

## Использование препроцессора.

Препроцессором называется первая фаза компилятора. Препроцессор входит в состав системы программирования C++ и обеспечивает обработку исходного текста программы перед ее компиляцией. В ходе препроцессорной обработки программы реализуются следующие действия:

1. выполняется подстановка макрорасширений вместо имен макросов и их аргументов.
2. в исходный текст программы включаются файлы, содержащие вспомогательную информацию (описания объектов, классов, прототипы функций и т. п.)
3. Выполняется условная компиляция программы - определение окончательного вида исходного текста программы путем оставления или исключения отдельных ее фрагментов.

Задание препроцессору описывается с помощью *директив препроцессора*, которые начинаются со знака #, размещаемого с первой позиции в строке, перед которым в строке могут находиться только пробельные символы.






Препроцессорная обработка позволяет сократить время разработки программ, повысить их наглядность и мобильность. В частности, при изменении характеристик аппаратно-программных средств достаточно изменить содержимое подключаемых файлов

### Директива #include

Директива #include <имя\_файла> вставляет содержимое указанного файла в ту точку исходного файла, где она записана. Включаемый файл также может содержать директивы #include. Поиск файла, если не указан полный путь, ведется в стандартных каталогах подключаемых файлов. Вместо угловых скобок могут использоваться кавычки (« ») – в этом случае поиск файла ведется в каталоге, содержащем исходный файл, а затем уже в стандартных каталогах.

Директива #include является простейшим средством обеспечения согласованности объявлений в различных файлах, она включает в низ информацию об интерфейсе из заголовочных файлов.

Заголовочные файлы обычно имеют расширение .h и могут содержать:




-  Определения типов, констант, встроенных функций, шаблонов, перечислений;
-  Объявления функций, данных, имен, шаблонов;
-  Пространства имен;
-  Директивы процессора;
-  Комментарий.

В заголовочном файле не должно быть определений функций и данных. Эти правила не являются требованиями языка, а также отражают разумный способ использования директивы.

При указании заголовочных файлов стандартной библиотеки расширения .h можно опускать. Старые версии компиляторов могут не поддерживать это свежее требование стандарта

### Директива #define

Директива #define определяет подстановку в тексте программы. Она используется для определения:

-  Символических констант:  
#define имя текст\_подстановки  
(все вхождения имени заменяются на текст подстановки);
-  Макросов, которые выглядят как функции, но реализуются подстановкой их текста в текст программы:  
#define имя( параметры ) текст\_подстановки
-  Символов, управляющих условной компиляцией. Они используются вместе с директивами #ifdef и #ifndef. Формат:  
#define имя

Примеры:

```
#define VERSION 1
#define VASIA "Василий Иванович"
#define MAX(x,y) ((x)>(y)?(x):(y))
#define MUX
```

Имена рекомендуется записывать прописными буквами, чтобы зрительно отличать их от имен переменных и функций. Параметры макроса используются при макроподстановке, например, если в тексте программы используется вызов макроса `y = MAX(sum1, sum2);`, он будет заменен на

```
y = ((sum1,>(sum2)?(sum1):(sum2));
```

Отсутствие круглых скобок может привести к неправильному порядку вычисления, поскольку препроцессор не оценивает вставляемый текст с точки зрения синтаксиса. Например, если к макросу `#define sqr(x) (x*x)` обратиться как `sqr(y+1)`, в результате подстановки получится выражение `(y+1*y+1)`.

Макросы и символические константы установлены из языка C, при написании программ на C++ их следует избегать. Вместо символических констант предпочтительнее использовать `const` или `enum`, а вместо макросов – встроенные функции и шаблоны.

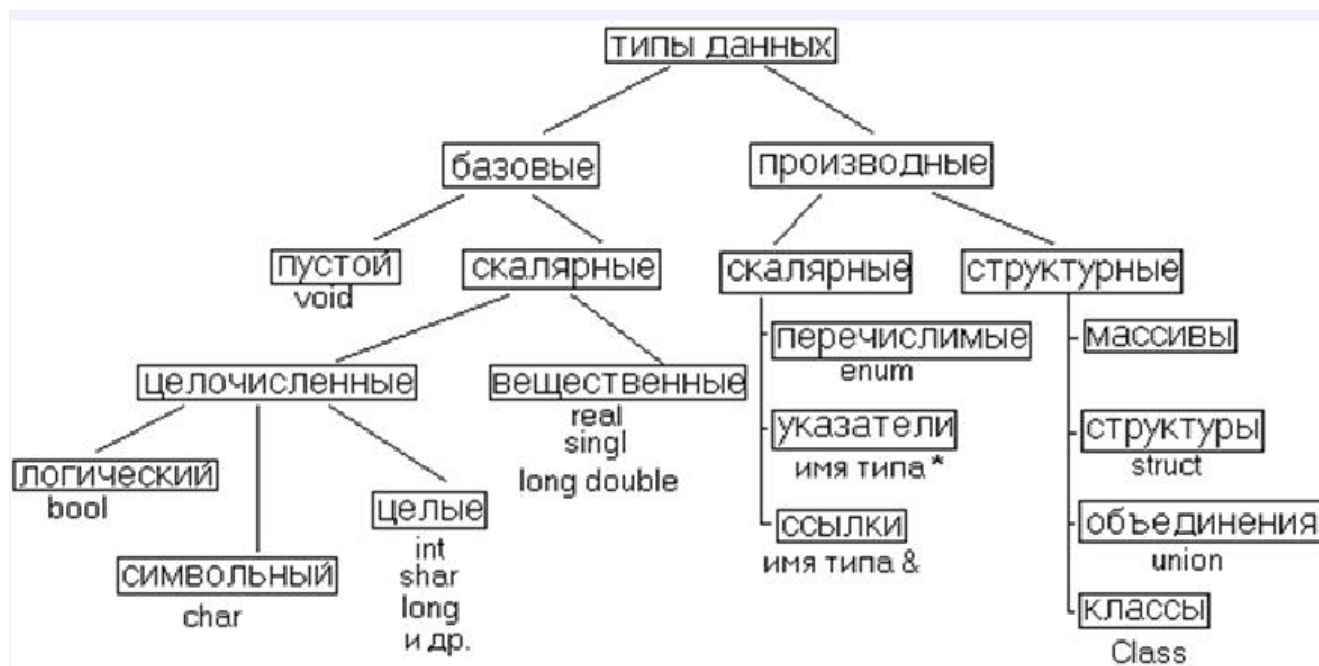
Таблица 1. Спецификации формата для функций ввода-вывода C.

