвоен какой-либо структуре данного типа.

2) Функция может возвращать указатель на структуру:

```
person* func2(); //прототип функции
```

Результат функции может быть присвоен некоторому указателю на структуру. Адрес, который будет возвращать функция не должен быть адресом локальной структуры функции или адресом элемента локального массива структур.

3) Функция возвращает ссылку на структуру:

```
person& func3(); //прототип функции
```

В этом случае функция возвращает *l-value* (объект), вызов функции следует рассматривать как некоторую структуру, поля которой можно изменять, например, в операторе присваивания.

Через аппарат формальных параметров информация о структуре или массиве структур может передаваться в функцию

1) по значению, то есть передаваемая структура, копируется по адресу формального параметра:

```
void func4 (person str);    //прототип функции
```

2) по адресу - через указатель передается адрес внешней по отношению к функции структуры, значения полей которой операторы функции могут изменять:

```
void func5 ( person *pst);  //прототип функции
```

3) по ссылке - в этом случае функция непосредственно работает с передаваемой структурой, а не с ее копией:

```
void func6 ( person& rst);  //прототип функции.
```

Применение ссылки или указателя на объект в качестве параметра позволяет избежать дублирование объекта в памяти.

Выделение памяти под структуру или массив структур можно осуществлять динамически, во время исполнения программы.

Для выделения памяти используется операцию *new*. Для освобождения – операция *delete*. Определим структурный тип – *book* с данными о книге:

```
struct book {  
char*name;    // можно и так: char * name, * author;  
char *author;  
int year ;      // int year, pages;  
int pages;};  
//Далее надо объявить указатель на структуру:  
book * une;  
une = new ( book);    // выделение памяти на одну структуру  
delete une;          // освобождение памяти  
une = new book [9];   // выделение памяти на массив структур  
une[0].name = "Денисов";  
delete [ ] une ;      //освобождение памяти
```

1.2. Объединения разнотипных данных

Объединение - это структура, все элементы которой размещены на одном и том же участке памяти.

Размер участка памяти, выделяемого для объединения, определяется максимальным из размеров его элементов.

Все элементы объединения имеют нулевое смещение от его начала, или все элементы имеют один и тот же начальный адрес.

Основное назначение объединений – обеспечить возможность доступа к одному и тому же участку памяти с помощью объектов разных типов – элементов объединения. Это позволяет трактовать содержимое одного и того же участка памяти как значения данных разных типов и иметь доступ к отдельным участкам памяти объединения.

Определение объединения внешне похоже на определение структуры.
Определение объединяющего типа:

union имя_типа {список описаний элементов_объединения через ';'};

Однотипные члены можно описывать с одним идентификатором типа и через запятую.

Пример:

union un { long L; double D; char C[3]; } uni;

Здесь объявлен объединяющий тип ***un*** и переменная этого типа ***uni***.

На рис. 1.2 проиллюстрировано внутреннее представление объединения ***uni***.

a95a0

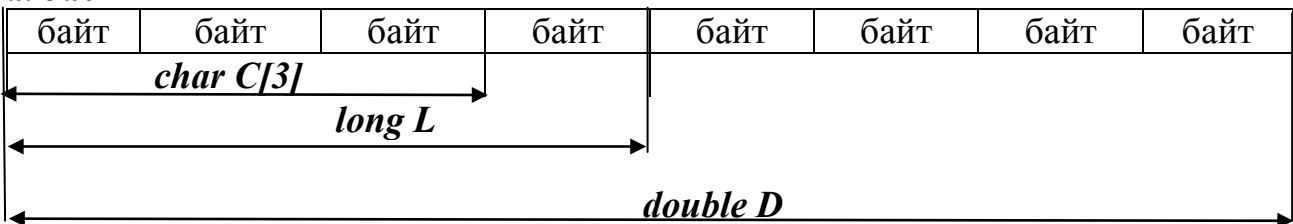


Рис. 1.2. Схема размещения в памяти объединения ***uni***

Все элементы объединения имеют один и тот же адрес и результатом оператора вывода:

cout << &uni.L << &uni.D << &uni.C;

будет трижды выведен один и тот же адрес, равный в данном случае ***a95a0***.

После определения типа объединения можно (по аналогии со структурой) определять конкретные объекты:

un u1, un [4] // объединение и массив объединений

un *pu // указатель на объединение

un& ru=u1 //ссылка на объединение

Форматы обращения к элементам объединения:

имя_объединения. имя_элемента

имя_указателя_на_объединение -> имя_элемента

(*имя_указателя_на_объединение). имя_элемента

Можно присваивать, вводить и выводить значения элементов.

Включение в объединение в качестве элемента символьный массив длиной с участок памяти, выделенный под объединение, позволяет обращаться к каждому байту памяти внутреннего представления объединения.

Например, определим объединение (рис. 1.3)

union { float fl; unsigned long l; char h[4]; } u;

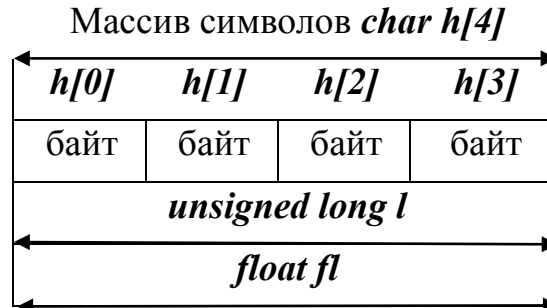


Рис. 1.3. Размещение в памяти объединения *u*

Занося в участок памяти, выделенный для объединения, вещественное число, например, так:

u.fl=2.718282f;

можно получить значение кода его внутреннего представления с помощью уточненного имени *u.l* и значения кодов, находящихся в отдельных байтах:

u.h[0], u.h[1], u.h[2], u.h[3].

Элементами объединения могут быть также массивы и структуры, возможны самые разнообразные их сочетания.

Разрешена инициализация объединений. Если при определении объединения надо провести его инициализация, то следует инициализировать только первый элемент объединения:

union comp { long L; int i [2] ; char ch[4] ; };

comp u1 = { 123456 };

comp u2 = { 'a', 'b', 'c', 'd' }; // ошибка

union { char ch [4]; long L; int i [2]; } = { 'a', 'b', 'c', 'd' }; //правильно

Можно создать анонимное объединение без явного указания имени типа и не вводить имя самого объединения

union { int im [3] ; char ch[6]; } = { 1, 2, 3 };

К элементам анонимного объединения обращаются как к отдельным объектам (переменным):

im [0]=10 ; // изменится также значение ch[0] и ch[1]

ch[5] = 'k' // изменится и значение im [3]

Можно определять массивы объединений и инициализировать их:

comp M [] = { 10L, 20L , 30L , 40L };

Количество элементов массива в данном случае определяется инициализацией. Инициализация проводится для первого элемента каждого объединения массива. Доступ к внутренним кодам возможен через элементы *M [ij]. ch[k]*.

1.3. Битовые поля структур и объединений

В качестве компонент объединений и структур (и классов) могут использоваться битовые поля.

Битовое поле – это целое или беззнаковое целое значение, занимающее в памяти фиксированное число битов, например, 1 бит.

Битовые поля – это не самостоятельные объекты, они могут быть только элементами структур, объединений или классов. Битовые поля не имеют адресов, нет указателей и ссылок на битовые поля, нет массивов битовых полей.

Битовые поля позволяют сформировать объекты с длиной внутреннего представления некратных байту, кратных биту.

Битовые поля используются для плотной упаковки данных.

Назначение битовых полей – обеспечить удобный доступ к отдельным битам данных.

Определение структуры с битовыми полями имеет такой формат:

```
struct имя_типа {  
тип_поля имя_поля: ширина_поля;  
тип_поля имя_поля: ширина_поля ;  
} имя_структуры;
```

тип_поля – один из базовых целых типов (*char, short, int, long*), их знаковых и беззнаковых вариантов;

ширина_поля – целое неотрицательное десятичное число, значение которого обычно не должно превышать длины слова конкретной ЭВМ. Таким образом, диапазон существенно зависит от реализации языка и системы. Однако стандарт C++ не запрещает вводить поля превышающие максимально допустимую величину битового поля. В этом случае лишние биты не учитываются в значении поля, а служат для разделения кодов в памяти.

Пример определения структуры:

```
struct {  
int a : 10;  
int b : 14;  
} xx, *px;
```

На рис. 1.4 показано размещение битовых полей структуры для компиляторов, работающих на IBM PC.

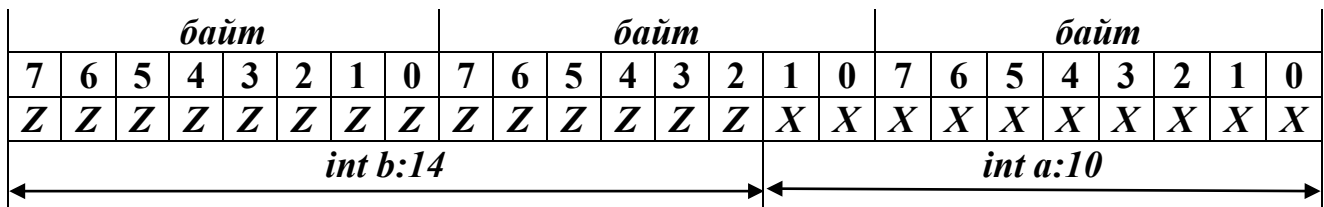


Рис. 1.4. Размещение битовых полей структуры

Определим функцию, которая упаковывает в один байт остатки от деления на 16 двух целых чисел и выводит их в двоичной форме.

Целочисленный остаток от деления должен быть меньше 16, то есть представляется не более чем четырьмя битами. Программа [4]:

```
#include <iostream>
using namespace std;
unsigned char F ( int a, int b ){
    union {
        unsigned char z ;
        struct {
            unsigned int x : 4;
            unsigned int y : 4;
        } st;
        struct {
            unsigned int a0 :1
            unsigned int a1 :1
            unsigned int a2 :1
            unsigned int a3 :1
            unsigned int a4 :1
            unsigned int a5 :1
            unsigned int a6 :1
            unsigned int a7 :1
        } st1;
    } un;
    //Заносим значения остатков от деления на 16 параметров функции в
    //объединение
    un. st. x =a % 16 ;
    un . st .y = b% 16 ;
    //Выводим биты внутреннего представления объединения – двоичный код
    //символа un.z
    cout << " \n" << un.st1. a0<< ... << un.st1.a7 ;
    //Возвращаем символ
    return un.z;
}
int main ( ) {
    int n , m ;
```

```

cin >> n ; cin>>m;
cout<< " \n" << F( m , n );
cout<<" \n" << int ( F(m,n));
return 0;
}

```

В результате будет выведен двоичный код символа, с новой строки будет выведен сам символ, затем с новой строки - опять двоичный код символа и с новой строки десятичный код символа.

Раздел 2

Класс как абстрактный тип

2.1. Класс – производный структурированный тип

Тип в языках программирования – понятие первичное. Тип объекта задает формат его внутреннего представления в памяти компьютера, множество значений, принимаемых объектом, и совокупность операций, выполняемых над этими значениями.

Реализация принципов абстракции данных при разработке программ предполагает создание новых типов данных с наибольшей полнотой отображающих особенности решаемой задачи [3]. В языке C++ программист имеет возможность вводить собственные типы данных и определять операции над ними с помощью классов.

Класс – это структурированный тип, состоящий из фиксированного набора возможно разнотипных данных и совокупности функций для обработки этих данных.

Таким образом класс в языке C++ рассматривается как естественное расширение понятия структуры.

Определение класса дается с помощью конструкции, называемой спецификацией класса:

```

ключ_класса имя_класса           // заголовок класса
{список_компонентов_класса};      // тело класса

```

Определение класса, как и структурного типа, всегда заканчивается ';',

ключ_класса – одно из слов **class**, **struct**, **union**,

имя_класса – правильный идентификатор,

список_компонентов_класса – (или членов класса - members) определения и описания типизированных данных (полей данных - data members) и принадлежащих классу функций (методов класса – member functions).

Функции – члены класса (member functions), называют также методами класса или компонентными функциями.

Данные класса (data members), называют полями данных, компонентными данными или элементами данных класса.

Определения разнотипных данных в списке отделяются ';', если данные однотипные, их идентификаторы можно перечислить через запятую, аналогично как в структурах.

Определим класс *book*:

```
struct book {  
char title [256];  
char author[40] ;  
float price;  
void show_title (void) {cout<<title<<"\n "};  
float get_price (void) {return (price); };  
};
```

Итак, класс — это тип, введенный программистом. Тип служит для определения объектов.

Создание объектов или экземпляров класса осуществляется также как создание структурных переменных, используя следующую конструкцию:

имя_класса имя_объекта;

Например,

```
book x1, x2, mas[ 5 ], * ptr =&x1, &ref = x2;
```

определены два объекта, массив объектов, указатель на объект и ссылка на объект класса *book*.

Как только объект определен, в него входят данные, соответствующие полям данных класса.

Методы класса предназначены для обработки данных конкретных объектов, но в отличие от полей данных не тиражируются при создании конкретных объектов.

Как только объект определен появляется возможность обращаться к его компонентам

1) с помощью квалифицированного имени:

```
имя_объекта. имя_класса :: имя_компонента
```

```
имя_объекта. имя_класса :: имя_компонентной_функции(аргументы)
```

2) с помощью уточненного имени:

```
имя_объекта. имя_компонента
```

```
имя_объекта. имя_компонентной_функции(аргументы)
```

Можно присваивать, копировать, вводить, выводить значения полей данных объектов.

```
cin>>x1.title ;  
x1.price = 750.5;  
cstrcpy ( x1.title, "Язык C++ ");  
cin >> mas[3].book::price ;    cout<<mas[3].price;
```

Вызов компонентной функции, используя квалифицированное имя:

x1. book::show_title ();

Такой вызов приведет к выводу – **Язык C++** .

Вызов функции с помощью уточненного имени:

cout<<mas[3].get_price();

Компонентная функция объекта обрабатывает данные того же объекта!

Другой способ доступа к данным предусматривает использование указателя на объект и операции косвенного выбора ->:

указатель_на_объект_класса -> имя_класса::имя_поля_данных
либо

указатель_на_объект_класса -> имя_поля_данных

Форматы обращения к методам:

указатель_на_объект_класса -> имя_класса::имя_метода (аргументы)
либо

указатель_на_объект_класса -> имя_метода (аргументы).

Определив выше указатель *ptr*, адресующий объект *x1*, можно следующим образом обращаться к полям данных и вызывать, например, метод *show_title ()* для объекта *x1*:

ptr->price = 650.7;

ptr-> show_title (); //выведет Язык C++

*cstrcpy(ptr ->title , "Turbo Pascal"); // поле title объекта x1 получит новое
// значение*

ptr-> show_title (); //выведет Turbo Pascal

При объявлении в классе компонентных функций (методов) имеется выбор: можно определить функцию вне класса или непосредственно в теле класса.

1) При объявлении функции вне класса, внутри тела класса помещается только прототип или описание функции - заголовок функции с указанием типов формальных параметров, оканчивающийся знаком ';':

Все создаваемые экземпляры (объекты) класса будут использовать единственный код функции, а компилятор будет генерировать код вызова функции при каждом обращении к ней.

Например, класс имеет три метода, определенных вне класса. Пусть в программе создаются 100 экземпляров (объектов) класса, тогда в программе присутствуют 100 экземпляров данных, но коды только трех функций. Данные тиражируются при создании объектов. Компонентные функции не тиражируются.

2) При полном объявлении функции внутри класса, функция по умолчанию считается *подставляемой*, то есть при каждом вызове такой функции код функции "встраивается" в точку вызова. Ранее мы рассматривали, так называемые, *inline-функции*. Вместо команд передачи управления

единственному экземпляру функции компилятор в каждое место вызова функции помещает код команд операторов тела функции. Это может увеличить скорость выполнения программы, но увеличивает также и размер кода программы. Если метод большой по объему, не следует настраивать компилятор на генерацию встраиваемого кода, а следует объявить этот метод вне класса.

Ограничения для подставляемых функций (включая и компонентные):

- 1) встраиваемая функция слишком велика;
- 2) встраиваемая функция рекурсивная;
- 3) обращение к встраиваемой функции размещено до ее определения;
- 4) встраиваемая функция используется более одного раза в выражении;
- 5) встраиваемая функция содержит операторы *for*, *while*, *do*, *switch*, *goto*.

2.2. Статический компонент класса

Каждый объект одного и того же класса имеет собственную копию полей данных. То есть данные класса тиражируются при каждом определении объекта класса. Обычные данные отличаются друг от друга тем какому объекту они принадлежат.

Иногда возникает необходимость объявить данные общие для всех создаваемых объектов данного класса.

Например, в следующих случаях:

- если нужен счетчик создаваемых объектов данного класса,
- или для объектов, связанных в список, для работы со всеми объектами списка (например, в функции просмотра списка) нужен указатель на начало списка – общий для всех связанных объектов,
- может понадобиться указатель на последний элемент списка.

Такое поле данных должно быть определено в классе как статическое, то есть иметь атрибут *static*.

Статический компонент класса не дублируется при создании новых объектов класса. Каждый статический компонент класса существует в единственном экземпляре и память на него выделяется при его инициализации и сохраняется до окончания действия программы.

Инициализация статического компонента размещается в глобальной области после определения класса следующим образом:

тип_компонента имя_класса :: имя_компонента инициализатор.

С этого момента статический компонент класса становится доступным (если он открыт) даже до объявления объектов данного класса по квалифицированному имени:

имя_класса :: имя_компонента.

После объявления объектов к статическому полю данных можно также обращаться как к обычному полю данных, то есть с использованием операций выбора (. и ->).

При этом любые изменения (в любом объекте) статического элемента становятся общими для всех объектов данного класса.