Специфика	Пояснение
<code>c</code>	аргумент рассматривается как отдельный символ
<code>d, i</code>	аргумент преобразуется к десятичному виду
<code>e, E</code>	аргумент, рассматриваемый как переменная типа <i>float</i> или <i>double</i> , преобразуется в десятичную форму в виде <code>[-]m.nnnnnne[+ -]xx</code> , где длина строки из <code>n</code> определяется указанной точностью. Точность по умолчанию равна 6
<code>f</code>	аргумент, рассматриваемый как переменная типа <i>float</i> или <i>double</i> , преобразуется в десятичную форму в виде <code>[-]mmm.nnnnnn</code> , где длина строки из <code>n</code> определяется указанной точностью. Точность по умолчанию равна 6
<code>g, G</code>	используется формат <code>%e</code> или <code>%f</code> , который короче; незначащие нули не печатаются
<code>o</code>	аргумент преобразуется в беззнаковую восьмеричную форму (без лидирующего нуля)
<code>p</code>	вывод указателя в шестнадцатеричном формате (эта спецификация не входит в стандарт, но она существует практически во всех реализациях)
<code>s</code>	аргумент является строкой: символы строки печатаются до тех пор, пока не будет достигнут нулевой символ или не будет напечатано количество символов, указанное в спецификации точности
<code>u</code>	аргумент преобразуется в беззнаковую десятичную форму
<code>x, X</code>	аргумент преобразуется в беззнаковую шестнадцатеричную форму (без лидирующих 0x)
<code>%</code>	выводится символ <code>%</code>

Список ключевых слов.

<code>asm</code>	<code>else</code>	<code>new</code>	<code>this</code>
<code>auto</code>	<code>enum</code>	<code>operator</code>	<code>throw</code>
<code>bool</code>	<code>explicit</code>	<code>private</code>	<code>true</code>
<code>break</code>	<code>export</code>	<code>protected</code>	<code>try</code>
<code>case</code>	<code>extern</code>	<code>public</code>	<code>typedef</code>
<code>catch</code>	<code>false</code>	<code>register</code>	<code>typeid</code>
<code>char</code>	<code>float</code>	<code>reinterpret_cast</code>	<code>typename</code>
<code>class</code>	<code>for</code>	<code>return</code>	<code>union</code>
<code>const</code>	<code>friend</code>	<code>short</code>	<code>unsigned</code>
<code>const_cast</code>	<code>goto</code>	<code>signed</code>	<code>using</code>
<code>continue</code>	<code>if</code>	<code>sizeof</code>	<code>virtual</code>
<code>default</code>	<code>inline</code>	<code>static</code>	<code>void</code>
<code>delete</code>	<code>int</code>	<code>static_cast</code>	<code>volatile</code>
<code>do</code>	<code>long</code>	<code>struct</code>	<code>wchar_t</code>
<code>double</code>	<code>mutable</code>	<code>switch</code>	<code>while</code>
<code>dynamic_cast</code>	<code>namespace</code>	<code>template</code>	

### Классификация типов данных



### Диапазоны значений базовых типов данных

<i>тип данных</i>	<i>название</i>	<i>размер, бит (16/32 разр.)</i>	<i>диапазон значений</i>
[signed] char	целый длиной не менее 8 бит	8/8	-128 .. 127
unsigned char	беззнаковый целый длиной не менее 8 бит	8/8	0 .. 255
[signed] short [int] (short)	короткий целый	16/16	-32768 .. 32767
unsigned short [int]	беззнаковый короткий целый	16/32	0 .. 65535
[signed] int	целый	16/32	-32768 .. 32767
unsigned int	беззнаковый целый	16/32	0 .. 65535
[signed] long [int] (long)	длинный целый	32/32	-2147483648 .. 2147483647
unsigned long [int]	беззнаковый длинный целый	32/32	0 .. 4294967295
float	вещественный одинарной точности	32/32	3.4E-38 .. 3.4E+38
double	вещественный двойной точности	64/64	1.7E-308 .. 1.7E+308
long double	вещественный максимальной точности	80/80	3.4E-4932 .. 1.1E+4932
enum	перечисляемый	16/16	-32768 .. 32767

Для вещественных типов в таблице приведены абсолютные величины минимальных и максимальных значений.

Минимальные и максимальные допустимые значения для целых типов зависят от реализации и приведены в заголовочном файле `<limits.h>` (`<climits>`), характеристики вещественных типов — в файле `<float.h>` (`<cfloat>`), а также в шаблоне класса `numeric_limits`.