Рассмотрим использования статических полей данных [3].

```
#include <iostream>
using namespace std;
struct goods {                // класс "товары"
    char name[40];            // наименование товара
    double price;              // закупочная цена
    static int percent;        // торговая наценка в % – статический компонент
    //методы для объектов класса:
    void vvod ( )              // ввод данных о товаре
    {cin >> name; cin>> price;}
    void vivod ( ) {          // вывод данных о товаре
    cout <<"\n" << name ;
    cout <<" - розничная цена =";
    cout << price*(1.0+ goods::percent*0.01)<<endl;
    };
    int goods::percent =25;    // инициализация статического компонента

    int main (void) {
    // определяется массив из 5 объектов, для двух объектов проводится
    //инициализация
    goods top [5]={ {"Костюм ", 10000}, { "Шляпа", 1100 } };
    // вводятся данные в оставшиеся объекты массива с помощью метода vvod( )
    for (int i =2 ; i <5 ; i++ ) top [i].vvod();
    cout<<"\nСписок товаров при наценке " <<
    top[0]. percent<< "%"<<endl;
    // выводятся данные о товарах
    for( i =0 ; i<5 ; i++ ) top[i].vivod( );
    // присваивается другое значение статическому компоненту
    goods::percent =30;
    //объявляем указатель на объект класса и инициализируем его
    //адресом первого элемента массива top
    goods* ptr = top;
    cout<<"\nСписок товаров при наценке " <<
    top[0]. percent<< "%"<<endl;
    for(i=0 ; i <5 ; i++ ) ptr++ -> vivod ();
    return 0;
    }
```

Результатом выполнения программы будут два списка тех же товаров, но с разной наценкой.

При моделировании связанного списка можно так определить класс:

```
class list { ...
```

```
static list * begin;    //указатель на вершину связанного списка  
...};  
list* list :: begin = NULL;    //инициализация статического элемента
```

2.3. Доступ к членам класса

Определены три уровня доступа к полям данных и методам класса, для каждого из которых предусмотрен свой спецификатор.

- 1) Данные или метода класса, описанные после ключевого слова ***public***, являются открытыми (общедоступными). Это означает, что к ним можно обращаться, из функций данного класса без формирования уточненного имени, из функций других классов, из внешних функций программы с использованием уточненных имен. Обычно в классах открывают функции общего назначения, чтобы с их помощью можно было бы посылать сообщения объектам.
- 2) Данные и методы, описанные после ключевого слова ***private***, являются закрытыми (частными, приватными). Это означает, что к ним можно обращаться только из функций данного класса (открытых или закрытых), из программы к ним обратиться нельзя. Обычно поля данных класса закрывают, выполняется, так называемый, принцип ***инкапсуляции*** данных внутри объекта. Иногда закрывают и функции внутреннего назначения.
- 3) Данные и методы, описанные после ключевого слова ***protected***, являются защищенными. Такие члены класса доступны только методам данного класса и методам классов, ***производных*** от данного. Защищенные отличаются от закрытых членов класса, если имеет место ***наследование***.

Закрытые члены класса недоступны и в производных классах!

Ключевые слова ***public***, ***private***, ***protected*** могут встречаться в описании класса в любом порядке и количестве.

В языке C++ действуют описанные ниже правила.

Если класс определен с помощью ключевого слова ***struct***, то все компоненты класса являются общедоступными без использования спецификаторов доступа. Они глобальны, их статус – ***public***, при этом не выполняется принцип ***инкапсуляции*** данных внутри объекта; ситуацию можно исправить, поставив перед определением данных в классе спецификатор ***private***.

В классах, определенных с помощью служебного слова ***union*** все элементы класса также общедоступны, но в каждый момент времени исполнения программы объект-объединение проявляет себя только как одно из полей данных, присутствующих в классе. Тем самым один и тот же код участка памяти, выделенной объекту, может рассматриваться как значения разных типов.

Все компоненты класса, определение которого начинается со служебного слова ***class***, недоступны для внешних обращений - статус ***private***. Такой класс

редко может оказаться полезным, поэтому обычно перед некоторыми методами ставят спецификатор **public**.

Изменить статус доступа полей данных и методов при определении класса позволяют спецификаторы доступа: **public** - общедоступный, **private** – собственный, **protected** – защищенный, за которыми помещается двоеточие.

Если в определении появился спецификатор доступа, то до следующего спецификатора все компоненты класса будут иметь указанный статус.

Как правило, данные объявляют со статусом **private**, а компонентные функции (методы), с помощью которых можно изменять эти данные, объявляют со статусом **public**.

```
class A {
int X, Y;           // закрытые по умолчанию поля данных
void f ();          // закрытая по умолчанию функция - метод
protected:
void init ();       // защищенный метод, доступная в производных классах
public:
void setX (int);    // открытая компонентная функция
void setY (int);    // то же
};
```

Если **статический компонент** класса имеет статус **private** или **protected**, то к нему извне можно обращаться только через компонентные функции, при наличии уже определенного объекта.

Возможность обращения к **статическому элементу** без имени конкретного объекта (и до определения объектов) обеспечивает открытый **статический метод класса**, определенный в классе со спецификатором **static**.

Такой метод сохраняет все свойства обычных (нестатических) методов класса, но дополнительно такую функцию можно вызвать, используя квалифицированное имя:

имя_класса :: имя_статической функции;

Рассмотрим использования статических данных и методов на примере класса **goods1**.

```
#include <iostream>
using namespace std;
class goods1 {           // класс "товары1"
//закрытые по умолчанию поля данных
char name[40] ;         // наименование товара
double price;           // закупочная цена
static int percent;      // торговая наценка
// открытые методы класса:
public:
void vvod ( )           // ввод данных о товаре
{ cin >> name; cin >> price; }
static void SetPer ( int newper) // открытая статическая функция для
```



```

{ preset = newper;}           //изменения статического элемента
void vivod ( ) {               // вывод данных о товаре
cout <<"\n" << name ;
cout <<" - розничная цена =";
cout << price*(1.0+ goods1::percent*0.01)<<endl;
};
//внешнее определение и инициализация статического компонента
int goods1::percent =25 ;
/* поле percent – закрытый компонент класса, обращение к нему возможно
только с использованием объектов класса и открытых методов или используя
доступную статическую компонентную функцию */
int main (void) {
goods1 top [5];
//вводим данные в объекты массива, используя компонентную функцию vvod()
for (int i =0 ; i <5 ; i++ ) top [i].vvod( );

for ( i =0 ; i <5 ; i++ ) top[i].vivod ;      // выводим данные о товарах
// изменяем значение статического компонента
goods1::SetPer(30);
good1* ptr = top;
//вновь выводим элементы массива после изменения торговой наценки.
for (i=0 ; i <5 ; i++ ) ptr++ -> vivod( );
return 0;
}

```

2.4 Дружба классов

Не редко возникает необходимость использования каких-либо внешних по отношению к классу функций общего назначения для обработки закрытых данных класса, при этом эти функции не могут быть методами класса.

Для этих целей в языке C++ предусмотрено объявление в классе дружественных функций.

Дружественной функцией класса называют функцию, которая не является компонентной функцией класса, но имеет доступ к защищенным и собственным компонентам класса.

Дружественная функция обладает следующими свойствами:

- 1) должна быть описана в теле класса со спецификатором **friend**;
 - 2) не является компонентной функцией (методом) класса, в котором она определена как дружественная;
 - 3) может быть глобальной:
- ```

class A { friend void f (...); ... } ;
void f (...) {...};

```
- 4) может быть компонентной функцией (методом) другого ранее определенного класса; и тогда при описании в классе надо использовать полное имя функции, включающее имя класса, которому она принадлежит:

```
class A {... void f1 (...); ...};
class B {... friend void A :: f1(...); ...};
```

5) может быть дружественной по отношению к нескольким классам

```
class A; // неполное определение класса A
class B {... friend void f2 (A, B); ...}; //полное определение класса B
class A {... friend void f2 (A, B) ; ...}; //полное определение класса A
void f2 (A tip1, B tip2) {тело функции} // определение функции f2
```

Рассмотрим методику работы с дружественными функциями пользовательского класса комплексных чисел.

```
#include <iostream>
#include <cmath>
using namespace std;
class MyComplex { //вариант класса "комплексное число"
double re, im ; //вещественная и мнимая части числа
public:
void define (double, double); // присваивает полям данных значения аргументов
double real (); //возвращает действительную часть числа
double image(); //возвращает мнимую часть
double mod (); //возвращает модуль числа
//функция для получения суммы двух комплексных чисел
MyComplex sum (MyComplex); //прототип метода sum
};
//внешние определения методов класса:
void MyComplex :: define (double _re, double _im)
{ re = _re; im = _im;}
double MyComplex :: real ()
{ return re; }
double MyComplex :: image ()
{ return im; }
double MyComplex :: mod ()
{ return sqrt (re*re + im* im); }
MyComplex MyComplex :: sum (MyComplex z)
{ z.re += re ; z.im += im;
return z ;
}
int main () {
MyComplex c1, c2, c3; //определяем три экземпляра класса
//заполняем поля данных объектов
c1.define (3, 4); c2.define (5, 6); c3.define (0, 0);
double m = c1.mod();
cout<< '\n'<< m;
```

```

c3 = c1.sum(c2);
cout<<"\n"<< "sum: " <<c3.real() ;
(c3.image()<0)? cout<< c3.image ()<< 'i' : cout<< '+' << c3.image() << 'i' ;
return 0; }

```

Рассмотрим функцию суммирования – метод класса. Функция вызывается для объекта *c1*, а в качестве аргумента она принимает объект *c2*:

```
c3 = c1.sum (c2);
```

Объект *c2* передается по значению, то есть создается его локальная копия в стеке с именем *z*. Функция формирует объект *z*, добавляя к нему данные объекта *c1*:

```

z.re += re;
z.im += im;

```

Этот временный объект *z* существует, пока выполняется функция, и по завершении работы функции он присваивается объекту *c3*.

При использовании компонентной функции форма вызова *sum* не симметрична относительно *c1* и *c2*. Естественно было бы вызывать функцию следующим образом:

```
c3= sum (c1, c2);
```

Однако при использовании функций-методов это невозможно, нельзя вызывать функцию без указания объекта, для которого она вызывается.

Симметричная форма реализована в следующей программе, в которой функция *sum* определена, как внешняя дружественная функция.

```

#include <iostream>
using namespace std;
class Complex { //другой вариант класса "комплексное число"
double re, im ;
public:
void define (float, float); //метод класса
friend Complex sum (Complex, Complex); //дружественная функция
}; // возвращает сумму двух комплексных чисел

//определение метода класса
void Complex :: define (double _re, double _im)
{re = _re ; im = _im;}
//определение внешней функции, дружественной классу
Complex sum (Complex z, Complex y)
{z.re += y.re; z.im += y.im;
return z;
}
int main () {
Complex c1, c2, c3;
c1.define (3, 4); c2.define (5, 6); c3.define (0, 0);
c3 = sum (c1, c2);
}

```

```
cout<<"\n"<< " sum: " << real (c3) ;
(image (c3)<0)? cout<< image (c3) << 'i' : cout<< '+' << image (c3) << 'i' ;
return 0;
}
```

Теперь сложение двух объектов выглядит симметрично, но для сложения полей данных (объектов) используется функция, а не операция '+', что само по себе нелепо.

### Дружественные классы.

Если все компонентные функции (методы) одного класса являются дружественными для другого класса, то класс является дружественным этому другому классу.

Объявление дружественного класса:

```
class X {...};
class Y {... friend class X; ...};
```

Все приватные и защищенные члены класса *Y* могут обрабатываться функциями класса *X*, то есть являются доступными в дружественном классе.

Дружественный класс может быть определен позже, нежели описан как дружественный:

```
class X1 {... friend class X2 ;...};
class X2 {...};
```

## 2.5 Конструкторы и деструкторы

Класс может содержать любое количество методов самого разнообразного назначения, но два типа методов занимают особое положение. Эти методы называются **конструктором и деструктором**.

### Конструкторы.

Для многих объектов естественно требовать, чтобы они были инициализированы (то есть получали начальное значение) при их определении.

Для упрощения процесса инициализации объектов предусмотрен специальный метод, называемый **конструктором**.

**Конструктор – это компонентная функция (метод класса), вызываемая автоматически при создании объекта класса и выполняющая необходимые инициализирующие действия.**

Формат определения конструктора в теле класса:

```
имя_класса (список_параметров) инициализатор_конструктора
{операторы_тела_конструктора}
```

Рассмотрим особенности конструкторов.

- 1) Имя конструктора должно совпадать с именем класса.
- 2) Конструктор не может возвращать результат, даже тип **void** не допустим.
- 3) Конструктор вызывается при определении объекта, или при размещении объекта в памяти с помощью операции **new**.

4) **Основное назначение конструктора** – превращать участки памяти в объекты класса, то есть проводить инициализацию полей данных объектов. Кроме того, предусмотрены также такие сопутствующие действия как - открытие файлов, вывод сообщений, инициализация объектов вспомогательных классов и тому подобное.

5) Существуют два способа инициализации полей данных создаваемых объектов.

Во – первых, можно в теле конструктора присваивать значения полям данных объектов. Эти значения обычно предоставляют параметры конструктора. Инициализатор, помещенный между списком параметров и телом конструктора, при этом опускается. Например, в классе *Men* можно так ввести конструктор:

```
class Men {
char * name; int age;
public:
Men (char*, int); //прототип конструктора класса
};
Men :: Men (char*n, int a) //внешнее определение конструктора
{name = n; age=0;}
```

Второй способ предусматривает применение инициализатора ("*ctor-initializer*"). Инициализатор представляет собой список инициализаторов полей данных объекта, расположенный после списка параметров и отделенный от него ':' (двоеточием). Каждый инициализатор относится к конкретному (не статическому) полю данных и имеет вид:

**имя\_поля\_данных (список выражений).**

Инициализаторы в списке отделяются друг от друга запятыми. Для полей данных простых типов в списке выражений используется одно выражение, в которое могут входить константы, параметры конструктора и имена уже инициализированных полей данных.

Рассмотрим пример, где используется конструктор с инициализатором.

```
class A {
int ii; float ee; char cc;
A (int i, float e, char c): ii (7), ee (ii + i * e), cc(c) { }
};
```

При создании нового объекта с привлечением конструктора с инициализатором последовательность действий такова:

- Выделяется память для объекта.
- Инициализируются не статические поля данных объекта с помощью инициализатора (в списке инициализатора могут присутствовать не все поля).
- Выполняются операторы тела конструктора, причем эти операторы могут изменить, полученные значения полей за счет инициализации, и значения статических полей класса.

6) В определении класса могут присутствовать несколько конструкторов. Конструкторы могут иметь различное число параметров, необходимое для инициализации создаваемого объекта. Параметры могут иметь умалчиваемые значения. При этом допускается только один конструктор с умалчиваемыми значениями или конструктор без параметров.

7) Перечислим названия (виды) конструкторов: **конструктор общего вида**, **конструктор без параметров** (единственный, но необязательный), **конструктор умолчания**, **конструктор копирования** (единственный).

8) Если в классе программист не определил ни одного конструктора, то по умолчанию формируются конструктор без параметров и конструктор копирования с прототипами соответственно (их автоматически добавляет компилятор):

***T::T();***

***T::T(const T&);***

при создании экземпляров класса компилятор автоматически выделяет под них память, хотя в этом случае поля данных не инициализируются.

9) Параметром конструктора не может быть его собственный объект, но может быть ссылка на него.

10) Нельзя получить адрес конструктора.

11) Конструктор нельзя явно вызывать как обычный метод класса. Конструктор неявно вызывается при создании именованного объекта класса или безымянного, например, в следующих конструкциях:

***имя\_класса имя\_объекта (аргументы конструктора);***

***имя\_класса (аргументы конструктора);***

Последний случай, например, используется при создании объекта в динамической памяти с использованием операции *new*:

***имя\_класса\*имя\_указателя = new имя\_класса (аргументы конструктора);***

### **Конструктор с параметрами или конструктор общего вида.**

Если в классе определен конструктор общего вида, то с его помощью можно создавать объекты с нужными значениями полей данных.

Форматы определений:

***имя\_класса имя\_объекта (аргументы\_конструктора);***

***указатель\_на\_объект = new имя\_класса (аргументы\_конструктора);***

***имя\_класса имя\_массива [размер\_массива] =***

***{имя\_класса (аргументы\_конструктора\_для\_0-го\_экземпляра), ...,***

***имя\_класса (аргументы\_конструктора\_для\_последнего\_экземпляра)***

***};***

Примеры создания объектов для класса *Men*:

***Men one ("Иван", 25);*** // *one.name == "Иван", one.age==25*

***Men\* ptr = new Men ("Олег", 55);*** // *ptr->name == "Олег", ptr->age==55*

### Конструктор с аргументами, задаваемыми по умолчанию.

Этот конструктор является частным случаем конструктора общего вида. При его определении параметрам задаются умалчиваемые значения. Пример для класса **Men** конструктора с умалчиваемыми значениями параметров:

*Men (char\*n="", int a=0);*

Этот конструктор позволяет при его вызове с параметрами инициализировать данные создаваемого объекта указанными значениями параметров. Также возможен вызов конструктора и без параметров, при этом поля данных будут инициализироваться значениями по умолчанию.

```
Men m1; // m1.name == "", m1.age==0
Men m2 ("Иванов", 45) // m2.name == "Иванов", m2.age == 45
Men m3 ("Петров") // m3.name == "Петров", m3.age= =0
Men m4 (18) // ошибка!
```

Последний случай (ошибка) указывает, что нельзя задавать параметр, перескакивая через умалчиваемое значение, этот аргумент (18) компилятором будет трактоваться как значение для параметра *n*, и естественно будет сообщение об ошибке несоответствия типов, так как *n* это указатель на *char*.

Пример:

```
struct mag {
int a, b, c;
// конструктор с умалчиваемыми значениями параметров
mag (int aa =1, int bb =2, int cc =3) {a = aa; b =bb; c = cc;}
void shownumber (void) { cout << a << b << c;}
};
mag one; // объект инициализируется значениями 1, 2, 3
mag one1 (10); // объект инициализируется значениями 10, 2, 3
mag one2 (10, 20); // объект инициализируется значениями 10, 20, 3
mag one3 (10, 20, 30); // объект инициализируется значениями 10, 20, 30
```

### Конструктор по умолчанию.

Это разновидность конструктора без параметров (присутствует в классе в единственном экземпляре).

Присутствует, когда надо единообразно инициализировать поля данных, или инициализирующие действия вообще не связаны с данными.

Этот конструктор не имеет параметров.

```
class A {
int x, y;
public:
A ();
};
A :: A () {x = 0 ; y = 0;} //единообразная инициализация полей
//A :: A () {} //инициализация данных в конструкторе не проводится
```

Конструктор умолчания вызывается, когда для создания объектов используются следующие определения:

*имя\_класса* *имя\_объекта*;

*имя\_класса* *имя\_массива\_объектов* [*размер*];

*указатель\_на\_объекты\_класса* = *new* *имя\_класса*;

*указатель\_на\_объекты\_класса* = *new* *имя\_класса*[*размер*];

При наличии конструктора по умолчанию, предложение:

*A one*;

создает объект со значениями полей данных  $x=0$  и  $y=0$ .

Если в конструкторе умолчания не проводилась инициализация данных, такой конструктор предоставляет возможность создавать неинициализированные объекты даже при наличии в определении класса еще одного конструктора с параметрами.

Вызов конструктор по умолчанию схож по форме с вызовом конструктора с умалчиваемыми значениями и при написании конструкции:

*имя\_класса* *имя\_объекта*;

компилятор не знает какой конструктор вызывать при создании объекта и выдает сообщение об ошибке.

Существует правило:

***При наличии в классе конструктора с параметрами, задаваемыми по умолчанию, объявлять в классе еще и конструктор по умолчанию нельзя!***

Вот если в определении класса вообще нет конструктора, компилятор автоматически предоставляет конструктор по умолчанию следующего вида:

*имя\_класса* () { }

который участвует в создании неинициализированных объектов.

### **Конструктор копирования.**

Конструктор копирования вызывается, когда:

- 1) надо создать объект, полностью совпадающий с уже созданным;
- 2) надо передать по значению в некоторую функцию экземпляр класса, при этом в стеке создается локальная копия объекта, с которым и работает функция;
- 3) надо создать объект полностью идентичный объекту, который возвращает некоторая функция посредством оператора *return*.

В большинстве случаев компилятор предоставляет нам конструктор копирования по умолчанию, который обеспечивает правильное копирование.

Рассмотрим копирование с использованием конструктора копирования по умолчанию:

*class T* {

*int x, y* ;

*public:*

*T ( int tx, int ty ) { x = tx; y = ty; }*

*int GetX () { return x; }*

*int GetY () { return y; }*



```

friend T sum (T, T);
};
void Print (T obj) {
cout<<"\n"<<" x = "<<obj.GetX() << " y = "<<obj.GetY();
}
T sum (T obj1, T obj2){
obj1. x += obj2. x;
obj1. y += obj2. y;
return obj1;
}
#include <iostream>
using namespace std;
int main () {
T e1 (1, 10); //создается объект, вызывается конструктор с параметрами
Print (e1); // создание локальной копии объекта e1
T e2 = e1; //создание объекта e2 копированием объекта e1
Print (e2); // создание локальной копии объекта e2
T e3 = sum (e1, e2); // создание локальных копий объектов e1и e2
//и создание объекта e3 копированием объекта, возвращаемого функцией sum()
Print (e3); // создание локальной копии объекта e3
return 0; }

```

Результат программы:

```

x= 1 y= 10
x= 1 y= 10
x= 2 y= 20

```

Определение конструктора копирования.

При поиске формы определения конструктора копирования следует учесть следующие соображения.

Во-первых, объект, копия которого создается, не должен копироваться в конструктор копирования, а должен передаваться в него по ссылке. Иначе в конструкторе копирования вызывался бы сам конструктор копирования и возникала бы бесконечная рекурсия.

И во-вторых – необходимо ввести запрет на модификацию копируемого объекта, и для этого следует перед объектом - параметром поставить ключевое слово **const**.

Учитывая вышесказанное, формат определения конструктора копирования для описанного выше класса должен быть следующим:

```

T (const T & obj) { x = obj.x; y = obj.y;}

```

Если пользователь определил в классе конструктор копирования, то в случаях создания объектов – копий будет вызываться конструктор копирования, определенный в классе. Однако свой конструктор копирования нужен не всегда. Копирование по умолчанию (когда конструктор копирования

предоставляет компилятор) осуществляет как правило правильное копирование полей данных и без явного определения в классе конструктора копирования. Обязательное его определение должно быть в тех случаях, когда класс содержит поля, являющиеся указателями на динамические участки памяти, при создании так называемых *ресурсоемких объектов*.

Прежде чем рассмотрим копирование ресурсоемких объектов, рассмотрим еще один очень важный метод классов, называемый деструктором.

### **Деструкторы.**

В языке C++ управление размещением объектов в памяти и их удаление из памяти, когда потребуется, полностью во власти программиста. Объекты с нужными свойствами создаются с помощью конструкторов. Для выполнения действий, которые сопровождают удаление объектов, в каждом классе явно или неявно определяется специальный метод, называемый *деструктором*.

*Деструктор - это компонентная функция класса (метод класса), которая автоматически выполняется, когда экземпляр класса уничтожается.*

Деструктор вызывается либо при выходе объекта за пределы области действия объекта, либо при освобождении динамической памяти операцией *delete*, выделенной объекту при его создании с помощью операции *new*.

Назначение деструктора – выполнение действий, сопровождающих удаление объекта. Наиболее важное это освобождение ресурсов, включенных в объект при его создании или при выполнении действий над объектом. Такими ресурсами могут быть участки памяти, динамически выделяемые для полей данных объекта, файлы, открытые при создании объекта и связанные с ним, и другие ресурсы. Деструкторы могут быть нужны и при уничтожении объектов, не захвативших никаких ресурсов, например, для вывода завершающих фраз.

Класс может иметь несколько конструкторов, но *деструктор может быть только один*.

Формат определения деструктора в теле класса:

*~имя\_класса () {операторы\_тела\_деструктора};*

Рассмотрим свойства деструкторов:

- 1) Между тильдой и именем класса нет пробелов.
- 2) У деструктора нет типа результата даже типа *void* и нет параметров даже типа *void*.
- 3) Деструктор выполняется неявно, автоматически, как только объект уничтожается. Его, как правило, никогда не вызывают, но можно вызывать и явно, если он определен в классе. Формат вызова конструктора:

*имя\_объекта. ~имя\_класса ();*

*имя\_указателя\_на\_объект\_класса -> ~имя\_класса ();*

при этом объект будет продолжать существовать, только выполнятся те действия, которые записаны в теле деструктора.

Определим класс *Men1*, вариант класса *Men*, в котором присутствуют и конструктор, и деструктор.

В *конструкторе* кроме инициализирующих действий имеется еще вывод контрольной строки. *Деструктор* пусть также содержит вывод другой контрольной строки.

```
class Men1 {
char* name; int age;
public:
Men1 (char * n, int a) //конструктор
{ name = n; age = a; cout<<name << " - begin " <<endl; }
void SetN (char*n) {name =n;} // метод для изменения поля name
void SetA (int a) { age = a;} // метод для изменения поля age
char* GetN () { return name;} //возвращает значение name
int GetA { return age;} // возвращает значение age
~Men1 () { cout<< name<< " - end"<<endl; } // деструктор
};
#include <iostream>
#include<conio.h>
using namespace std;
int main () {
Men1 obj1 ("Петров", 34); //создаем статический объект
//Men1 m1; -не верно, в классе не объявлен конструктор без параметров
Men1 * obj2 = new Men1 ("Иванов", 25); //динамический экземпляр класса
cout<<obj1. GetN () << " " << obj1. GetA () << endl;
cout<< obj2->GetN () << " " << obj2-> GetA () << endl;
_getch ();
return 0;}
```

Результат программы:

```
Петров - begin
Иванов - begin
Петров 34
Иванов 25
Петров - end
```

Итак, конструктор вызывался автоматически дважды при создании объектов - статического и динамической памяти.

Деструктор вызывался только один раз. Для объекта *obj2* деструктор не вызывался, так как память на объект выделялась "вручную" и также "вручную" должна быть освобождена, то есть система не констатирует факта уничтожение этого объекта при завершении программы.

Если перед *\_getch ()* вставить строку:

```
delete (obj2);
```

то результат программы будет следующий:

*Петров - begin*  
*Иванов - begin*  
*Петров 34*  
*Иванов 25*  
*Иванов - end*  
*Петров - end*

Причем при выводе, как первого, так и второго результата программа останавливается на вызове *\_getch ()*, выводятся все строки результатов, кроме последней строки, после нажатия клавиши выводится и последняя строка.

Продолжим рассмотрение **конструктора копирования**.

Рассмотрим использование **конструктора копирования** при копировании ресурсоемких объектов. Конструктор копирования, предоставляемый компилятором по умолчанию, производит побитовое копирование полей, что может приводить к абсурдным результатам при копировании ресурсоемких объектов.

Рассмотрим программу, в которой вызов конструктора копирования по умолчанию приводит к "неправильному" копированию.

```
#include <iostream>
using namespace std;
class My {
int* p;
public:
My (int a) //конструктор с параметром
{p= new int ; *p = a; }
int Get () { return *p; } //возвращает значение по адресу указателя
~My() {delete(p);} //освобождает выделенную в конструкторе память
};
void Print (My obj) //функция вывода значения
{ cout << '\n' << obj.Get ();}

int main () {
My a1 (10); // создаем объект, данное инициализируется 10
My a2 = a1; // создаем объект a2 копию объекта a1- вызов конструктора
//копирования по умолчанию

Print (a1);
/* в стеке создается локальный объект obj - копия объекта a1, вызывается
конструктор копирования, значение данного (10) выводится на экран*/
Print (a2); //выводится значение 10
a1.~My (); //вызовом деструктора объекта a1 память выделенная для
//данного освобождается
Print (a1); // выведется мусор
```

```
Print (a2); // выведется мусор
}
```

Результат программы:

```
10
10
7853
7853
```

После вызова деструктора для объекта **a1** освобождается динамическая память и в поле данных (**\*p**) объекта **a1** содержится теперь мусор (операция **delete** в процессе возврата в систему памяти затирает эту память).

Но и в поле данных объекта **a2** (копии **a1**) находится теперь тот же мусор. Результат безусловно не верный.

При отсутствии в определении класса конструктора копирования программа будет использовать конструктор копирования по умолчанию следующего вида:

```
My (My & obj) { p = obj.p; }
```

Поле данных объекта класса – это указатель **p**, адресуемый участок памяти, выделенный операцией **new** в конструкторе с параметрами при создании объекта. Участок памяти предназначен для хранения значения параметра конструктора.

Побитовое копирование объекта **a1** дает копирование значения указателя, то есть в поле данных объекта **a2** заносится тоже значение указателя, и оказывается, что указатели обоих объектов "указывают" на один и тот же участок памяти (рис. 2.1).

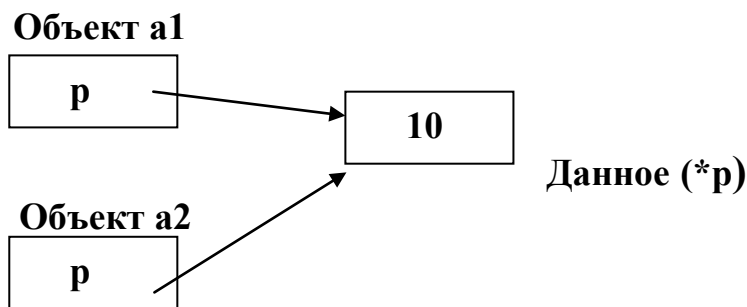


Рис. 2.1. Распределение памяти при копировании объектов по умолчанию

При вызове деструктора для объекта **a1** освобождается память и затирается единственное данное, общее для двух объектов.

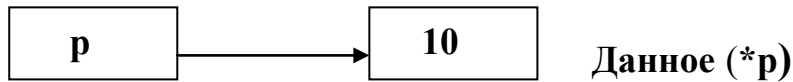
"Уничтожение" объекта **a1** привело к уничтожению и его копии **a2**, в которой находится теперь тот же мусор.

Когда класс содержит поля, являющиеся указателями на динамические участки памяти, для правильного копирования в классе следует определять конструктор копирования вида:

```
My (const My & obj)
{p = new int; *p = *obj.p;},
```

в котором выделять память для данного нового объекта, куда и скопировать значение (рис. 2.2).

**Объект a1**



**Объект a2**

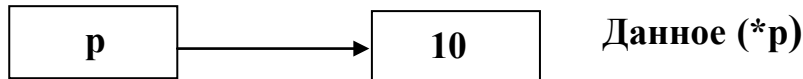


Рис. 2.2. Распределение памяти при копировании объектов с использованием конструктора копирования

В этом случае объектам *a1* и *a2* будут принадлежать разные участки памяти для данных. Уничтожение одного из них вызовом деструктора, никак не отразится на существовании другого.

Ниже приведена правильная программа.

```
#include <iostream>
#include <conio.h>
using namespace std;
class My {
int* p;
public:
My (int a) {p= new int; *p = a; }
My (const My & obj) //конструктор копирования
{ p = new int; *p = *obj.p; }
int Get () { return *p; } // возвращает значение по адресу указателя
~My() {delete(p);} //освобождает память
};
void Print (My obj)
{cout<< '\n' << obj.Get ();}
int main () {
My a1 (10); // создаем объект, данное инициализируется значением 10
My a2 = a1; // с помощью конструктора копирования создаем объект a2,
Print (a1);
Print (a2);
a1.~My (); // вызовом деструктора память для данного a1 освобождается
Print (a1); // мусор
Print (a2); // выведется 10
}
```

Результат программы:

10  
10  
7853  
10

## 2.6. Указатели на поля данных и на методы класса.

Две операции языка C++ `.*` и `->*` предназначены для работы с указателями на поля данных и методы класса.

Формат определения указателя на поле данных класса:

***тип\_данных имя\_класса :: \*имя\_указателя;***

В определение указателя можно включить его явную инициализацию, используя адрес конкретного поля данных из соответствующего класса:

***&имя\_класса :: имя\_поля\_данных;***

При этом поле данных должно быть открытым.

Рассмотрим пример использования указателя на поля данных.

```
struct complex1{
double re, im;
complex1(double _re, double _im) {re=_re; im=_im;}
double real ()
{ return re; }
double image ()
{ return im; }
};
int main (){
complex1 comp (16.0, 30.5); complex1*ptr= ∁
//определение указателя на поле данных класса с инициализацией
double complex1::*pdat=& complex1::re;
cout << (comp.*pdat) <<endl;
pdat= & complex1::im; cout << (comp.*pdat) <<endl;
cout << (ptr->*pdat);
return 0; }
```

Результат приведенных операторов:

16  
30.5  
30.5

Можно определить указатель на компоненты-функции. Вне класса можно следующим образом определить указатель на метод класса:

***тип\_возвращаемого\_методом\_значения***  
***(имя\_класса ::\*имя\_указателя\_на\_метод)***  
***(спецификация\_параметров\_функции);***

Рассмотрим пример, определения и использования указателя на метод.

```
double(complex1::*ptcom)(); //определение указателя на метод класса
ptcom = &complex1::real; // "настройка" указателя на метод real
complex A(5.2,2.7);
```

// Теперь для объекта *A* можно вызвать его метод через указатель  
`cout<<(A.*ptcom)();`

Можно определить также тип указателя на функцию:

`typedef double (complex1::*PF)();`

а затем определить и сам указатель, например, с инициализацией:

`PF ptc=&complex::real;`

## 2.7. Указатель *this*

При вызове нестатической функции класса для обработки данных конкретного объекта, ей неявно передается указатель на тот объект для которого она вызывается. Этот указатель называется *this*, он константный, значением его является адрес объекта для которого вызывается метод и определен он неявно следующим образом:

*имя\_класса \*const this = адрес\_обрабатываемого\_объекта;*

При вызове нестатического метода указатель *this* автоматически инициализируется адресом того объекта, для которого был вызван метод и объект благодаря указателю *this* становится доступный внутри этого метода.

При работе с полями данных можно так использовать указатель *this*:

`class point { int x, y ;`

`public:`

`point( int xx=0, int yy=0)`

`{ this-> x=xx ; this ->y =yy; }; //эквивалентно: x=xx ; y = yy;`

`void print ( void)`

`{cout<< this->x <<" " << this->y; }; // эквивалентно: cout<< x <<" " <<y;`

`};`

В таком использовании нет никаких преимуществ, так как данные объектов доступны в методах класса и с помощью имен полей данных.

Иногда *this* используется при конфликте имен, когда имена формальных параметров метода совпадают с именами полей данных класса:

`class point { int x, y ;`

`public:`

`point( int x=0, int y=0)`

`{this-> x=x ; this ->y =y ;}`

`};`

Для снятия неоднозначности можно с тем же успехом использовать квалифицированные имена полей данных: `point :: x = x; point :: y = y;`

Однако в некоторых случаях использование указателя *this* становится практически незаменимым и удобным. Например, когда в теле метода класса необходимо явно использовать адрес того объекта, для которого метод вызывается. Пример:

`class A{`

`int x, y ;`

`public:`

`A ( int xx=0, int yy =0){ x=xx ; y = yy; }`



```

A func () ;
};
A A :: func () // функция, преобразующая данное x в четное
{ if (x%2) x++;
return *this;
}
int main ()
{ A a1 (17, 55);
A a2 = a1.func ();
}

```

Чаще всего **this** используется при организации связанных списков, звеньями которых должны быть объекты класса и встает необходимость включать в связи указатель на тот объект, который в данный момент обрабатывается.

В качестве примера рассмотрим класс "очередь". Спецификация класса определена в файле **mem.h**:

```

#ifndef _mem
#define _mem
class que {
static que*first ; // указатель(адрес)первого элемента очереди
que*next ; // указатель на следующий элемент очереди
char буква; //содержимое элемента очереди
public: // общедоступные функции
que(char c) { буква = c } ; // конструктор
void add (void) ; // функция добавления элемента в очередь
static void print (void); // вывод содержимого очереди
};
#endif

```

Схема построения очереди показана на рис. 2.3.