Размер целых типов зависит от реализации, но для всех версий C++ принято:

$$\text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$$

Кроме то гарантируется во всех реализациях:

$$1 = \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short})$$

$$1 \leq \text{sizeof}(\text{bool}) \leq \text{sizeof}(\text{long})$$

Для вещественных типов выполняется соотношение:

$$\text{sizeof}(\text{float}) \leq \text{sizeof}(\text{double}) \leq \text{sizeof}(\text{long double})$$

Таблица 2. Форматы констант, соответствующие каждому типу.

Константа	Формат	Примеры
<b>Целая</b>	Десятичный: последовательность десятичных цифр, начинающаяся не с нуля, если это не число ноль  Восьмеричный: ноль, за которым следуют восьмеричные цифры (0,1,2,3,4,5,6,7)  Шестнадцатеричный: 0x или 0X, за которым следуют шестнадцатеричные цифры (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)	8, 0, 199226  01, 020, 07155  0xA, 0x1B8, 0X00FF
<b>Вещественная</b>	Десятичный: [цифры] . [цифры] (могут быть опущены целая часть, либо дробная, но не обе сразу).  Экспоненциальный: [цифры] [ . ] [цифры] {E e} [ +   - ] [цифры] (могут быть опущены целая часть, либо дробная, но не обе сразу. Если указаны обе части, символ точки обязателен)	5.7, .001, 35.  0.2E6, .11e-3, 5E10
<b>Символьная</b>	Один или два символа, заключенных в апострофы	клавиатурные: 'A', 'ю', '*', кодовые: '\n', '\a', '\t', '\?', кодовые числовые: '\0', '\012',
<b>Строковая</b>	Последовательность символов, заключенная в кавычки	"Здесь был Vasia", "\tЗначение r=\0xF5\n"

Таблица 3. Управляющие последовательности в языке C++

Изображение	Шестнадцатеричный код	Наименование
\a	7	Звуковой сигнал
\b	8	Возврат на шаг
\f	C	Перевод страницы (формата)
\n	A	Перевод строки
\r	D	Возврат каретки
\t	9	Горизонтальная табуляция
\v	B	Вертикальная табуляция
\\	5C	Обратная косая черта
\'	27	Апостроф
\"	22	Кавычка
\?	3F	Вопросительный знак
\0ddd	—	Восьмеричный код символа
\xddd	ddd	Шестнадцатеричный код символа

Таблица 4. Характеристики переменных

Модификатор выделения памяти	Место объявления переменной	Область видимости	Время жизни	Область действия
не указан (по умолчанию static)	вне функции	до конца файла	постоянное	глобальная переменная файла
не указан (по умолчанию auto)	в функции	до конца блока	временное	локальная переменная
auto	запрещено вне функции			
auto	в функции	до конца блока	временное	локальная переменная
register	запрещено вне функции			
register	в функции	до конца блока	временное	быстрая локальная переменная
static	вне функции	до конца файла	постоянное	глобальная переменная файла
static	в функции	до конца блока	постоянное	сохраняемая локальная переменная
extern	вне функции	до конца файла	постоянное	глобальная переменная программы
extern	в функции	до конца файла	постоянное	специальная глобальная переменная

### Пространство имен

Поименованные области служат для логического группирования объявлений и ограничения доступа к ним. Чем больше программа, тем более актуально использование поименованных областей. Простейшим примером применения является отделение кода, написанного одним человеком, от кода, написанного другим. При использовании единственной глобальной области видимости форматировать программу из отдельных частей очень сложно из-за возможного совпадения и конфликта имён. Использование **поименованных областей (пространства имен, namespace)** препятствует доступу к ненужным средствам.

**Пространство имен (namespace)** представляет собой некоторую объявляемую с помощью специальной конструкции область, служащую для локализации (ограничения области действия) имен переменных и объектов с целью избежания конфликтов при использовании одинаковых имен. Ситуация, когда в программе применяется несколько одинаковых идентификаторов, обычно возникает в случае использования библиотек функций и классов разных производителей. При использовании пространства имен, помимо необязательного глобального пространства имен, выделяется несколько частных пространств. В разных частных пространствах можно использовать одинаковые идентификаторы переменных, классов и объектов, которые участвуют в рамках своего частного пространства имен и имеют в нем свой смысл.

Пространство имен задается следующим образом:

```
namespace имя
{
```

```
//объявления
```

```
}
```

Здесь:

*namespace* — ключевое слово, задающее пространство имен; *имя* — любой идентификатор.

В фигурных скобках находится само пространство имен, имеющее название с заданным именем. В результате такого определения пространства все имена, входящие в данное пространство имен, будут видимы только внутри введенного пространства.

**Пример 1.** Определение и использование пространства имен.

```
#include <iostream>
namespace A
{
    int x,y;
    class M
    {
    public:
        void wodx(int c) {x=c;}
        int dostx() {return x; }
    }
    void vyvod (int k) {cout << ;}
}
void main()
{
    int z,t,y;
    y=3;
    A::y=7;
    A::M b;
    z=9;
    b.vvodb (z) ;
    A::x=z;
    cout << "\n A::x=";
    A::vyvodb(A::x);
    cout << "\n y=";
    A::vyvodb(y);
    cout << "\n A::y=" << A::y;
    t=b.dostx() ;
    cout << "\n t=" << t;
}
```

В результате работы программы на экран будут выданы следующие сообщения:

```
A::x=9
y=3
A::y=7
t=9
```

В примере переменные *x*, *y*, класс *M* и функция *vyvodb()* находятся в области видимости, определенной пространством имен *A*. Описание функции *main()* находится в глобальном пространстве имен. В глобальном пространстве имен видимы переменные *r*, *t* и *y*. Таким образом, в программе объявлены две переменные с одним именем *y*: первая видима в пространстве имен *A*, а вторая — в глобальном пространстве.