Рис.2.3. Схема формирования очереди из объектов класса **que**

В классе *que* имеется статический элемент *first*, общий для всех объектов класса – это указатель на первый элемент очереди. Когда очередь пуста, значение *first* должно быть равно нулевому указателю. Связь между объектами в очереди осуществляется с помощью указателя *next*. Подключение элементов в очередь выполняет метод *add()*. Статическая функция *print()* "перебирает" звенья очереди от начала к концу и выводит символы – данные объектов. Конструктор инициализирует поле данных *char буква* для каждого создаваемого объекта.

В файле *func\_que.h* определим методы класса.

```
#ifndef _method
#define _method
#include <iostream>
//Включаем спецификацию класса
#include "mem.h"
//-----Определения функций-----
//Добавление элемента в конец очереди
void que::add() {
 que* list= first; //текущий указатель устанавливается на начало очереди
 que * uk; // вспомогательный указатель
 while (list!=0) //пока не достигнем конца продвигаемся по очереди
 { uk = list; list=list->next }
 if(uk!=0) {uk->next=this; } //присоединение объекта в конец очереди
 else first = this; //очередь пустая, первому элементу присваивается this
 this->next= 0; }
//Вывод содержимого очереди
void que::print (void)
{ que *list= first; //указатель устанавливаем на начало очереди
 if (list == 0) {cout << "\n Очередь пуста"; return; }
 else cout<<"\nСодержимое очереди: "<<endl;
 while (list!=0)
 {cout<<list->bukva; list=list->next;}
 }
#endif
//Программа – связанная очередь
#include <iostream>
using namespace std;
#include "func_que.h" //Определение класса que
//Инициализация статического поля данных
que * que :: first = 0;
int main()
{ //определяем объекты класса
 que A('a'); que B('b'); que C('c'); que D('d');
 que::print (); // вызов статического метода
```

```
A.add (); B.add (); C.add (); D.add (); //включаем в очередь объекты
que::print (); //выводим очередь
return 0; }
```

Результат выполнения программы:

*Очередь пуста*

*Содержимое очереди:*

*abcd*

## 2.8. Перегрузка функций

C++ позволяет определить в программе произвольное количество функций с одним именем, при условии, что все они имеют разный состав параметров, или сигнатуру. Этот вопрос подробно рассматривался в первой части пособия. Перегрузка функций используется, когда данные и код для их обработки различны, но при этом удобно, чтобы функции назывались одинаково. Перегрузка обычных функций не дает, каких-либо существенных преимуществ, это просто удобно. Другое дело с конструкторами классов.

### Перегрузка конструкторов

Перегрузка конструкторов приобретает особое значение, из-за того, что конструктору нельзя назначать произвольное имя, оно всегда должно совпадать с именем класса.

Наличие в классе нескольких конструкторов, имеющих одно имя, но разный состав параметров, возможно благодаря реализации средства перегрузки функций. Таким образом, перегрузка для конструкторов просто неизбежна.

Мы уже рассматривали конструкторы, которые могут быть включены в класс:

```
class A { int x, y;
public:
A (int, int); // конструктор с параметрами для инициализации данных
A(); // конструктор по умолчанию
A (A&); // конструктор копирования
... };
```

Часто при разработке класса предусматривается несколько вариантов инициализации объектов и значит должно быть несколько конструкторов с параметрами, например,

```
class Window {
int x; // x-координата левого верхнего угла окна
int y; // y-координата левого верхнего угла окна
int w; // ширина окна
int h; // высота окна
public:
Window () // конструктор по умолчанию
{ x = y = 0; w = h = 100; }
Window (int w1, int h1) // конструктор инициализации размеров окна
```

```

{ x = y = 0; w = w; h = h; }
Window (int x1, int y1, int w1, int h1) // инициализация положения и
{ x = x1; y = y1; w = w1; h = h1; } //размеров окна
Window (Window & win) // конструктор копирования
{ x = win.x; y = win.y; w = win.w; h = win.h; }
};

```

Можно создавать, например, следующие объекты:

```

Window w1;
Window w2 (300, 250);
Window w3 (5, 10, 400, 300);
Window w4 = w3;

```

## Раздел 3

### Перегрузка операций

#### 3.1. Расширение действий (перегрузка) стандартных операций

*Перегрузка операций - это распространение действий стандартных операций на операнды, для которых эти операции не предполагались или предание стандартным операциям другое назначение.*

В синтаксисе языка Си такого правила нет. Классы дают такую возможность.

Если операнды операции (или хотя бы один) – объекты некоторого класса, то есть введенного пользователем типа, можно использовать специальную функцию, называемую "*операция – функция*" (*operator function*), определяющую новое поведение операции. Формат определения операции-функции:

```

тип_возвращаемого_значения operator знак_операции
(спецификация_формальных_параметров)
{тело_операции_функции }

```

Механизм перегрузки операций во многом схож с механизмом определения функций, если принять что

- 1) конструкция *operator знак\_операции* есть имя некоторой функции;
- 2) *список\_формальных\_параметров* – список операндов с указанием их типов, которые участвуют в операции;
- 3) *тип\_возвращаемого\_результата* – это тип значения, которое возвращает операция;
- 4) *тело\_операции-функции* – это алгоритм нового действия операции.

Часто перегрузка операции не меняет общий смысл операции, однако в некоторых случаях перегрузка совершенно изменяет назначение операции. Так стандартные операции ">>" и "<<" - это битовые сдвиги, однако при использовании их с объектами потоковых классов эти операции приобретают смысл "извлечения из потока и вставки данных в поток".

Большинство операций перегружаемы, однако, не все.

Операции, не допускающие перегрузки:

- . - операция прямого доступа к методу или полю данных класса;
- . \* - операция обращения к методу или полю через указатель на данный элемент класса;
- ?: - условная операция;
- :: - операция указания области видимости;
- sizeof** – операция вычисления размер объекта;
- # - препроцессорная операция, преобразующая строку замещения;
- ## - препроцессорная операция конкатенации лексем в строке замещения.

Перегрузку можно проводить следующими способами:

1) *операция - функция* является компонентной функцией класса;

2) *операция – функция* является глобальной функцией:

а) *операция - функция* является дружественной функцией класса;

б) *операция - функция* является недружественной функцией класса, но хотя бы один параметр функции (недружественной) был бы объектом или ссылкой на объект некоторого класса.

Количество параметров операции-функции определяется арностью операции и тем, является ли функция глобальной или компонентной.

Рассмотрим все эти случаи на примере перегрузки операция сложения '+' для объектов некоторого класса *A*.

1) *Операция – функция* является *компонентной* функцией класса *A*.

При этом функция не должна быть статической, так как будет вызываться, и обрабатывать обычные (не статические) данные конкретного объекта. При определении *компонентная операция - функция* имеет на один параметр меньше, чем арность операции. Первым операндом такой операции по умолчанию является тот объект, для которого будет вызываться функция.

Определим такую компонентную операцию - функцию внешне, а в классе представим только прототип функции:

```
class A { ...
A operator + (A obj);
};
A A :: operator + (A obj)
{тело_перегрузки}
```

Пусть *B*, *C*, *D* - объекты класса *A*, выражение *B = C+D* следует трактовать как вызов метода с именем *operator +* для объекта *C*:

*B = C.operator+(D);*

Таким образом, *C* – это тот объект, для которого вызывается *операция – функция*, а объект *D* – ее параметр.

Бинарная операция "+" - не симметрична относительно операндов.

- 2) *Операция – функция* является глобальной функцией и  
 а) является дружественной функцией классу *A*:

```
class A {...

friend A operator + (A obj1, A obj2);

};

A operator + (A obj1, A obj2)

{тело_функции_перегрузки}
```

- б) не является дружественной функцией классу *A*:

```
class A {...};

A operator + (A obj1, A obj2)

{тело_функции_перегрузки}
```

*B*, *C* и *D* – объекты класса *A*, выражение  $B = C + D$  следует трактовать как:

$B = \text{operator+}(C, D);$

где *operator+* рассматривать как имя функции, а *C* и *D* – ее параметры.

В этом случае операция "+" - симметрична относительно операндов.

Следует помнить, что дружественные функции имеют доступ к закрытым данным класса, а обычные внешние функции – не имеют доступа, поэтому для перегрузки операций, как правило, используют дружественные функции.

Рассмотрим перегрузку ряда операций для класса *Complex*.

Спецификация класса (файл *Complex.h*):

```
#ifndef _comp

#define _comp

class Complex {

 double re, im;

public:

 Complex (double r = 0.0, double i = 0.0) // конструктор с аргументами,

 : re (r), im (i) {} // задаваемыми по умолчанию

 Complex operator – (); // перегрузка одноместной операции "-", метод класса

 Complex operator – (Complex&); // перегрузка двуместной операции "-",

 // метод класса

friend Complex operator + (Complex&, Complex&); // перегрузка

 //двуместной операции "+" – дружественная функция

friend ostream & operator << (ostream &, const Complex&); // перегрузка

 //операции вывода данных объекта "<<" – дружественная функция

friend istream & operator >> (istream &, Complex&); // перегрузка

 //операции ввода данных объекта ">>" – дружественная функция

};

#endif
```

В файле *func.h* определим методы класса и внешние дружественные функции.

```
#ifndef _func

#define _func
```

```

#include <iostream>
//Включаем спецификацию класса
#include "Complex.h"
//-----Определения функций-----
//----- Перегрузки операций – методы класса-----
Complex Complex :: operator- ()
{return Complex (-re, -im);}
Complex Complex :: operator- (Complex& z) // передача по ссылке, чтобы
{return Complex (re - z.re, im- z.im); } // не копировать объект в стек
// -----Перегрузки операций – внешние, дружественные классу функции-----
Complex operator+ (Complex &z1, Complex &z2)
{ return Complex (z1.re + z2.re , z1.im+ z2.im); }
//дружественная операция – функция для вывода
ostream & operator << (ostream & out, const Complex&z)
{out <<"real =" << z.re <<" , \timage = " << z.im<<endl;
return out;}
//дружественная операция – функция для ввода
istream & operator >> (istream & in, Complex&z)
{cout <<"real = "; in >> z.re; cout <<"image = "; in >> z.im;
return in;}
#endif

```

В дружественной операции – функции для вывода возвращаемым значением и первым параметром является ссылка на объект класса выходных потоков *ostream*. Вторым параметром является константная ссылка на объект класса *Complex* (функция не должна изменять это параметр). Если *obj* объект класса *Complex*, то выражение *cout <<obj* – это вызов операции – функции *operator<<(cout, obj)*. Функция возвращает ссылку на *cout*, а именно леводопустимое выражение. Следовательно, можно организовывать цепочки вывода данных класса *Complex* и других типов.

В операции – функции для ввода возвращаемым значением и первым параметром является ссылка на объект класса входных потоков *istream*, из которого будет выполняться чтение данных. Вторым параметром является ссылка уже не константная на объект класса *Complex*, функция должна изменять это объект. В этой функции также обеспечивается возможность организации цепочки ввода данных вида *cin>> a>> b>> c>> d*, где *f, d, c, d* – как объекты класса *Complex*, так и переменные других типов.

```

//Программа – файл p_01.cpp
#include <iostream>
using namespace std;
#include "func.h" //Спецификация класса
int main(){
Complex c1 (5.1, -2.1);
Complex c2, c3;

```

```

cin >> c2 // вызов operator>>(cin, c2);
c3 = c1 + c2; // копирование в c3 результата вызова operator+ (c1, c2);
cout << "c3 :t" << c3; // operator<< (cout, c3);
c3 = -c3 // c3.operator- ();
Complex c4 = c2-c1; // копирование в c4 результата вызова c2.operator- (c1);
cout<< c1<<c2<<c3<<c4; // "цепочка"
return 0;}

```

Результат выполнения программы:

```

real = 0.4 <ENTER>
image = -2.3 <ENTER>
c3: real = 5.5, image = -4.4
real = 5.1, image = -2.1
real = 0.4, image = -2.3
real = -5.5, image = 4.4
real = -4.7, image = -0.2

```

Здесь <ENTER> - условное обозначение нажатия клавиши.

Видим, что синтаксис правил использования функций - перегрузки операций, определенных как методы класса или как друзья класса, совершенно одинаков.

Однако определение операций - функций как дружественных создает преимущество в смысле симметрии и, главное, в тех случаях, когда для выполнения действий над операндами требуется преобразование типа операндов.

Дело в том, что компилятор автоматически выполняет преобразование типа для аргументов функций, но не для объекта, для которого вызывается функция-член.

Если **функция-оператор** (другое название **операции - функции**) реализуется как друг и получает оба аргумента как параметры функции, то компилятор выполняет автоматическое преобразование типов двух аргументов операции.

Распространение действий операций на объекты вводимого программистом класса служит средством для встраивания (агрегации) класса в систему типов, уже существующих в языке. Например, определяя операция + для комплексных чисел, приходится учитывать сложение комплексного числа с вещественным и целым числом и т. д.

### 3.2. Преобразование типов в классах пользователя

Иногда в выражениях целесообразно использовать переменные класса (объекты) с переменными других типов, для этого необходимо определить правила преобразования типов.

Это можно сделать с помощью конструктора класса с умалчиваемыми значениями, который в этом случае выступает в роли конструктора



преобразования типов, или с помощью операции – функции преобразования типов.

Конструктор класса ***Complex***, с прототипом  
***Complex (double r =0.0, double i=0.0);***

позволяет создавать объекты, передавая в конструктор разное количество параметров:

***Complex c1 (1.5, 2.5);***

***Complex c2 (4.5);***

***Complex c3;***

Наличие в конструкторе умалчиваемых значений делает допустимыми следующие операции комплексного числа с арифметическими данными:

***Complex c4 (5.5, 6.5);***

***Complex c5 = c4+ 2;***

***Complex c6 = c4+'0';***

Рассмотрим выполнение операторов.

В первом операторе автоматически в функции перегрузки операции + целое **2** преобразуется к типу ***double 2.0***, затем создается временный объект с вызовом конструктора с умалчиваемыми значениями, в который передаются параметры ***(2.0, 0.0)***. Таким образом выражение ***c4+ 2*** означает вызов ***operator+ (c4, Complex (double (2)))***.

В результате с использованием конструктора копирования создается объект ***c5*** с данными ***(7.5, 6.5)***, который представляет собой сумму двух комплексных чисел – объекта ***c4*** и объекта с данными ***(2.0, 0.0)***.

Во втором операторе символ **'0'** преобразуется в целое **48** (десятичный код символа), которое затем преобразуется в вещественное **48.0**, и создается временный объект с данными ***(48.0, 0.0)***. Выражение ***c4+'0'*** означает вызов ***operator+ (c4, Complex (double (int('0'))))***.

В результате создается объект ***c6*** с данными ***(53.5, 6.5)***, представляющий сумму двух комплексных чисел – объекта ***c4*** и объекта с данными ***(48.0, 0.0)***.

Иногда бывает необходимость преобразовать переменные некоторого класса в базовые типы.

Проблема решается с помощью специальной **операции - функции преобразования типа**.

В классе определяется метод - перегрузка операции преобразования типа, которая имеет следующий формат:

***operator имя\_нового\_типа () {...}***

Рассмотрим это на примере класса ***stroka***.

***#include <cstring>***

***#include <iostream>***

***using namespace std;***

***class stroka {***

***char\*ch;*** // указатель на строку, на символьный массив

***int len ;*** // длина строки

```

public:
stroka(char*cch) //конструктор 1
{ len = cstrlen(cch) ;
ch=new char [len+1];
strcpy (ch,cch); }
stroka(int N=20) //конструктор 2
{ch = new char[N+1]; len=0; ch[0]='\0' ;}

operator char* () { return ch;} //операции – функции преобразования типов.

void vivod () // выводит данные
{cout<< "строка: " <<ch <<" , длина строки="<<len;};
~stroka() // деструктор
{delete [] ch;}
friend stroka& operator+(stroka&, stroka&); //дружественная функция-операция
};
stroka& operator+(stroka&A, stroka & B)
{ int N = A.len+ B.len;
stroka*ptr=new stroka(N); // выделяем память на суммарную строку
strcpy(*ptr , A) ;// копируем в новую строку первую строку
strcat(*ptr , B) ; // присоединяем к ней и вторую строку
return *ptr ; //возвращаем объект
}

```

В класс включен метод преобразование типа *stroka* в тип *char\**.

Чтобы объекты класса можно было бы передавать функциям модуля *<cstring>*, надо в описание класса включить компонентную функцию:

```
operator char* () {return ch;}
```

И, например, в операторах

```
stroka s1("string");
char* s; s=s1;
```

будет происходить автоматическое преобразование объекта *s1* к типу *char\**, и переменной *s* присвоится значение *"string"*.

Можно было определить в классе преобразование объекта к целому значению:

```
operator int () {return len;}
```

И тогда допустим оператор:

```
int l = s1;
```

то есть объект *s1* преобразуется к целому значению.

Рассмотрим еще несколько важных особенностей механизма перегрузок (расширения действия) стандартных операций C++:

- 1) C++ запрещает вводить операции с новым обозначением.
- 2) Нельзя изменить приоритет стандартной операции, перегрузив ее.
- 3) Нельзя изменять аргументность операции.

4) Перегрузка бинарной операции определяется либо как метод класса с одним параметром, либо как внешняя функция, возможно дружественная, с двумя параметрами. Выражение  $X <операция> Y$  означает вызовы:

*X.operator <операция> (Y);* // если *операция-функция* - метод класса  
*operator <операция> (X, Y)* // если *операция-функция* –внешняя.

5) Перегрузка унарной операции определяется либо как компонентная функция без параметра, либо как внешняя функция, возможно дружественная, с одним параметром. Выражение  $<операция> X$  означает вызовы:

*X.operator <операция> ()* // если *операция-функция* - метод класса  
*operator <операция> (X)* // если *операция-функция* –внешняя

6) В соответствие с семантикой бинарных операций  $=, [, ->$  операции-функции с названием *operator=*, *operator [ ]*, *operator->* не могут быть внешними функциями, а должны быть нестатическими методами того класса, для которого они определены.

7) Унарные операции, имеющие префиксную *@obj* и постфиксную *obj@* формы, (здесь *@* обозначает знак операции, *obj* – объект некоторого класса) в соответствии со Стандартом языка выполняются по-разному. Префиксная операция выполняется в соответствие с общими правилами (пункт 5), а постфиксная операция-функция должна иметь дополнительный параметр, который всегда равен нулю.

Особенности перегрузки префиксных и постфиксных операций рассмотрим на примере операций *инкремента* ( $++$ ) и *декремента* ( $--$ ), которые могут быть префиксными и постфиксными.

Определим некий класс "пара чисел", при этом перегрузку операции инкремента произведем с помощью дружественной функции-операции, а декремента – с помощью функции-операции, являющейся методом класса. Текст определения класса и дружественных функций (файл pair.h):

```
#include <iostream>
using namespace std;
class pair {
int N; double X;
//дружественные операции-функции
friend pair & operator ++ (pair &); // префиксная
friend pair & operator++ (pair& , int); // постфиксная
friend ostream & operator<< (ostream &, pair);
public:
//методы:
pair (int n=0 , double x=0.0): N(n), X(x) {} //конструктор
pair& operator- - () // префиксная
{N - = 1; X - = 1.0;
return * this ; }
pair& operator- - (int k) // постфиксная
```

```

{N -= 1; X -= 1.0;
return *this;}
};
pair & operator ++ (pair & P) // префиксная
{P.N += 1; P.X += 1.0; return P; }
pair & operator ++ (pair & P, int k) // постфиксная
{P.N += 1; P.X += 1.0; return P; }
ostream & operator<< (ostream &out, pair p)
{out<<"\tN= "<<p.N<<"\tX= "<<p.X<<endl; return out;}

```

Программа (файл P\_02):

```

int main ()
{ pair A (9, 19.0);
cout<<"A: "<<A;
cout<<"++A: " << ++A; // operator ++(A)
cout<<"A++: " << A++; // operator ++(A,0)
cout<<"--A: " << --A; // operator --()
cout<<"A- -: " << A- -; // operator --(0)
cout<<"operator ++(A): " << operator ++(A);
cout<<"operator ++(A, 0): " << operator ++(A, 0);
cout<<"operator --(): " << operator --();
cout<<"operator --(0): " << operator --(0);
return 0;
}

```

Результат выполнения программы:

|                    |        |        |
|--------------------|--------|--------|
| A:                 | N = 9  | X = 19 |
| ++A:               | N = 10 | X = 20 |
| A++                | N = 11 | X = 21 |
| --A:               | N = 10 | X = 20 |
| A- -:              | N = 9  | X = 19 |
| operator ++(A):    | N = 10 | X = 20 |
| operator ++(A, 0): | N = 11 | X = 21 |
| operator --():     | N = 10 | X = 20 |
| operator --(0):    | N = 9  | X = 19 |

### 3.3 Перегрузка операции присваивания

Перегрузку операции присваивания ("="), как было сказано выше, разрешается реализовывать только через компонентную функцию класса.

Рассмотрим перегрузку для класса комплексных чисел:

```

Complex& Complex :: operator = (Complex& z)
{re = z.re; im = z.im;
return *this; }

```

При выполнении

```
c2=c1;
```

операция функция вызывается для объекта *c2*, объект *c1* передается в нее как параметр. Данным объекта *c2* присваиваются данные объекта *c1*.

Функция возвращает текущий объект для того, чтобы можно было использовать операцию множественного присваивания:

*c3 = c2 = c1;*

Заметим, что в данном случае перегружать операцию присваивания нет особой необходимости. Если не перегружать "=", то компилятор генерирует операцию присваивания по умолчанию, которая выполняет побайтовое копирование данных.

Однако в некоторых случаях, при наличии в классе данных, являющихся указателями на динамические участки памяти, буквальное копирование может привести к утечкам памяти (рис. 3.1).

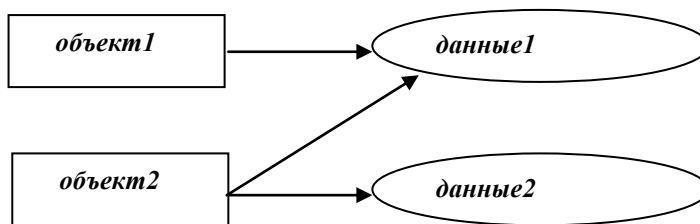


Рис. 3.1. Распределение памяти при выполнении операции присваивания

При прямом копировании *объект2=объект1* указателю *объекта2* присвоится значение указателя *объекта1*. Участок памяти с *данными2* станет недоступным, в программе происходит утечка памяти. Уничтожение данных *объекта1* автоматически уничтожит данные *объекта2*.

В этом случае необходимо перегрузить операцию присваивания, как в следующем примере:

```
class A {
 int* data; //поле данных - указатель на динамические участки памяти
 A (int a) //конструктор с параметром
 { data = new int;
 *data= a; }
 A (A&z) //конструктор копирования
 { data = new int;
 data=*(z.data);}
 A& operator= (const A & z) //перегрузка операции присваивания
 {delete data; // или { *data = *(z.data);
 data = new int; // return *this ;}
 *data = *(z.data);
 return * this}
 ...
};
```

**Различие между копированием и присваиванием.**

Дублирование объекта может быть выполнено и с помощью операции присваивания, и с помощью вызова конструктора копирования.

```
class A { int x;
public:
A (int ax=0){ x=ax; }
int GetX () { return x; };
void main ()
{ A m1(5) , m2;
m2 = m1; // операция присваивания
A m3 = m1 ; // дублирование вызовом конструктора копирования
cout<< m1.GetX () << m2.GetX () << m3.GetX ();
}
```

В предложении:

```
m2 = m1;
```

передача значений выполняется в уже созданный объект. Дублирование объекта производится операцией присваивания по умолчанию.

В предложении:

```
A m3 = m1;
```

создается новый объект, и ему передаются данные копируемого объекта. В этом случае вызывается конструктор копирования по умолчанию.

В ряде случаев объекты должны создаваться в единственном экземпляре, то есть создание идентичных объектов должно быть запрещено.

### Блокировка копирования и присваивания

Для того, чтобы исключить непреднамеренное дублирование объектов класса, конструктор копирования и перегрузку операции присваивания следует описывать в закрытой области класса после спецификатора *private*:

```
private:
```

```
A (A&);
```

```
A operator = (A &);
```

И тогда ни одно из приведенных в главной функции дублирований объектов скомпилировать не удастся.

## Раздел 4

### Включение и наследование классов

#### 4.1. Отношения включения

Если в одном проекте определены и используются несколько классов, то между их объектами могут быть различные отношения. Например, независимость классов. В более сложных случаях семантика отношений между классами может быть реализована по схеме *наследования* или по схеме *включения*.

Об отношении *включения* говорят, используя выражение "включает как часть" (*has a* – владеет, содержит в себе как часть).

При *наследовании* базовый класс – представляет объекты общего вида, производный класс описывает более конкретные объекты, которые являются разновидностью (частным случаем объектов базового класса).

Об отношении наследования можно сказать, используя выражение "является частным случаем" (*is a*).

Например, самолет является частным случаем транспортного средства. Это отношение наследования классов.

Самолет имеет крылья, мотор – здесь реализуется отношение включения.

Рассмотрим на примере отношения включения между классами [3].

Определим класс "точка на плоскости".

```
class point {
double x, y;
public:
point (double x1=0.0, y1=0.0): x(x1), y(y1) {}
friend ostream & operator <<(ostream &out, point p);
};
ostream & operator <<(ostream &out, point p){
out<< "x= " << p.x << "\ty= " << p.y<<endl; return out;}
```

В классе определен только конструктор с параметрами, играющий роль и конструктора приведения типа и конструктора умолчания. Конструктор копирования и операцию присваивания компилятор добавит автоматически.

Определим класс "окружность", включающий в качестве поля данных объект класса *point*.

```
class circle {
point center;
double radius;
public:
circle(point c, double r):
center(c), radius(r){ } //вызов конструктора копирования класса point ,
// созданного компилятором автоматически
circle(double x1, double x2, double r):
center(x1, x2), radius(r){ } //вызов конструктора общего вида, определённого в
// классе
friend ostream & operator <<(ostream &out, circle circ);
};
ostream & operator <<(ostream &out, circle circ){
out<<"Center: " << circ.center;
out<<"\tRadius: " << circ.radius<<endl;
return out;}
```

В основной программе создадим объекты обоих классов.

```
int main(){
point p;
circle cir1(p, 9.7);
```

```
cout<<cir1;
circle cir2(50,70, 20);
cout<<cir2;
return 0;}
```

Результат:

```
Center: x=0 y=0 Radius: 9.7
Center: x=50 y=70 Radius: 20
```

## 4.2. Отношения наследование

**Наследование** – одна из наиболее фундаментальных концепций ООП.

Суть концепции в следующем. Одни классы можно трактовать как, классы для определения объектов общего вида, так называемые **базовые классы**. Пользователь может создавать **производные классы (порожденные, классы потомки, наследники)**, которые описывают более конкретные объекты, являющиеся разновидностью объектов базового класса. Классы потомки могут наследовать возможности родительских базовых классов (поля данных и методы), при этом производные классы могут пополняться собственными компонентами (данными и собственными методами).

**Наследование – это процесс создания новых классов–наследников на основе уже существующих классов, называемых базовыми.**

Допускается **множественное наследование** - возможность для некоторого класса наследовать компоненты нескольких базовых классов, несвязанных между собой.

Простейший синтаксис определения (спецификации) производного класса:

```
ключ_класса имя_производного_класса:
 список_спецификаторов_базовых_классов
{поля_данных_и_методы_производного_класса};
```

где **ключ\_класса** – одно из служебных слов **struct, class**. Следует обратить внимание, что ни базовый, ни производный класс не могут быть объявлены с помощью **union**. Классы **union** не могут использоваться при наследовании!

Спецификаторы базовых классов в списке разделены запятыми и могут быть представлены одним из следующих конструкций:

- 1) **спецификатор\_доступа имя\_класса**
- 2) **virtual спецификатор\_доступа имя\_класса**
- 3) **спецификатор\_доступа virtual имя\_класса**

Производный класс получая в наследство поля и методы базового класса, не перемещает к себе наследуемые компоненты, они остаются в базовом классе. Однако в каждый объект производного класса входит безымянный объект базового класса со всеми своими полями данных и методами.



При наследовании классов важную роль играет статус доступа компонентов базового класса и спецификатор доступа в определении производного класса.

При наследовании относительно доступности компонентов принято следующее соглашение:

- 1) собственные (***private***) методы и данные доступны только внутри класса, где они определены.
- 2) защищенные (***protected***) компоненты доступны внутри класса, где они определены, и также доступны во всех производных классах.
- 3) общедоступные (***public***) компоненты класса – глобальны, т.е. доступны из любой точки программы.

На доступность унаследованного компонента влияют: 1) статус доступа компонента в базовом классе, 2) каким ключом класса вводится производный класс (***class*** или ***struct***) и 3) какой спецификатор доступа стоит перед базовым классом в определении производного класса.

Если производный класс имеет спецификатор *class*, то компоненты родителя получают по умолчанию в классе потомке статус - *private*, - если это *struct* – унаследованные компоненты имеют статус *public* или *protected*.

Рассмотрим сказанное на примере:

```
class A {
protected: int x;
public: double y ;...};
class B: A { ... }; // x, y наследуются как private
struct C: A { ... }; // x наследуются как protected, а y - как public
```

Изменить статус доступа, получаемый по умолчанию можно опять с помощью спецификаторов *public*, *protected* и *private*.

|                                      |                                                                               |
|--------------------------------------|-------------------------------------------------------------------------------|
| <i>class D: protected A { ... };</i> | // <i>x, y</i> наследуются как <b>protected</b>                               |
| <i>class E: public A { ... };</i>    | // <i>x</i> наследуются как <b>protected</b> , а <i>y</i> - как <b>public</b> |
| <i>class M: private A { ... };</i>   | // <i>x, y</i> наследуются как <b>private</b>                                 |

|                                                   |                                                                                                           |
|---------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| <b><i>struct N: protected</i></b> <i>A {...};</i> | // <b><i>x, y</i></b> наследуются как <b><i>protected</i></b>                                             |
| <b><i>struct P: public</i></b> <i>A {...};</i>    | // <b><i>x</i></b> наследуются как <b><i>protected</i></b> , а <b><i>y</i></b> - как <b><i>public</i></b> |
| <b><i>struct N: private</i></b> <i>A {...};</i>   | // <b><i>x, y</i></b> наследуются как <b><i>private</i></b>                                               |

Рассмотрим пример наследования:

[illegible]

```

public :
B (int yy=0) { y = yy ; cout << " B ! "; } // конструктор класса B
int GetY { return y ; } // функция-метод класса B
~B () { cout << " DB "; } // деструктор класса B
};
int main () {
B b (5) ;
cout<<endl << " b = " << b.GetY () ;
cout<<endl << " a = " << b.GetX () ;
return 0 ; }

```

В производный класс **B** включаются все данные и функции родителя **A**, при этом данное **x** - недоступно для прямого обращения из объектов класса **B**, но к нему можно обращаться из доступных компонентных функций класса **A**, которые стали полноправными членами класса **B** и, следовательно, допустим вызов: **b.GetX()**. Открытая функция **GetY()** производного класса **B** предоставила доступ к закрытому данному **y** производного класса.

В **main** создается объект производного класса **B**, компонентному данному которого передается значение **5**. Данное **x** базового класса иницируется умалчиваемым значением **0**.

*При создании объекта производного класса сначала автоматически вызывается конструктор базового класса, который участвует в создании объекта базового класса, после этого вызывается конструктор производного класса, который проводит инициализацию полей данных производного класса.*

В нашем случае сначала вызывается конструктор **A()**, который по умолчанию иницирует **x** значением **0**, а затем вызывается конструктор **B(5)**, иницирующий **y** значением **5**.

*Деструкторы автоматически вызываются в обратном порядке в соответствии с порядком уничтожения объекта. Сначала уничтожается то, что добавилось в производном классе, а затем и базовая часть.*

Результат программы:

```

A ! B !
b= 5
a= 0
DB DA

```

Наличие производного класса не запрещает создавать отдельно объекты базового класса, например, **A a ( 7 )**; Если **A** – базовый класс, а **B** – производный класс, то их отношения можно представить графически (рис. 4.1).

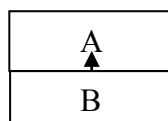


Рис. 4.1. Отношения наследования классов

Итак, при наследовании:

- производный класс беспрепятственно обращается к доступным для него полям данных и методам базового класса;
- базовый класс не имеет доступа к полям данных и методам производного класса;
- в объект производного класса включаются поля данных и методы базового класса, то есть в объект производного класса входит экземпляр объекта базового класса;
- в производном классе у наследуемой функции базового класса может быть "разная судьба".

Возможны три варианта.

1. Производный класс получает некоторую функцию  $A::f()$  без каких либо изменений и соответственно вызывает ее со своим экземпляром наряду со своими родными функциями.

2. Производный класс может заменить унаследованную функцию  $A::f()$  своей  $B::f()$ . У функции производного класса  $B$  та же спецификация параметров, в этом случае говорят, что метод производного класса "экранирует" одноименный метод базового класса.

3. Производный класс может определить функцию с тем же именем, но с другой спецификацией параметров – имеет место перегрузка.

В случаях 2 и 3 алгоритм функции  $B::f()$  может быть абсолютно независим от алгоритма функции  $A::f()$  базового класса. Однако функция  $B::f()$  может дополнить действия функции  $A::f()$ . Для этого в теле функции вызывается функция  $A::f()$ , указывая область видимости: **имя\_класса::имя\_компонента.**

### Передача параметров в базовый класс

Обычно при создании объекта производного класса требуется инициализировать данные не только производного, но и базового класса. Для этого в конструкторе производного класса надо явно вызвать конструктор базового класса. Рассмотрим этот вопрос на примере:

```
class A {
int x1, x2 ;
public:
A (int ax1 , int ax2) { x1 = ax1 ; x2 = ax2; }
};
class B : public A {
int y ;
public :
B (int _x1, int _x2 , int _y): A (_x1 , _x2) { y = _ y ; }
};
```

В конструкторе производного класса перечисляются в качестве параметров все переменные как производного, так и базового класса, которые надо инициализировать (с написанием их типов).

После ":" проводится вызов конструктора базового класса с перечисленными выше параметрами для базового.

При вызове конструктора производного класса **B()**, ему необходимо передать три параметра, два из которых будут переданы конструктору базового класса **A ()**.

При следующем определении объекта производного класса:

**B b (2, 3, 4);**

значения **2** и **3** будут переданы полям данных базового класса **x1** и **x2**, а значение **4** – переменной производного класса **y**.

**Для правильного построения конструктора производного класса необходимо иметь описание конструктора базового класса!**

### Конструкторы с инициализацией по умолчанию в иерархии классов

Рассмотрим в качестве примера конструктор производного класса с инициализацией по умолчанию.

```
class A { // базовый класс
int x1, x2 ; //закрытые данные
public: //открытые функции
A(int ax1, int ax2) { x1=ax1 ; x2 = ax2 ;} //(1) - конструктор с параметрами
int GetX1 () { return x1 ;}
int GetX2 () { return x2 ;} } ;
class B: public A{ //производный класс
int y;
public:
//(2) - конструктор с умалчиваемыми значениями
B (int ax1 =1, int ax2 = 2 , int y1 = 3) : A (ax1, ax2) { y = y1}
int GetY () { return y ;} } ;
int main () {
b1 (10, 20, 30);
cout<<endl<< b1.GetX1 () << " " <<b1.GetX2 () << " " << b1.GetY ();
// 10 20 30
B b2 (10, 20);
cout<<endl<< b2.GetX1 () << " " <<b2.GetX2 () <<" " << b2.GetY ();
//10 20 3
B b3 (10);
cout<<endl<< b3.GetX1 () << " " <<b3.GetX2 () << " " << b3.GetY ();
//10 2 3
B b4;
cout<<endl<< b4.GetX1 () << " " <<b4.GetX2 () << " " << b4.GetY ();
//1 2 3
```

```
return 0;
}
```

В данном примере конструктор базового класса не имеет параметров по умолчанию, то есть, если бы мы создавали просто объект базового класса, необходимо было бы указывать оба параметра в обязательном порядке:

**A a (40, 50);**

В производном классе **B** конструктор имеет параметры по умолчанию для всех данных и производного и базового класса. Это позволяет создавать объекты производного класса, передавая в конструктор разное количество параметров.