Применение имен идентификаторов внутри пространства имен производится обычным образом. При использовании имен идентификаторов другого пространства имен необходимо применять операцию *::* указания области видимости. Например, в функции *main()* оператор *A::x=z*; присваивает значение переменной *z* (из глобального пространства имен) переменной *x*, принадлежащей пространству имен *A*. Для объявления объекта *b* класса *M*, описание которого находится в пространстве имен *A*, необходимо также использовать операцию *::* указания области видимости при объявлении объекта: *A::M b*;

Существует возможность введения более одного пространства имен с одним и тем же именем, причем последующие объявления рассматриваются как расширения предыдущих.. Это позволяет разделить пространство имен на несколько файлов или на несколько частей внутри одного файла.

Например:

```
namespace L { int x;}
namespace L { float y;}
```

Здесь объявлены две части одного пространства имен *L*.

**Пример 2.**

```
namespace demo
```

```

{
    int    i = 1
    int    k = 0
    void func1 (int);
    void func2 (int) { /* ... */ }
}
namespace demo{ // Расширение
    // int    I = 2 неверно - двойное определение
    void func1 (double); // Перегрузка
    void func2 (int);     // Верно (повторное объявление)
}

```

В объявлении поименованной области могут присутствовать как объявления, так и определения. Логично помещать в неё только объявления, а определять их позднее с помощью имени области и оператора доступа к области видимости `::`, например:

```
void demo :: func1 (int)    { /* ... */ }
```

Это применяется для разделения интерфейса и реализации. Таким способом нельзя объявить новый элемент пространства имён.

Объекты, объявленные внутри области, являются видимыми с момента объявления. К ним можно явно обращаться с помощью имени области и оператора доступа к области видимости `::`, например:

```
demo :: i = 100; demo :: func2(10);
```

Если необходимо часто применять операцию указания области видимости, то целесообразно использовать оператор **using**, который позволяет получить доступ к конкретному имени в пространстве имен, либо ко всем именам пространства имен.

Для получения доступа к отдельному имени в пространстве имен используется оператор в следующем формате:

```
using имя_пространства_имен :: имя;
```

Например, для того, чтобы обеспечить доступ к переменной *x* пространства имен *A* из функции *main()* приведенной программы, достаточно выполнить оператор *using A::x*.

Для обеспечения доступа ко всем именам некоторого пространства имен используется оператор *using* в следующем формате:

```
using namespace имя_пространства_имен;
```

После выполнения такого оператора указанное пространство имен добавляется в текущую область видимости.

Операторы *using* и *using namespace* можно использовать и внутри объявления поименованной области, чтобы сделать в ней доступными объявления из другой области:

```

namespace Department_of_Applied_Mathematics
{
    using demo:: i;
    // ...
}

```

Имена, объявленные в поименованной области явно или с помощью оператора *using*, имеют приоритет по отношению к именам, объявленным с помощью оператора *using namespace* (это имеет значение при включении нескольких поименованных областей, содержащих совпадающие имена).

Короткие имена пространств имён могут войти в конфликт друг с другом, а длинные непрактичны при написании реального кода, поэтому допускается вводить синонимы имён:

```
namespace DAM = Department_of_Applied_Mathematics;
```

Для уменьшения вероятности конфликта имен в последних версиях компиляторов библиотека C++ определена в собственном пространстве имен *std*. Использование оператора *using namespace std;* дает возможность добавить в текущую область видимости все имена, содержащиеся внутри библиотеки C++.

В языке C++ можно объявить **безымянное пространство имен**.

```

namespace { //объявления
}

```

Безымянные пространства имен дают возможность полностью закрыть доступ извне к именам пространства имен, поскольку получить доступ к именам с помощью оператора *using* или операции `::` невозможно, т. к. отсутствует имя у пространства имен.

Если имя области не задано, компилятор определяет его самостоятельно с помощью уникального идентификатора, различного для каждого модуля. Объявление объекта в неименованной области равнозначно его описанию как глобального с модификатором *static*. Помещать объявления в такую область полезно для того, чтобы сохранить локальность кода. Нельзя получить доступ из одного файла к элементу неименованной области другого файла.

**Пример 2. Применение нескольких пространств имен.**

```

#include <iostream>
//Определение первого пространства имен
namespace name_A
{
    int i;
    class P
    {
        int x;
        public:
        P(int newy) {x=newy;}
        void izmx(int c) {x=c;}
        int dost() {return x;}
    };
    char str[]="\n Пространство имен";
}
//Определение первой части второго пространства имен
namespace name_B
{
    int y;
}
//Определение второй части второго пространства имен
namespace name_B
{
    int z;
}
int main()
{
    //Создание объекта b класса P
    name_A :: P b(2) ;
    cout<<"\n Значение элемента данных x объекта b:"<<b.dost();
    cout << endl;
    b.izmx (15);
    cout<<"Новое значение элемента данных x объекта b:"<<b.dost();
    //Задание видимости строки str в текущем пространстве имен using name_A ::
    str;
    cout << str << endl;
    //Задание видимости пространства имен name_A в текущей //области видимости
    using namespace name_A;
    for (i=1;i<11;i++)
        cout << i << " ";
    //Использование обеих частей второго пространства имен
    name_B :: y=100;
    cout << "\n name_B :: y=" << name_B :: y;
    name_B :: z=200;
    cout << "\n name_B :: z=" << name_B :: z;
    //Присоединение пространства имен name_B к текущему //пространству имен
    using namespace name_B;
    P by(y), bz(z);
    cout<<"\n Значение элемента данных x объекта by:"<<by.dost();
    cout<<"\n Значение элемента данных x объекта bz:"<<bz.dost();
    cout << endl;
    return 0;
}

```

В результате работы программы на экран будет выведено следующее:

```

Значение элемента данных x объекта b:2
Новое значение элемента данных x объекта b:15
Пространство имен
12345678910
name_B :: y=100
name_B :: z=200
Значение элемента данных x объекта by:100
Значение элемента данных x объекта bz:200

```

В приведенной программе определены два пространства имен: *name\_A* и *name\_B*. Кроме того, существует глобальное пространство имен программы. В пространстве имен *name\_A* видимы класс *P*, переменная *i* и



строка *str*. Пространство *name\_B* состоит из двух частей. В первой части пространства *name\_B* видима переменная *y*, а во второй — переменная *z*. Описание функции *main()* находится в глобальном пространстве имен.

При расширении области видимости с помощью оператора *using* необходимо следить за тем, чтобы в полученном совместном пространстве имен отсутствовали одинаковые имена. В случае наличия таковых действующим является имя, которое объявлено в текущем пространстве имен.

Механизм пространств имен вместе с директивой *#include* обеспечивают необходимую при написании больших программ гибкость путем сочетания логического группирования связанных величин и ограничения доступа.

Как правило, в любом функционально законченном фрагменте программы можно выделить интерфейсную часть (например, заголовки функций, описания типов), необходимую для использования этого фрагмента, и часть реализации, то есть вспомогательные переменные, функции и другие средства, доступ к которым извне не требуется. Пространства имен позволяют скрыть детали реализации и, следовательно, упростить структуру программы и уменьшить количество потенциальных ошибок. Продуманное разбиение программы на модули, четкая спецификация интерфейсов и ограничение доступа позволяют организовать эффективную работу над проектом группы программистов.

Таблица 5. Операции C++

#### Унарные:

&	получение адреса операнда	
*	обращение по адресу (разыменование)	
-	унарный минус, меняет знак арифметического операнда	
~	побитовое инвертирование внутреннего двоичного кода (побитовое отрицание)	
!	логическое отрицание (НЕ). В качестве логических значений используется 0 - ложь и не 0 - истина, отрицанием 0 будет 1, отрицанием любого ненулевого числа будет 0.	
++	увеличение на единицу: префиксная операция - увеличивает операнд до его использования, постфиксная операция увеличивает операнд после его использования.	
--	уменьшение на единицу: префиксная операция - уменьшает операнд до его использования, постфиксная операция уменьшает операнд после его использования.	
sizeof	вычисление размера (в байтах) для объекта того типа, который имеет операнд	

#### Бинарные операции.

##### Аддитивные:

+	бинарный плюс (сложение арифметических операндов)	
-	бинарный минус (вычитание арифметических операндов)	

##### Мультипликативные:

*	умножение операндов арифметического типа	
/	деление операндов арифметического типа (если операнды целочисленные, то выполняется целочисленное деление)	
%	получение остатка от деления целочисленных операндов	

**Операции сдвига (определены только для целочисленных операндов).**

Формат выражения с операцией сдвига:

*операнд\_левый операция\_сдвига операнд\_правый*

<<	сдвиг влево битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого операнда	
>>	сдвиг вправо битового представления значения правого целочисленного операнда на количество разрядов, равное значению правого операнда	

**Поразрядные операции:**

&	поразрядная конъюнкция (И) битовых представлений значений целочисленных операндов	
	поразрядная дизъюнкция (ИЛИ) битовых представлений значений целочисленных операндов	
^	поразрядное исключающее ИЛИ битовых представлений значений целочисленных операндов	

**Операции сравнения:**

<	меньше, чем	
>	больше, чем	
<=	меньше или равно	
>=	больше или равно	
==	равно	
!=	не равно	

**Логические бинарные операции:**

&&	конъюнкция (И) целочисленных операндов или отношений, целочисленный результат ложь(0) или истина(1)	
	дизъюнкция (ИЛИ) целочисленных операндов или отношений, целочисленный результат ложь(0) или истина(1)	

Таблица 6. Приоритеты операций C++

При-ори-тет	Знаки операций	Названия операций	Порядок выполнения
1	::	разрешение области видимости	слева
2	. -> [] ( ) ++ -- typeid dynamic_cast static_cast reinterpret_cast constcast	выбор элемента по имени выбор элемента по указателю выбор элемента по индексу вызов функции или конструирование значения постфиксный инкремент постфиксный декремент идентификация типа преобразование с проверкой при выполнении преобразование с проверкой при компиляции преобразование без проверки константное преобразование	слева
3	sizeof ++ -- ~ ! + - & * new delete (имя_типа)	размер операнда в байтах префиксный инкремент префиксный декремент инверсия (поразрядное НЕ) логическое НЕ унарный плюс унарный минус адрес разыменование выделение памяти или создание освобождение памяти или уничтожение преобразование типа	справа
4	.* ->*	выбор элемента по имени через указатель выбор элемента по указателю через указатель	слева
5	* / %	умножение деление остаток от деления целых (деление по модулю)	слева
6	+ -	сложение вычитание	слева
7	<< >>	сдвиг влево сдвиг вправо	слева
8	< > <= >=	меньше больше меньше или равно больше или равно	слева
9	== !=	равно не равно	слева
10	&	поразрядное И	слева
11	^	поразрядное исключающее ИЛИ	слева
12	&	поразрядное ИЛИ	слева
13	&&	логическое И	слева