Если бы конструктор базового класса имел параметры по умолчанию, то эти значения могли бы реализовываться только в объектах базового класса:

**A( int ax1=4, int ax2=5 ) { x1= ax1 ; x2 = ax2 ;} //(3) – конструктор умолчания**  
*// базового класса*

И тогда результат следующего фрагмента:

**A a;**  
**cout<<endl<< a.GetX1 ( ) << " " <<a.GetX2 ( );**  
 будет: 4 5

При создании объекта производного класса, полям данных базового класса будут присвоены значения, переданные для них при вызове конструктора производного класса (2), либо значения по умолчанию из конструктора опять же производного класса, если таковые имеются, несмотря на умалчиваемые значения в конструкторе базового класса.

Если при построении объектов производного класса нас устраивает инициализация по умолчанию данных базового класса (3), то в конструкторе класса **B** можно не вызывать конструктор базового класса **A**.

Если в производном классе **B** отсутствуют собственные данные, и нас устраивают значения по умолчанию базового класса **A**, то конструктор в классе **B** вообще не нужен, при создании объекта производного класса:

**B b;**

в объекте **b** данные базового класса **A** инициализированы значениями по умолчанию из конструктора базового класса (3).

Если в производном классе нет своих данных, но при создании объектов производного класса нас не устраивают значения по умолчанию из конструктора базового класса. Тогда в конструкторе класса **B** нужно вызвать конструктор класса **A**, чтобы передать ему нужные аргументы.

Приведенные примеры естественно не рассматривают всех возможных ситуаций построения производных классов.

Рассмотрим еще пример, иллюстрирующий доступность компонентов при наследовании [3].

Определим класс "точка" следующим образом:

```

class point {
double x, y; //координаты точки
public:
point (double x1=0.0, double y1=0.0):
x(x1), y(y1) {}
void move(double dx, double dy) {x+=dx; y+=dy;} //перемещение точки
void display()
{cout<<"x= "<<x<<"\ty= "<<y<<endl;} //вывод значения координат
};

```

Определим производный класс "эллипс":

```

class ellipse : public point {
double dmin, dmax; //полуоси эллипса
public:
ellipse (double dmin1, double dmax1, double x1, double y1):
point(x1, y1), dmin(dmin1), dmax(dmax1) {}
void display()
{cout<<"Centre: ";
point:: display ();
cout<<"dmin = " <<dmin<<"\tdmax= " <<dmax<<endl;}
};
double square () {return 3.14159*dmin*dmax;}
};
int main (){
ellipse elli (50, 120, 100, 200);
elli.display(); //метод класса ellipse
elli.move(-5.0, 5.0); //наследуемый метод класса point
elli.display();
cout<<" square = " << elli.square();
return 0;}

```

Любой производный класс может в свою очередь быть базовым для других классов и таким образом формируется цепочка наследования, называемая *иерархией классов*. Иерархия классов определяет для каждого класса приложения родственные связи ("родитель - потомок") его с другими классами приложения.

Класс является *прямым базовым классом*, если он входит в список базовых при определении производного класса.

А если сам базовый класс является производным от некоторого родителя, причем этот родитель не входит в список базовых классов, то этот родитель является *непрямым (косвенным) базовым классом*.

Иерархию производных классов принято отображать в виде *направленного ациклического графа (НАГ)*, где стрелкой изображают связь "производный от".

Производные классы располагаются ниже базовых. В том же порядке они должны располагаться в программе и так их объявления рассматривает компилятор (рис. 4.2).

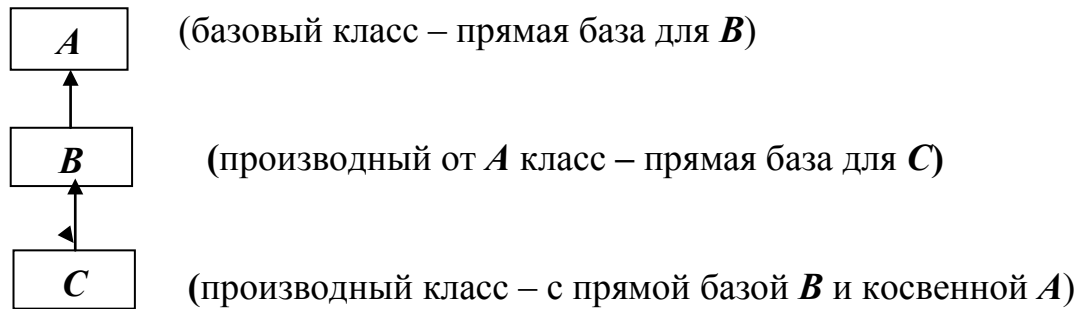


Рис. 4.2. Иерархия наследования

Последовательность объявления классов должна быть такой:

```

class A {...};
class B: public A {...};
class C: public B {...};

```

#### 4.3. Множественное наследование. Виртуальные базовые классы

На практике часто возникает необходимость создать производный класс, наследующий возможности нескольких классов.

Наличие в определении производного класса несколько прямых базовых классов называют **множественным наследованием**. Пример (рис.4.3):

```

class A { ... } ;
class B { ... } ;
class C { ... } ;
class D : public A , public B , public C { ... } ;

```

Родители в определении перечисляются через запятую.

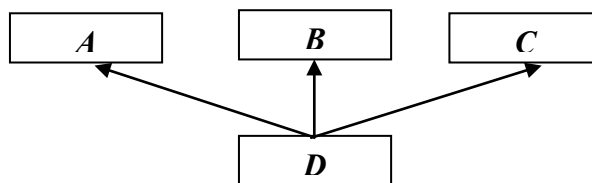


Рис. 4.3 Схема множественного наследования

Как и в случае одиночного наследования, при создании объекта производного класса сначала конструируются объекты базовых классов (в том порядке, в котором базовые классы перечислены в объявлении производного), и лишь после этого составляется объект производного класса.

Деструкторы выполняются в обратном порядке.

При множественном наследовании никакой класс более одного раза не может быть прямым базовым классом. Однако класс более одного раза может

быть непрямым базовым. Пример дублирования базового класса приведен на рис. 4.4.

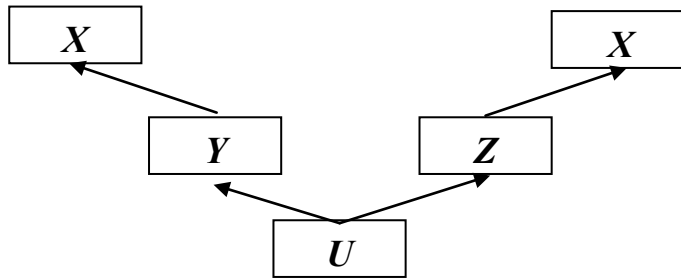


Рис.4.4. Дублирование непрямого базового класса

И соответственно объявления классов:

```
class X {long double x;};
```

```
class Y: public X {double y;};
```

```
class Z : public X {int z;};
```

```
class U:public Y, public Z {...};
```

Размеры объектов при дублировании базового класса:

```
sizeof(X)= 12 (long double)
```

```
sizeof(Y)= 20 (long double+ double)
```

```
sizeof(Z)= 16 (long double+ int)
```

```
sizeof(U)= 36 (long double+ double + long double+ int)
```

При наследовании, особенно множественном могут возникать неоднозначности при доступе к одноименным компонентам разных базовых классов. Способ устранения неоднозначностей – использование квалифицированных имен компонентов (включающих имена классов и операцию принадлежности "::"). Для данного примера:

*U::Y::X::x* или *U::Z::X::x*

Чтобы устранить дублирование объектов непрямого базового класса при множественном наследовании, этот базовый класс наследуется как **виртуальным**. Слово *virtual* помещается в спецификатор базового класса. Причем это делается не в объявлении самого базового класса (X), а в классах, производных от него. Пример графа приведен на рис. 4.5.

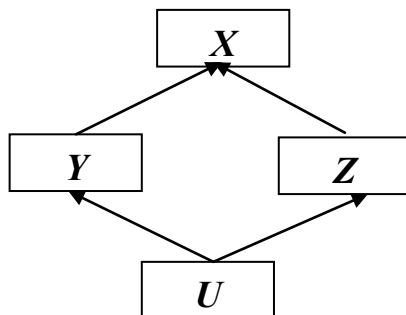


Рис.4.5. Виртуальное наследование классов

И соответствующее определение классов:



```

class X {long double x;};
class Y: virtual public X {double y;};
class Z : virtual public X {int z;};
class U:public Y, public Z {...};

```

При реализации виртуального наследования компилятор добавляет в производный класс в качестве данного указатель на виртуальный базовый класс. Это можно обнаружить, если определить размеры объектов классов с виртуальным базовым.

Размеры объектов без дублирования базового класса:

```

sizeof(X)= 12 (long double)
sizeof(Y)= 24 (long double+ double+ type*)
sizeof(Z)= 20 (long double+ int + type*)
sizeof(U)= 32 (long double+ double + int+ type*+ type*)

```

Итак, класс производный от виртуального включает:

- 1) объект (данные) базового класса;
- 2) указатель на объект базового класса;
- 3) данные производного класса.

Отметим, что виртуальность класса – это не свойство класса, а результат особенностей процедуры наследования.

Один и тот же класс при множественном наследовании может, включен в производный класс при непрямом наследовании и как виртуальный и как не виртуальный (рис.4.6).

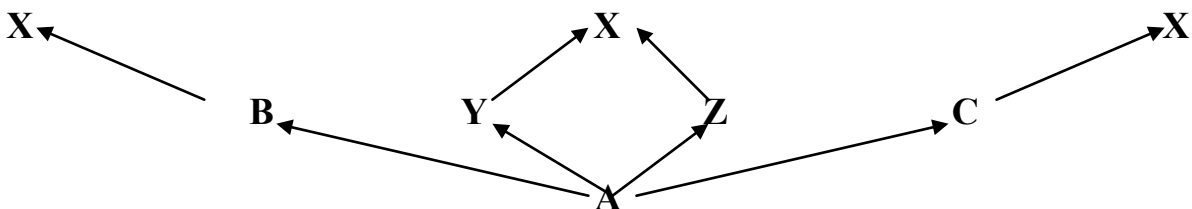


Рис.4.6. Виртуальное и не виртуальное наследование

Определение классов примера:

```

class X { long double } ;
class Y : virtual public X { double y } ;
class Z : virtual public X { int z } ;
class B : public X { int b } ;
class C : public X { int c } ;
class A : public B , public Y , public Z , public C { ... } ;

```

Размеры объектов:

```

sizeof(X)= 12 (long double)
sizeof(Y)= 24 (long double+ double+ type*)
sizeof(Z)= 20 (long double+ int + type*)
sizeof(B)= 16 (long double+ int)
sizeof(c)= 16 (long double+ int)

```

***sizeof(A)= 64 (long double+ long double + long double +int + int + double + int type\*+ type\*)***

Объект класса **A** включает три экземпляра класса **X**: один виртуальный, общий для классов **Y** и **Z**, и два не виртуальных, относящихся к классам **B** и **C**.

Виртуальный класс может быть прямым родителем (рис.4.7):

```
class X {...};
class A : virtual public X{...};
class B : virtual public X{...};
class D : public A, public B , virtual public X {...};
```

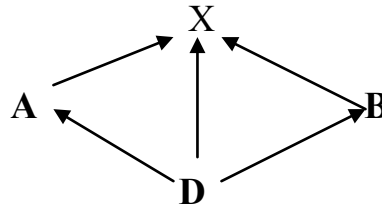


Рис.4.7. Виртуальное наследование

### Локальные классы

*Локальные классы* (ЛК) - это классы, определенные внутри блока (например, внутри функции, внутри другого класса). Перечислим их особенности:

- 1) недоступность компонент класса вне области его определения;
- 2) в классе разрешено использовать статические данные из охватывающего класс блока и внешние для этого блока данные и функции;
- 3) ЛК не может иметь статических компонентов;
- 4) в ЛК запрещено использовать переменные (объекты) автоматической памяти, определенные в охватывающем класс блоке;
- 5) методы ЛК могут быть определены только внутри его спецификации, тем самым они создаются только как подставляемые (*inline*) функции.
- 6) функция, включающая определение ЛК, не имеет особых прав доступа к его компонентам. Действуют общие правила локализации объектов.

### 4.4. Методы при наследовании классов

Поля данных базового класса входят в каждый объект производного класса, и их доступность для объектов производного класса мы рассматривали выше. В свою очередь, методы производного класса могут быть

- унаследованы;
- написаны программистом;
- созданы компилятором, независимо от действий программиста.

В зависимости от того какие методы явно определил программист в производном классе, компилятор может добавить или не добавить так называемые *специальные методы*:

- конструктор умолчания
- конструктор копирования;

- деструктор;
- операцию - функцию присваивания,

которые сильно влияют на создание, копирование и уничтожение объектов.

Нередко указанные функции создаются, наследуются и вызываются неявно.

Но при наследовании в производном классе в ряде случаев необходимо обращаться явно, даже к неявно определенным специальным базовым методам.

***Конструкторы и операция присваивания не наследуются!***

То есть если в производном классе не определена операция присваивания компилятор создаст ее автоматически по своему усмотрению.

Если в производном классе нет конструкторов, то конструктор умолчания и конструктор копирования будут в нем определены компилятором.

***Деструктор базового класса не может быть закрытым.*** Он может быть только открытым или защищенным. Деструктор производного класса вызывается раньше, чем деструктор базового, он автоматически вызывает деструктор базового класса. При множественном наследовании деструкторы базовых классов вызываются в обратном порядке по отношению к перечислению базовых классов.

### **Присваивание при наследовании**

Рассмотрим простейшие случаи наследования классов:

```
struct Bas{int b;}
```

```
struct Dir:Bas{ double d;}
```

В классах неявно определены конструкторы, деструкторы и операция функция ***operator=()***.

```
int main (){
```

```
Dir one, two; //конструктор умолчания
```

```
one.b=12;
```

```
one.d=8.8;
```

```
two=one; //выполняется Dir::operator =
```

```
cout <<" two.b=" <<two.b<<"\t two.d= " <<two.d<<endl;
```

```
return 0;
```

```
}
```

Результат:

```
two.b= 12 two.d=8.8
```

В данном примере все корректно. При создании объектов использовались конструкторы умолчания. При выполнении присваивания вызваны операции-функции перегрузки присваивания неявно определенные в обоих классах.

Без участия программиста при создании объектов из конструктора ***Dir()*** был вызван конструктор ***Bas()***, а при присваивании из метода ***Dir::operator=()*** был неявно вызван метод ***Bas::operator=()***.

Так бывает не всегда, например, в тех случаях, когда классы определяют ресурсоемкие объекты или, когда значения полей производного класса зависят

от значения полей базового класса. При явном определении метода производного класса нельзя надеяться, что компилятор автоматически и правильно выполнит обращение к методу базового класса.

**Нужный метод базового класса (даже при отсутствии его явного определения) необходимо явно вызывать из соответствующего метода производного класса.**

Пусть в классе *Dir* мы хотим явно определить операцию-функцию перегрузки присваивания, определение может быть таким:

```
Dir&Dir::operator=(const Dir&x){
if(this==&x)return *this;
//вызов перегрузки для базовой части
Bas::operator= (dynamic_cast<const Bas&>(x)); //приведение ссылки на объект
//производного класса к значению ссылки на объект базового класса
this ->d=x.d;
return *this;};
```

### Перегрузка операций ввода/вывода при наследовании

Рассмотрим какие особенности появляются при перегрузке операций ввода/вывода для классов, находящихся в отношении наследования. Перегрузку операций ввода/вывода нельзя выполнить с помощью методов класса, чаще эти операции вводятся как дружественные. Покажем примером обращение из дружественных функций производного класса к дружественным функциям базового класса (рис. 4.7).

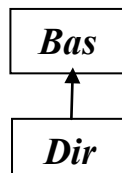


Рис. 4.7. Наследование классов

```
//определение базового класса
class Bas {
protected: int k;
friend istream & operator >> (istream&in, Bas&b);
friend ostream & operator << (ostream&out, const Bas&b);
};
istream & operator>>(istream&in,Bas&b)
{cout<<"k= ";
in>>b.k;
return in;
}
ostream & operator<<(ostream&out, const Bas&b)
{out<<"k= "<<b.k<<endl;
return out;
```

```

}
//определение производного класса
class Dir : Bas {
double z;
friend istream & operator>>(istream&in, Dir&d);
friend ostream & operator<<(ostream&out, const Dir&b);
};
istream & operator>>(istream&in,Dir&d)
{cout << "Bas: ";
operator >> (in, dynamic_cast< Bas&>(d));
cout<< "Dir::z= ";
in>>d.z;
return in;
}
ostream & operator<<(ostream&out, const Dir&d)
{ out << "Bas: ";
out << dynamic_cast<const Bas&>(d);
out << "Dir::z= " << d.z << endl;
return out;
}

```

## Раздел 5.

### Виртуальные функции и абстрактные классы

#### 5.1. Виртуальные функции

Виртуальные функции включены в C++ для обеспечения одного из видов **полиморфизма**.

Термин **полиморфизм** дословно означает "множество форм".