14		логическое ИЛИ	слева
15	? :	условная	справа
16	= *= /= %= += -= <<= >>= &= ^=  =	присваивания (простое и составные)	справа
17	throw	генерация исключения	справа
18	,	последовательность выражений	слева

### Арифметические преобразования в выражениях

1. Прежде всего каждый операнд типа *char* или *short* преобразуется в значение типа *int*, а операнды типа *unsigned short* преобразуются в значение типа *unsigned int*.
2. Кроме того, операнды типа *float* до начала операции преобразуются в значение типа *double*.
3. Затем если один из операндов имеет тип *double*, то другой преобразуется в значение типа *double* и результат будет иметь тип *double*.
4. Иначе если один из операндов имеет тип *unsigned long*, то другой преобразуется в значение типа *unsigned long* и таким же будет тип результата.
5. Иначе если один из операндов имеет тип *long*, то другой преобразуется в значение типа *long* и таким же будет его результат.
6. Иначе если один из операндов имеет тип *long*, а другой – тип *unsigned int*, то оба операнда преобразуются в значение *unsigned long*.
7. Иначе если один из операндов имеет тип *unsigned*, то другой преобразуется в значение типа *unsigned* и тип результата *unsigned*.
8. Иначе оба операнда должны быть типа *int* и таким же будет тип результата.
9. Программист может задать преобразования типа явным образом.

### Пример использования оператора переключатель

(программа реализует простейший калькулятор на 4 действия):

```
#include <iostream.h>
int main ()
{
    int a, b, res;
    char op;
    cout << "\nВведите первый операнд:"; cin >> a;
    cout << "\nВведите знак операции:" ; cin >> op;
    cout << "\nВведите второй операнд:"; cin >> b;
    bool f = true;
    switch (op)
    {
        case '+': res = a + b; break;
        case '-': res = a - b; break;
        case '*': res = a * b; break;
        case '/': res = a / b; break;
        default: cout << "\nНеизвестная операция";
        f = false;
    }
    if (f) cout << "\nРезультат:" << res;
    return 0;
}
```

## Циклы

Часто встречающиеся **ошибки программирования циклов** –использование в теле цикла неинициализированных переменных и неверная запись условия выхода из цикла.

Чтоб избежать ошибок рекомендуется :

- проверить, всем ли переменным, встречающимся в правой части операторов присваивания в теле цикла, присвоены до этого начальные значения;
- проверить, изменяется ли в цикле хотя бы одна переменная, входящая в условие выхода из цикла;
- предусмотреть аварийный выход из цикла по достижению некоторого количества итераций;
- и конечно, не забывать о том, что если в теле цикла требуется выполнить более одного оператора, нужно заключать их в фигурные скобки.

Операторы цикла взаимозаменяемы. Но можно привести некоторые рекомендации по выбору наилучшего в каждом конкретном случае.

Оператор *do while* обычно используют, когда цикл требуется выполнить хотя бы раз.

Оператором *while* удобнее пользоваться в случаях, когда число итераций заранее неизвестно, очевидных параметров цикла нет.

Оператор *for* предпочтительнее в большинстве остальных случаев (однозначно для организации циклов со счетчиками). В C++ он является более гибким средством, чем аналогичные операторы циклов в других языках программирования.

### Возможности применения цикла типа *for*.

1. Можно применять операцию уменьшения для счета в порядке убывания.

**Пример.** Требуется вычислить  $y^5$ . Возможное решение имеет вид:

```
for ( i=5, r=1; i>=1; i--) r=r*y;
cout<<"r="<<r;
```

2. При желании можно организовать счет двойками, тройками, десятками и т.д.

**Пример.** Приращение при счете, отличное от 1.

```
for ( n=5; n<61; n+=15) cout<<n;
```

3. Можно в качестве счетчика использовать не только цифры, но и символы.

**Пример.** Требуется напечатать алфавит. Возможное решение имеет вид:

```
for ( chr='A'; chr<='Z'; chr++) cout<<chr;
```

4. Можно задать возрастание значений счетчика не в арифметической, а в геометрической прогрессии.

**Пример.** Требуется подсчитать долг. Возможное решение имеет вид:

```
for ( k=100; k<185; k*=1.1) cout<<"Долг="<<k;
```

5. В качестве третьего выражения можно использовать любое правильно составленное выражение. Оно будет вычисляться в конце каждой итерации.

**Пример.** Использование в качестве счетчика выражения.

```
for (k=1; z<=196; z=5*k+23 ) cout<<z;
```

6. Можно пропускать одно или несколько выражений. (При этом нельзя пропускать символы «точка с запятой».)

**Пример.** Неполный список выражений в заголовке тела цикла.

```
for (p=2; p<=202;) p=p+n/k;
```

7. Первое выражение не обязательно должно инициализировать переменные, оно может быть любого типа.

**Пример.** Произвольное первое выражение в заголовке цикла.

```
for (cout<<"Вводите числа."; p<=30;) cin>>p;
```

8. Переменные, входящие в выражения спецификации цикла, можно изменять в теле цикла.