Применительно к языкам, говорят о полиморфизме, когда функции или операторы в различных условиях проявляют себя по-разному.

Процедурный полиморфизм мы уже рассматривали – перегрузка функций. В этом случае имя функции становится многозначным – ему соответствуют разные алгоритмы.

Еще вид процедурного полиморфизма – это перегрузка операций.

Другой вид полиморфизма связан с наследованием и реализуется с помощью виртуальных функций.

Рассматривается указатель на базовый класс, который может содержать как адреса объектов своего класса, так и объектов производных классов. Кроме того, механизм виртуальных функций позволяет с помощью указателя с типом базового класса обращаться к переопределенным методам в производных классах. Рассмотрим, как эту возможность можно реализовать.

Проблема доступа к методам, переопределенным в производных классах, через указатель на базовый класс решается в C++ посредством использования **виртуальных функций**.

Чтобы сделать некоторый нестатический метод виртуальным, надо в базовом классе предварить его заголовок спецификатором ***virtual***, метод становится **виртуальной функцией**. Если эту функцию переопределить в производном классе даже без спецификатора ***virtual***, в производном классе также создается **виртуальная функция**. Сигнатуры функций должны различаться только их принадлежность разным классам, а алгоритмы могут быть различными. Все сказанное похоже на механизм замещения.

Однако виртуальность этих функций (или полиморфизм) проявляется в том, что выбор требуемой из множества определенных в иерархии классов виртуальных функций с одним именем осуществляется не во время компиляции программы, а динамически, по конкретному значению указателя базового типа, с помощью которого и вызывается функция.

Какая функция будет вызываться зависит от типа указателя и типа того объекта, адрес которого присвоен указателю.

Виртуальность функций проявляется только в том случае, если она вызывается через **указатель или ссылку на базовый класс**.

Указатель на базовый класс может принимать конкретные значения.

Если значение указателя к моменту вызова функции есть адрес объекта базового класса, вызывается вариант функции из базового класса.

Если этот указатель имеет значение адреса объект производного класса (фактически указывает на данные базового класса в объекте производного класса), то вызывается вариант функции из производного класса.

Рассмотрим вышесказанное на примере.

```
class A {
public:
 virtual void F1 ();
 virtual int F2 (char*); };
class B : public A {
public:
 virtual void F1 ();
 virtual int F2 (char*);};
class C : public A {
public:
 void F1 ();
 int F2 (char*); };
int main () {
 A * ap = new A;
 B * bp = new B;
 C * cp = new C;
 ap-> F1 (); // вызов функции базового класса A
```

```

ap = bp ;
ap-> F1 () ; // вызов замещенной функции класса B
ap = cp ;
ap-> F1 () ; // вызов замещенной функции класса C
return 0 ;
}

```

Вызов виртуальной функции через указатель на базовый класс позволяет в зависимости *от значения* этого указателя (не от типа этого указателя, а от значения!) вызывать варианты этой функции из разных классов.

Если бы функции **F1** были бы обычными компонентными функциями, то при всех трех вызовах вызывалась бы всегда функция базового класса. То есть вызов через указатель не виртуальных компонентных функций зависит от *типа указателя*. Если указатель на базовый класс, то вызывается базовая функция, если указатель на производный класс, то вызывается функция производного класса.

Если из текста программы однозначно следует, какая функция вызывается, компилятор включает в текст оператор *call* с именем функции, компоновщик заменяет имя на фактический адрес функции. Такой процесс вызова функции носит название *раннего связывания*.

При использовании виртуальных функций, выбор требуемой функции из числа возможных переносится на время выполнения программы. Такой процесс носит название *позднего связывания* или *динамического связывания*. Это означает, что на этапе компиляции компилятор не определяет какой из методов должен быть вызван, а передает ответственность программе, которая принимает решение на этапе выполнения, когда уже точно известно, каков тип объекта, на который указывает наш указатель. Все сказанное относится также к вызову методов *по ссылке* на базовый класс.

Для реализации динамического связывания компилятор создает *таблицу виртуальных функций*. Обычно таблица виртуальных функций – это массив или связанный список с указателями на виртуальные функции (*virtual table pointer – vptr*). Такой массив или список компилятор автоматически включает в реализацию каждого класса, в котором определены или унаследованы виртуальные функции, то есть каждый класс имеет собственную таблицу, элементы такой таблицы адресуют коды виртуальных функций именно этого класса. Каждый объект такого класса включает дополнительный член - указатель (*pointer*) на таблицу виртуальных функций класса.

Механизм идентификации типа во время выполнения программы (RTTI) позволяет определять, на какой тип в текущий момент времени ссылается указатель.

### Работа с виртуальными функциями.

Механизм виртуальных функций можно использовать как для вертикальной цепочки классов, производных друг от друга так и для

горизонтальной группы классов, находящихся на одном уровне иерархии и производных от одного и того же базового класса.

Рассмотрим последний случай.

```
class A {
public :
virtual void Func ();};
class B : public A {
virtual void Func();};
class C : public A {
virtual void Func();};
void A:: Func() {cout<<"A"; }
void B:: Func() {cout<<"B"; }
void C:: Func() {cout<<"C"; }
int main () {
A* ap; // указатель на базовый класс A
B* bp = new B;
C* cp = new C;
ap = bp;
ap->Func(); // вызывается функция B:: Func()
ap = cp ;
ap->Func(); // вызывается функция C:: Func()
return 0;
}
```

Рассмотрим случай, когда некоторая внешняя функция в качестве аргумента имеет указатель на базовый класс. В теле функции через указатель вызывается виртуальный метод **Func()** того класса, адрес объекта которого будет передан в функцию при вызове.

```
void F (A* a) { a->Func(); }
int main () {
A* ap ;
B* bp = new B;
C* cp = new C;
ap = bp;
F(ap); // вызывается B:: Func()
ap = cp;
F(ap); // вызывается C:: Func()
return 0;}
```

В качестве аргументом функции **F()** можно использовать и ссылку на базовый класс:

```
void F (A& a) { a . Func(); }
int main () {
A* ap ;
B* bp = new B;
```



```

C* cp = new C;
ap = bp;
F(*ap); // вызывается B:: Func()
ap = cp;
F(*ap); // вызывается C:: Func()
return 0;
}

```

Можно использовать механизм виртуальных функций, если указатели на базовый класс инициализировать адресами объектов производных классов:

```

int main () {
A* bp = new B;
A* cp = new C;
bp->Func(); // вызывается функция B:: Func()
cp->Func(); // вызывается функция C:: Func()
cp=bp;
cp->Func(); // вызывается функция B:: Func()
return 0;}

```

Механизм виртуального вызова может быть подавлен с помощью использования полного квалифицированного имени.

Через указатель на базовый класс нельзя обращаться к не виртуальным компонентной функции производного класса.

Механизм виртуальных функций снимает этот запрет и позволяет с помощью указателя на базовый класс обращаться к виртуальным функциям производных классов (после присваивания этому указателю значения указателя на соответствующий объект производного класса).

Классы, включающие виртуальные функции, играют особую роль в объектно-ориентированном программировании. Они носят название **полиморфных классов**.

**Отметим важный момент. Если базовый класс содержит хотя бы один виртуальный метод, то рекомендуется всегда снабжать этот класс виртуальным деструктором, даже если он ничего не делает. Наличие такого виртуального деструктора предотвратит некорректное удаление объектов производного класса, адресуемых через указатель на базовый класс, так как в противном случае деструктор производного класса вызван не будет [1].**

## 5.2. Пустая и чистая виртуальные функции. Абстрактный класс

Реально в конкретных задачах вызываются лишь функции производных классов. "Исходная" виртуальная функция базового класса часто нужна только для того, чтобы в производных классах было, что замещать.

Содержимое базовой функции в этом случае не имеет значения и может быть пустым:

```

class A {

```

```
public :
virtual void Func () {}
};
```

Также можно объявить в базовом классе *чистую виртуальную функцию*, которая вводится с помощью такого определения:

```
virtual int имя (спецификация параметров) = 0;
```

Это некоторая абстракция и такую функцию обязательно надо замещать в производных классах, в которых она и наполнится разумным содержанием.

Объявление такой функции в базовом классе носит формальный характер для указания на виртуальность функций с данным именем.

Класс, в котором есть хотя бы одна чистая виртуальная функция, называется *абстрактным классом*.

Перечислим основные свойства абстрактного класса:

- Невозможно создать самостоятельных объектов абстрактного класса.
- Абстрактный класс может использоваться только в качестве базового класса.
- Если в производном классе от абстрактного базового класса происходит замещение чистой виртуальной функции, то производный класс не является абстрактным.

Если замещение не производится, то производный класс также является абстрактным. Пример:

```
class A { // абстрактный класс
public:
virtual int Func (char*) =0;
void F ();};
class B: public A { // B - не абстрактный класс
int Func (char*);};
class C: public A { // C – также абстрактный класс
void F ();};
```

- Абстрактные классы предназначены для представления общих понятий, которые предстоит конкретизировать. На базе общих понятий строятся частные производные классы для описания конкретных объектов.

- Абстрактный класс может иметь поля данных, а также методы, отличные от чисто виртуальных, и как всякий класс может иметь явно определенный конструктор.

- Конструктор абстрактного класса не может использоваться для создания объектов, но может использоваться при наследовании. С его помощью инициализируются поля данных абстрактного базового класса, входящие в объект производного класса.

- Указатель на абстрактный класс может использоваться в качестве формального параметра. Соответствующий фактический параметр должен иметь тип указателя на объекты производного (уже не абстрактного) класса.

- Абстрактный класс может быть производным как от абстрактного, так и от обычного классов.

Рассмотрим в качестве примера абстрактного класса класс "фигура на плоскости", производный от не абстрактного класса "точка на плоскости" [3].

Определим в файле *point.h* базовый класс:

```
class point {
protected:
double x, y;
public:
point (double x1=0.0, double y1=0.0): x(x1), y(y1) {}
void move (double x1=0.0, double y1=0.0) {x=x1; y=y1;}
};
```

Определим в файле *figure.h* абстрактный производный класс:

```
#include "point.h" // включаем определение базового класса
#include <string>
class figure :public point {
protected:
double dx, dy; // "габариты" фигуры
public:
figure (double x1=0.0, double y1=0.0, double dx1=0.0, double dy1=0.0):
point (x1, y1), dx(dx1), dy(dy1) {}
//изменить на заданную величину габариты
void grow (double k) {dx +=k; dy +=k;}
//вычислить площадь еще неизвестной фигуры
virtual double area() = 0; //чистая виртуальная функция
virtual string className() = 0; //чистая виртуальная функция
friend ostream & operator << (ostream & out, figure&);
};
ostream & operator << (ostream & out, figure &fig)
{out<<fig.className() << ":\t center: x= " << fig.x<< ",\ty= " << fig.y;
out<< ";\n\t dx= " << fig.dx<< ",\tdy= " << fig.dy;
<< ";\t area = " << fig.area();
return out;}
```

Определим производные классы "эллипс" и "прямоугольник":

```
//файл realfigures.h
#include "figure.h"
struct ellipse: public figure {
ellipse (double x1=0.0, double y1=0.0, double dx1=0.0, double dy1=0.0):
figure(x1, y1, dx1, dy1){}
virtual double aarea ()
{return ((dx/2)*(dy/2)*3/14159);}
string className() {return string("ellipse");}
};
```

```

struct square: public figure {
square (double x1=0.0, double y1=0.0, double dx1=0.0, double dy1=0.0):
figure(x1, y1, dx1, dy1){}
virtual double aerea ()
{return dx*dy;}
string className() {return string("square");}
};

```

Программа:

```

#include <iostream>
using namespace std;
#include "realfigures.h"
int main() {
ellipse A(10.0, 8.0, 20.0, 20.0), B;
cout<<"Object A:\n " <<A<<endl;
A.move(5.0,5.0);
cout<<" A.move(5.0,5.0):\n" <<A<<endl;
A.grow(-18.0);
cout<<" A.grow(-18.0):\n" <<A<<endl;
B=A;
cout<<"Object B:\n " <<B<<endl;
square C (1.0, 3.0, 5.0, 6.0), D;
cout<<"Object C:\n " <<C<<endl;
C.grow(-5.0);
cout<<" C.grow(-5.0):\n" <<C<<endl;
D. move (-10, 50);
cout<<" D.move(-10, 50):\n" <<D<<endl;
D.grow(10.0);
cout<<"Object D:\n " <<D<<endl;
return 0;
}

```

Результат выполнения программы:

```

Object A:
ellipse: centre: x=10, y=8
dx=20 dy=20 area=314.159
A.move(5.0,5.0):
ellipse: centre: x=5, y=5
dx=20 dy=20 area=314.159
Object B:
ellipse: centre: x=5, y=5
dx=2 dy=2 area=3.14159
Object C:
square: centre: x=1, y=3
dx=5 dy=6 area=30

```

**C.grow(-5.0):**

**square:     centre: x=1,     y=3**  
**dx=0     dy=1     area=0**

**D.move(-10, 50):**

**square:     centre: x=-10,     y=50**  
**dx=0     dy=0     area=0**

**Object D:**

**square:     centre: x=-10,     y=50**  
**dx=10     dy=10     area=100**

*Присваивание при наследовании с виртуальными функциями рассмотрим на примере:*

```
class Bas { //определение базового класса
int k;
public:
Bas(int k1=0): k(k1){ } //конструктор с параметром с умолчанием
//перезгрузка операции вывода
friend ostream & operator<< (ostream&out, const Bas&b);
};
ostream & operator<< (ostream&out, const Bas&b)
{out<<"k= "<<b.k<<endl; return out; }
class Dir : Bas { //определение производного класса
double z;
public:
Dir (int k1=0, double z1=0): Bas(k1), z(z1) { } //конструктор
//перезгрузка операции вывода
friend ostream & operator<< (ostream&out, const Dir&b);
};
ostream & operator << (ostream&out, const Dir&b)
{ out<<"Bas:: ";
/*при вызове операции – функции базового класса проводится операция
преобразования типа ссылки на объект производного класса к ссылке на
объект базового класса, повышающее преобразование, так как в иерархии
классов базовый класс выше производного*/
out<< dynamic_cast<const Bas*>(d);
out<<"Dir::z= "<<d.z<<endl; return out; }
int main()
Dir one(15,4.0),two;
Bas *p1=&one,*p2=&two;
*p1=*p2; // присваивания только полей базового класса!
cout<<two<<endl
two=one; //присваивание полей и производно класса
cout<<two<<endl;
```

```

}
Bas::k=15
Dir::z=0
Bas::k=15
Dir::k=4

```

В первом случае автоматически вызывался метод для базового класса:

```
Bas& Bas::operator=(const Bas&)
```

Во втором случае автоматически вызывался метод для производного класса:

```
Dir& Dir::operator=(const Dir&)
```

Чтобы с помощью указателей на базовый класс можно было обратиться к методу присваивания производного класса *Dir*, надо явно определить в базовом классе функцию-перегрузку присваивания и определить ее как виртуальную.

Но в произвольном классе функция перегрузки должна иметь те же параметры.

Решение может быть таким. В базовом классе определить:

```

virtual Bas& operator=(const Bas&b)
{ k=b.k;
 return *this;
}

```

Функция та же, что и представляется классу по умолчанию, кроме виртуальности.

В производном классе следует определить, например, такую функцию:

```

virtual Dir& operator=(const Bas&d)
{ this-> Bas::operator=(d);
 const Dir& r= dynamic_cast<const Dir&>(d); //формируем ссылку на объект
 //производного класса, используя понижающее преобразование типа
 z=r.z;
 return *this;
}

```

В этом случае уже при первом присваивании *\*p1=\*p2*; произойдет правильное присваивание объектов и базовой части и производной.

Деструкторы при наследовании с виртуальными функциями рассмотрим на примере:

```

class Bas {
public:
 ~Bas(){cout<<"~Bas"<<endl;}
};
class Dir:public Bas {
int* Arr;
public:
 Dir():Arr(new int[10]){}
 ~Dir(){

```

```

delete[]Arr;
cout<<"~Dir"<<endl;}
};
int main()
{Bas*b1=new Dir;
delete b1;
cout<<"New object"<<endl;
Dir obj;
return 0;}

```

Результат:

```

~Bas
New object
~Dir
~Bas

```

При уничтожении объекта операцией *delete* выполнен только деструктор базового класса. В памяти остался беспризорный массив из 10 элементов. Деструктор не виртуальный и через указатель на базовый класс будет вызван деструктор базового класса.

Рекомендуется деструктор базового класса определять, как виртуальный и при необходимости переопределять его в производном классе.

```
virtual ~Bas(){cout<<"~Bas"<<endl;}
```

и тогда:

```

~Dir
~Bas
New object
~Dir
~Bas

```

## Раздел 6 Исключения

### 6.1. Общие сведения об исключениях

**Исключительная ситуация**, или **исключение** — это возникновение во время выполнения программы непредвиденного или аварийного события, которое делает дальнейшее выполнение программы в соответствии с базовым алгоритмом невозможными или бессмысленными.

Есть целый ряд встроенных ситуаций, таких как деление на ноль, достижение конца файла, переполнение в арифметических операциях, обращение по несуществующему адресу памяти и т. п. Обычно эти события приводят к завершению программы с системным сообщением об ошибке. C++ дает программисту возможность восстанавливать программу и продолжать ее выполнение. Делается это с помощью исключений. Исключительные ситуации, можно разделить на два основных типа: синхронные и асинхронные.

**Синхронные исключения** могут возникнуть только в определённых, заранее известных точках программы. Так, например, ошибка чтения файла или нехватка памяти — синхронные исключения, так как возникают они только в операции чтения из файла или в операции выделения памяти соответственно.

**Асинхронные исключения** могут возникать в любой момент времени и не зависят от того, какую конкретно инструкцию программы выполняет система. Типичные примеры таких исключений: аварийный отказ питания.

**Исключения C++** не поддерживают обработку асинхронных событий, таких, как обработку аппаратных прерываний, например, нажатие клавиш **Ctrl+C**. Механизм исключений предназначен только для событий, которые происходят в результате работы самой программы.