**Пример.** Изменение управляющих переменных в теле цикла.

```
delta=0.1;
for (k=1; k<500; k+=delta) if (a>b) delta=0.5;
```

9. Использование операции «запятая» в спецификации цикла позволяет включать несколько инициализирующих и корректирующих выражений.

**Пример.** Использование операции «запятая» в спецификации цикла.

```
for (i=1, r=1; i<=10; i++, r*=y)
    cout<<"y в степени "<<i<<"="<<r;
```

10. Можно проверять любое другое условие, отличное от числа итераций.

### Функции для работы со случайными числами в C++

Для эффективного использования в программе случайных чисел необходимо два средства:

1. функция-генератор, которая при каждом вызове будет возвращать случайное число;
2. функция-инициализатор для случайной инициализации генератора.

Необходимость во второй функции вызвана тем, что случайные числа, полученные программным путем, на самом деле являются псевдослучайными. Они вычисляются по определенной формуле, и при одинаковых начальных условиях последовательность чисел будет повторяться. При вызове функции-инициализатора для функции-генератора устанавливается новое начальное значение.

C++ предлагает целый ряд стандартных функций для работы со случайными объектами.

#### Функции `rand` и `srand`

##### Синтаксис:

```
#include<stdlib.h>
int rand(void);
```

##### Файл, содержащий прототип

`stdlib.h`

##### Прототип: `int rand (void);`

мультипликативный конгруэнтный генератор случайных чисел с периодом  $2$  в  $32$  степени. При каждом вызове возвращает следующее псевдослучайное число в пределах от  $0$  до `RAND_MAX`. `RAND_MAX` - символьная константа, объявленная в файле `stdlib.h`, которая определяет наибольшее случайное число.

```
/* (в модуле stdlib.h) */
#define RAND_MAX 32767;
```

Совместимость: *POSIX*, *Win32*, *ANSI C*, *ANSI C++*.

##### Пример:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int i;
    printf("10 случайных чисел от 0 до 99 \n\n");
    for (i=0; i<10; i++)
        printf("%d\n", rand()%100);
    return 0;
}
```

##### Синтаксис:

```
#include <time.h>
void srand (unsigned seed);
```

##### Файл, содержащий прототип

`time.h`

##### Прототип:

```
void srand (unsigned seed);
```

инициализирует генератор случайных чисел. Начальная инициализация происходит вызовом `srand` с аргументом  $1$ . Новое начальное значение устанавливают вызовом этой функции с другим аргументом.

Совместимость: *POSIX*, *Win32*, *ANSI C*, *ANSI C++*.

##### Пример.

```
#include <iostream>
#include <time.h>
int main() {
```

```

int a;
srand(time(0)); //инициализация генератора псевдослучайных чисел
a=rand();       // получаем очередное псевдослучайное число
cout<<a; // вывод на экран
cin.get();
}
/*srand - Это инициализация генератора случайных чисел, time - возвращает
текущее календарное время системы, в качестве аргумента она принимает указатель
на переменную типа time_t, которой и будет присвоено календарное время.
Прототип функции выглядит так:
    time_t time(time_t* time);
Календарное время и возвращается функцией, и помещается в переданный аргумент,
но можно передавать нулевой указатель (то есть 0 или null). 0 в данном случае
не число, а нулевой указатель.*/

```

#### Пример:

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>

int main ()
{
    int num;
    //Инициализируем генератор случайных чисел
    srand ( time(NULL) );
    printf ("RAND_MAX: %d \n",RAND_MAX);
    num = rand();
    printf ("Guess the number: %d \n",num);
    num = rand() % 10;
    printf ("Guess the number (0 to 9): %d \n",num);
    num = rand() % 10 + 1;
    printf ("Guess the number (1 to 10): %d \n",num);
    num = (rand() % 2)*2 - 1;
    printf ("Guess the number (-1 or 1): %d \n",num);
    getch();
    return 0;
}

```

#### Пример.

```

/* устанавливает генератор случайных чисел в начальное состояние и сохраняет в
массиве 20 первых сгенерированных случайных чисел. */
#include <stdlib.h>
#include <time.h>

int main ()
{
    int x, ranvals [20];
    srand(17);
    for (x = 0; x < 20; ranvals[x++] = rand() );
    return 0;
}

```

## Функции

**Пример** функции, возвращающей сумму двух целых величин:

```
#include <iostream.h>
int sum(int a, int b);
//объявление функции (прототип) можно так int sum(int, int);
int main ( )
{
    int a=2, b=3, c, d;
    c=sum(a, b); // вызов функции
    cin >> d;
    cout << sum(c, d); // вызов функции
    return 0;
}
#include "sum.cpp"; //включение файла sum.cpp с функцией sum
```

Описание функции sum находится в файле sum.cpp и имеет следующий вид:

```
int sum(int a, int b) //определение функции
{return (a+b);}
```

**Пример.** Функция без возвращаемого значения.

Составить программу, которая включает функцию, обеспечивающую выдачу на экран символа, набранного на клавиатуре. При нажатии клавиши с символом С выполнение программы завершить.

```
# include<iostream.h>
void pr(char); //прототип
void main ()
{
    char x;
    do
        {cout<<"\n введите символ";
        cin>>x;
        pr (x); //вызов как оператор
        }
    while(x!= 'C');
}
void pr (char a)
{cout<<"\n введен символ:" <<a;}
```

## Массивы как параметры функций

### Одномерные массивы

Массивы могут быть параметрами функций и функции в качестве результата могут возвращать указатель на массив. При использовании массивов в качестве параметров функции возникает необходимость определения в теле функции количества элементов массива, который является аргументом при обращении к функции.