**Исключения позволяют логически разделить вычислительный процесс на две части — обнаружение аварийной ситуации и ее обработку.**

Это важно не только для лучшей структуризации программы.

Главной причиной является то, что функция, обнаружившая ошибку, может не знать, что предпринимать для ее исправления, а использующий эту функцию код может знать, что делать, но не уметь определить место возникновения.

Кроме того, для передачи информации об ошибке в вызывающую функцию не требуется применять возвращаемое значение, параметры или глобальные переменные, поэтому интерфейс функций не раздувается. Это важно, например, для конструкторов, которые не могут возвращать значение.

## 6.2. Обработка исключительных ситуаций

Механизм обработки исключений является общим средством управления программой. С его помощью можно обрабатывать не только аварийные, но и любые другие ситуации, возникшие в результате выполнения программы.

*Но именно для выхода из тупиковых положений служит механизм исключений – средство, позволяющее отделить выявление особой ситуации от обработки информации о ней [1].*

### Механизм обработки исключений (МОИ)

Для реализации МОИ в язык C++ введены следующие ключевые слова: **try** (контролировать, пытаться, пробовать), **catch** (ловить, перехватывать), **throw** (бросать, генерировать, посылать).

Общая схема посылки и обработки исключений:

```
try{ операторы
throw выражение_1;
операторы
throw выражение_2
операторы
}
catch (спецификация_исключения_1)
```



```
{операторы_обработки_исключения_1}
catch (спецификация_исключения_2)
{операторы_обработки_исключения_2}
```

Место, в котором может произойти ошибка, должно входить в контролируемый блок — составной оператор, перед которым записано ключевое слово **try**:

```
try {операторы}
```

Среди операторов, заключенных в фигурные скобки могут быть любые операторы языка, описания объектов и функций, определения локальных переменных. Кроме того, в блок контроля за исключениями помещаются специальные операторы, генерирующие исключения и имеющие формат:

```
throw выражение;
```

С помощью выражения формируется специальный объект, называемый исключением. Все исключения создаются как временные объекты, а тип и значение каждого исключения определяется формирующим его выражением.

Оператор **throw** выполняет посылку исключения, то есть передает управление и пересылает исключение непосредственно за блок контроля.

В этом месте обязательно располагается ловушка или обработчик исключения, ловушек может быть несколько (сколько исключений).

**Обработчики исключений** должны располагаться непосредственно за **try-блоком**. Они начинаются с ключевого слова **catch**, за которым в скобках следует тип обрабатываемого исключения. Обработчик имеет формат:

```
catch (спецификация_исключения)
{операторы_обработки_исключения}
```

Можно записать один или несколько обработчиков в соответствии с типами обрабатываемых исключений.

Внешне и функционально обработчик исключений похож на определение функции с одним параметром — типом исключения, не возвращающей значения.

Когда за блоком контроля размещены несколько ловушек, то они должны отличаться друг от друга.

Существует три формы записи:

```
catch(тип_исключения имя) { /* тело обработчика */ }
catch(тип_исключения){ /* тело обработчика */ }
catch(...) { /* тело обработчика */ }
```

Первая форма применяется, когда имя параметра используется в теле обработчика для выполнения каких-либо действий — например, вывода информации об исключении.

Вторая форма не предполагает использования информации об исключении, играет роль только его тип.

Многоточие обозначает, что обработчик перехватывает все исключения. Так как обработчики просматриваются в том порядке, в котором они записаны, обработчик третьего типа следует помещать после всех остальных.

Пример:

```
catch(int i){ // Обработка исключений типа int}
catch(const char *){ // Обработка исключений типа const char*}
catch(Overflow){ // Обработка исключений класса Overflow}
catch(...){ // Обработка всех необслуженных исключений}
```

После обработки исключения управление передается первому оператору, находящемуся непосредственно за обработчиками исключений.

Туда же, минуя код всех обработчиков, передается управление, если исключение в *try-блоке* не было сгенерировано.

### Перехват исключений

Когда с помощью выражения в операторе **throw** генерируется исключение в контролируемом блоке, функции исполнительной библиотеки C++ выполняют следующие действия:

- 1) создают копию параметра **throw** в виде статического объекта, который существует до тех пор, пока исключение не будет обработано;
- 2) осуществляют поиск подходящего обработчика; одновременно, вызываются деструкторы локальных объектов, выходящих из области действия;
- 3) передают объект исключения и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.

Обработчик считается найденным, если тип выражения после **throw**:

- тот же, что и указанный в параметре **catch**, параметр может быть записан в форме **T**, **const T**, **T&** или **const T&**, где **T**— тип исключения;
- является производным от указанного в параметре **catch** (если наследование производилось с ключом доступа **public**);
- является указателем, который может быть преобразован по стандартным правилам преобразования указателей к типу указателя в параметре **catch**.

Из вышеизложенного следует, что обработчики производных классов следует размещать до обработчиков базовых, поскольку в противном случае им никогда не будет передано управление. Обработчик указателя типа **void** автоматически скрывает указатель любого другого типа, поэтому его также следует размещать после обработчиков указателей конкретного типа.

Если исключение послано, но соответствующий обработчик не найден, то вызывается специальная функция **terminate ()**, и тем самым завершается программа.

Оператор, формирующий исключение, может иметь две формы:

**throw** *выражение*;

**throw**;

В первом случае исключение формируется как статический объект, значение и тип которого определяются выражением. Копия этого объекта передается за блок контроля и существует пока исключение не будет полностью обработано.

Во втором случае – оператор **throw** используется только в обработчике исключений, в том случае, когда существует вложение блоков контроля за исключениями. Цель этого оператора – ретрансляция исключения во внешний блок контроля. Если обработчик не настроен на обработку исключения, он может оператором **throw** направить его дальше (Б. Страуструп [2]).

Рассмотрим пример применения механизма исключений.

```
#include <fstream>
using namespace std;
class Hello{// Класс, информирующий о создании и уничтожении объектов
public:
Hello(){cout << "Hello!" << endl;}
~Hello(){cout << "Bye!" << endl;}};
void f1(){
ifstream ifs("NAME"); // Открываем файл для чтения
if (!ifs){ cout << "Генерируем исключение" << endl;
throw "Ошибка при открытии файла";}
void f2(){
Hello H; // Создается локальный объект
f1(); // Вызывается функцию, генерирующую исключение
}
int main () {
try{ cout << "Входим в try-блок" << endl;
f2();
cout << "Выходим из try-блока" << endl;}
catch (int i){ cout << "Вызван обработчик int, исключение - " << i << endl;
return -1;}
catch(const char * p){ cout << "Вызван обработчик const char*, исключение - "
<< p << endl;
return -1;}
catch(...){ cout << "Вызван обработчик всех исключений" << endl;
return -1;
}
return 0; // Все обошлось благополучно
}
```

Результаты выполнения программы, если файл не удалось открыть:

**Входим в try-блок**

**Hello!**

**Генерируем исключение**

**Bye!**

**Вызван обработчик const char\*, исключение - Ошибка при открытии файла**

Обратите внимание, что после порождения исключения был вызван деструктор локального объекта, хотя управление из функции **f1** было передано

обработчику, находящемуся в функции *main*. Сообщение «*Выходим из try-блока*» не было выведено.

Таким образом, механизм исключений позволяет корректно уничтожать объекты при возникновении ошибочных ситуаций.

*Как уже упоминалось, исключение может быть, как стандартного, так и определенного пользователем типа.*

При этом нет необходимости определять этот тип глобально — достаточно, чтобы он был известен в точке порождения исключения и в точке его обработки. Класс для представления исключения можно описать внутри класса, при работе с которым оно может возникать. Конструктор копирования этого класса должен быть объявлен как *public*, поскольку иначе будет невозможно создать копию объекта при генерации исключения.

В качестве примера рассмотрим применения МОИ при определении наибольшего общего делителя (НОД) двух целых чисел ( $x$ ,  $y$ ).

Классический алгоритм Евклида:

- если  $x=y$ , то ответ:  $\text{НОД}=x$ ;
- $x < y$ , то  $y$  заменяется на  $y-x$ ;
- $x > y$ , то  $x$  заменяется на  $x-y$ .

Алгоритм применим при условиях:

- оба числа неотрицательны;
- оба числа отличны от нуля.

В программе будет использован класс *string* стандартной библиотеки, описание которого находится в заголовочном файле *<string.h>*. Программа [5]:

```
#include <iostream>
#include <string>
using namespace std;
struct data{ int n, m;
string s;
data(int x, int y, string str): // конструктор с параметрами
n(x),m(y), s(str){ } // и с инициализатором
};
int GCD (int x, int y) { //определение функции
if (x==0||y==0) throw (data (x, y, "zero! ");
if(x<0) throw data(x, y, "Negative parameter 1. ");
if(y<0) throw data(x, y, "Negative parameter 2. ");
while(x!=y) {
if(x>y) x= x-y;
else y= y-x;}
return x;}
int main () {
try{
cout<<" GCD(66,44)= "<< GCD(66,44)<<endl;
cout<<" GCD(0, 7)= "<< GCD(0, 7)<<endl;
```

```
cout<<" GCD(-12, 8)= "<< GCD(-12, 8)<<endl;}
catch(data d){
serr << d.s << " x= " << d.n << ", y= " << d.m;}
return 0;}
```

Результат:

```
GCD(66,44)= 22
zero! x=0, y=7
```

Третий вызов **GCD(-12,8)** не будет выполнен.

Исключения в программе определены как объекты специального класса **data**. Объекты класса **data** формируются внутри одной функции, но доступны внутри другой.

Это особое свойство исключений. Они создаются как временные статические объекты в одном блоке, но доступны в другом.

Что касается определения класса объектов исключений, то следует отметить, что оно не обязательно размещается в глобальном пространстве имен, как класс **data**. Следовательно, при спецификации параметра обработчика (ловушки) исключений для обозначения типа параметра должна использоваться форма: **имя\_пространства\_имен :: имя\_класса**.

Рассмотрим еще пример применения механизма исключений. Используем опять для типов исключений пользовательские классы и рассмотрим случай, когда в обработчике исключений не будет значения, а только тип и случай, когда параметр вообще отсутствует. Ниже представлен листинг программы [3].

```
#include <iostream>
using namespace std;
class zero_divide{ }; //класс объектов исключений
class overflow{ }; // класс объектов исключений
double div (double n, double d) { //определение функции деления
if (d==0.0&& n==0.0) throw 0.0;
if(d==0) throw zero_divide() ;
double b=n/d;
if(b>1e+30) throw overflow ();
return b;}
double x=1e-20, z=1e+20, w=0.0;
void RR() { //применение функции деления
try { w=div(4,w);
z=div(z, x);
w=div(0.0,0.0); } // конец контрольного блока try
catch (overflow){serr<<" overFlow "<<endl; z=1e+30; x=1.0;}
catch(zero_divide){serr <<" zeroDivide "<<endl; w=1.0;}
catch(...){ serr << " Indeterminacy "<<endl;}
} // конец функции RR
int main () {RR();RR();RR();
return 0;
```

```
}
```

Результат:

***zeroDivide***

***overflow***

***Indeterminacy***

При первом обращении к функции ***RR ()*** функция ***div ()*** вызывается один раз. Сразу же возникает исключительная ситуация и посылается исключение типа ***zero\_divide***, обработчик выводит первое сообщение.

При втором вызове ***RR ()*** функция ***div ()*** вызывается уже два раза первый вызов проходит, а при втором возникает исключительная ситуация и посылается исключение уже типа ***overflow***, и соответствующий обработчик выводит второе сообщение.

При третьем обращении к функции ***RR()*** функция ***div()*** вызывается три раза. Для последнего случая не был определен тип, но ситуация исключительная, поэтому обработку проводит обработчик без параметров для всех случаев исключений.

### Список исключений функции

В заголовке функции можно задать список исключений, которые она может прямо или косвенно породить.

Типы исключений перечисляются в скобках через запятую после ключевого слова ***throw***, расположенного за списком параметров функции, например:

```
void f1() throw (int, const char*) { /* Тело функции */ }
```

```
void f2() throw (myclass*) { /* Тело функции */ }
```

Функция ***f1*** должна генерировать исключения только типов ***int*** и ***const char\****.

Функция ***f2*** должна генерировать только исключения типа указателя на класс ***myclass*** или производных от него классов.

Если ключевое слово ***throw*** не указано, функция может генерировать любое исключение.

Пустой список означает, что функция не должна порождать исключений:

```
void f() throw () { // Тело функции, не порождающей исключений }
```

Если внутри функции, такой как ***f()*** порождается исключение, оно должно быть обработано внутри нее.

Если функция создает исключение отличное от указанных в ее спецификации исключений, то управление передается специальной функции ***unexpected()***, которая вызывает функцию ***terminate ()***, и программа завершается.

## 6.3. Исключения в конструкторах и деструкторах

Язык C++ не позволяет возвращать значение из конструктора и деструктора. Механизм исключений дает возможность сообщить об ошибке, возникшей в конструкторе или деструкторе объекта.

Для иллюстрации создадим класс *Vector*, в котором ограничивается количество запрашиваемой памяти:

```
class Vector {
public :
class Size { }; // Класс исключения
enum {max = 32000}; // Максимальная длина вектора
Vector (int n) {if (n<0 || n>max) throw Size (); ...} // Конструктор
```

При использовании класса *Vector* можно предусмотреть перехват исключений типа *Size*:

```
try {Vector *p = new Vector(i);}
catch (Vector::Size){ // Обработка ошибки размера вектора }
```

Обрабатывать исключения, возникающие в конструкторах, можно двумя способами. Во-первых, можно создавать блок контроля за исключениями и набор обработчиков в том месте, где вызывается конструктор, то есть создаются объекты класса. Во-вторых, введена специальная форма генерации и обработки исключений непосредственно в конструкторах.

Внешнее определение конструктора может выглядеть так:

```
имя_класса::имя_класса (спецификация_параметров)
try:список_инициализаторов
{операторы_тела_конструктора}
последовательность_обработчиков_исключений
```

Обработчики перехватывают исключения, возникшие при инициализации и при выполнении операторов тела конструктора.

В следующей программе [5] рассматриваются оба способа.

Определим класс точек на плоскости с счетчиком их количества.

```
#include <iostream>
#include <string>
using namespace std;
#define print(X) cout<<#X"= "<<X<<endl;
class point{ double x, y;
static int N;
public:
point(double xn=0.0, double yn=0.0;)
static int& count(){return N;} ;}
int point::N=0;
point ::point(double xn=0.0, double yn=0.0) try: x(xn), y(yn) {
N++;
if(N==1) throw "The begin! ";
if(N>2) throw string "The end! ";
}
catch (const char*ch)
{cout<<ch<<endl;}
int main(){
```

```

try{ print(point::count());
point A(0.0,1.0);
print(A.count());
point B;
print(point::count());
point C;
print(point::count());
point D(1.0,2.0);
print(D.count());}
catch (const string ch)
{cout<<ch<<endl;}
return 0;}

```

Результат:

```

point::count()=0
The begin
A.count()=1
point::count()=2
The end

```

Исключение типа *const char\** послано при первом обращении к конструктору при создании первого объекта. Оно обработано в конструкторе, выводится сообщение *"The begin!"*.

Исключение типа *string* посылается, когда число объектов превысит 2. Оно перехватывается в основной программе (*"The end! "*).

## ЛИТЕРАТУРА

1. Страуструп Б. Язык программирования С++. Специальное издание. - Издательство: "Бином", 2011, ISBN: 978-5-9518-0425-9, 1136 с.
2. Страуструп Б. Дизайн и эволюция языка С++. Издательство: "ДМК-Пресс", 2014, ISBN: 978-5-94074-994, 488 с.
3. Подбельский В.В. Стандартный Си++: учебное пособие – М: Финансы и статистика, 2008, 688 с.
4. Подбельский В.В. Язык Си++. – М: Финансы и статистика, 2000, 560 с.
5. Страуструп Б. Программирование. Принципы и практика использования С++. Издательство: "Вильямс", ISBN: 978-5-8459-1621-1 (рус.), 2011, 1206 с.



## СОДЕРЖАНИЕ

|                                                             |    |
|-------------------------------------------------------------|----|
| Раздел 1. Структуры и объединения                           | 3  |
| 1.1. Структура как совокупность данных                      | 3  |
| 1.2. Объединения разнотипных данных                         | 8  |
| 1.3. Битовые поля структур и объединений                    | 11 |
| Раздел 2. Класс как абстрактный тип                         | 13 |
| 2.1. Класс – производный структурированный тип              | 13 |
| 2.2. Статический компонент класса                           | 16 |
| 2.3. Доступ к членам класса                                 | 18 |
| 2.4. Дружбы классов                                         | 20 |
| 2.5. Конструкторы и деструкторы                             | 23 |
| 2.6. Указатели на поля данных и на методы класса            | 34 |
| 2.7. Указатель this                                         | 35 |
| 2.8. Перегрузка функций                                     | 38 |
| Раздел 3. Перегрузка операций                               | 39 |
| 3.1. Расширение действий (перегрузка) стандартных операций  | 39 |
| 3.2. Преобразование типов в классах пользователя            | 43 |
| 3.3. Перегрузка операции присваивания                       | 47 |
| Раздел 4. Включение и наследование классов                  | 49 |
| 4.1. Отношения включения                                    | 49 |
| 4.2. Отношения наследование                                 | 51 |
| 4.3. Множественное наследование. Виртуальные базовые классы | 58 |
| 4.4. Методы при наследовании классов                        | 61 |
| Раздел 5. Виртуальные функции и абстрактные классы          | 64 |
| 5.1. Виртуальные функции                                    | 64 |
| 5.2. Пустая и чистая виртуальные функции. Абстрактный класс | 68 |
| Раздел 6. Исключения                                        | 74 |
| 6.1. Общие сведения об исключениях                          | 74 |
| 6.2. Обработка исключительных ситуаций                      | 75 |
| 6.3. Исключения в конструкторах и деструкторах              | 81 |
| ЛИТЕРАТУРА                                                  | 83 |