При работе со **строками**, т. е. с массивами типа `char[]`, проблема решается просто, поскольку последний элемент строки имеет значение `'\0'`. Поэтому при обработке массива-аргумента каждый его элемент анализируется на наличие символа конца строки.

**Пример 1.** Строки как параметры функций.

Требуется составить программу, содержащую функцию, которая подсчитывает число элементов в строке. Возможный вариант программы имеет вид:

```
#include <iostream.h>
#include <conio.h>
int dl(char[]); // прототип функции dl
void main ()
{
    clrscr();
    char c[]="kafedra ASOI";
    cout << "\n kol-vo simvolov:" << dl(c);
```



```

    getch();
}

int dl(char c[])
{
    int i;
    for (i=0; ;i++)
        if (c[i]=='\0') break;
    return i;
}

```

В результате выполнения программы на экран будет выдано сообщение:  
Длина строки равна: 12.

Если массив-параметр функции не является *символьной строкой*, то необходимо либо использовать массивы с заранее известным числом элементов, либо передавать размер массива с помощью дополнительного параметра.

**Пример 2.** Одномерный массив в качестве параметра функции.

Пусть требуется составить программу, в которой функция вычисляет сумму элементов массива, состоящего из 5 элементов.

```

//Одномерный массив в качестве параметра
#include <iostream.h>
float sum (float x[5])
{
    float s=0;
    for (int i=0; i<5; i++)
        s=s+x[i];
    return s;
}

void main()
{
    float z[5],y;
    for (int i=0; i<5; i++)
    {
        cout <<"\nВведите очередной элемент массива:";
        cin >> z[i];
    }
    y=sum(z) ;
    cout <<"\n Сумма элементов массива:"<< y;
}

```

В результате выполнения программы на экран будет выведено значение суммы элементов массива *z*.

В данном примере отсутствует прототип функции *sum()*, поскольку описание функции *sum()* компилятор анализирует раньше, чем обращение к ней. В приведенном примере заранее известно число элементов — 5, поэтому в функции всего один параметр — массив *x*. Поскольку в функции существует возвращаемое значение, то вызов функции может быть только выражением или частью выражения. В программе оператор присваивания *y=sum(z);* содержит такое выражение в правой части. Здесь аргументом функции является массив *z*.

**Пример 3.** Одномерный массив с произвольным числом элементов в качестве параметра функции.

Пусть требуется составить программу с обращением к функции, которая возвращает максимальный элемент одномерного массива с произвольным числом членов.

```

//Поиск максимального элемента
#include <iostream.h>

float max(int n, float a[])
{
    float m=a[0];
}

```

```

    for (int i=1;i<n;i++)
        if (m<a[i]) m=a[i];
    return m;
}

void main()
{
    float z[6];
    for (int i=0;i<6;i++)
    {
        cout <<"\nВведите очередной элемент массива:";
        cin >> z[i];
    }
    cout <<"\nМаксимальный элемент массива:"<< max(6,z);
}

```

В результате выполнения программы на экран монитора будет выдано сообщение со значением максимального элемента массива *z*, который в данном примере состоит из шести элементов.

В приведенной программе прототип отсутствует, поскольку описание функции следует раньше ее вызова. В функции *max()* используется два параметра: первый указывает число элементов в массиве, а второй — имя и размерность массива. Аргументы указаны при вызове функции *max(6, z)*. Здесь 6 — размер массива, а *z* - имя массива.

Имя массива представляет собой **константный указатель адреса** нулевого элемента массива. Поэтому, имя массива является константным указателем адреса элемента массива с нулевым индексом. Отсюда следует, что при использовании имени массива в качестве параметра функции допускается изменять значения элементов массива.

**Пример 4. Массив как параметр функции.**

Пусть требуется составить программу, содержащую обращение к функции *der()*, которая уменьшает на 1 значения всех элементов массива-параметра. Возможный вариант решения имеет вид:

```

#include <iostream.h>
void der(int, float[]); // прототип функции
void main () {
    const int k=10;
    float a[k];
    int i;
    for (i=0; i<k; i++)
    {
        cout << "\nВведите элемент массива ";
        cin >> a[i];
    }
    der(k,a);           // обращение к функции
    for (i=0; i<k; i++)
        cout << "\ta[" << i << "]= " << a[i];
}
void der (int k1, float a1[])
{
    int j;
    for (j=0; j<k1;j++)
        a1[j]--;
}

```

**Пример 5. Массивы с произвольным числом элементов как параметры функции.**

Пусть требуется составить программу с функцией, формирующей в качестве результата массив, каждый элемент которого является максимальным из соответствующих значений элементов двух других массивов-параметров.

```

//Указатели на массив в качестве параметров
#include <iostream.h>
#include <conio.h>

```

```

void maxl(int,int*,int*,int*); //прототип

void main ()
{
    clrscr();
    int a[]={0,1,2,3,4};
    int b[]={5,6,0,7,1};
    int d[5];
    maxl (5,a,b,d) ;
    cout << "\n";
    for (int i=0;i<5;i++)
        cout << "\t" << d[i] ;
    getch();
}

void maxl (int n, int *x, int *y, int *z)
{
    for (int i=0;i<n;i++)
        z[i]=x[i]>y[i] ? x[i] :y[i];
}

```

В результате выполнения программы на экран монитора будут выданы элементы результирующего массива *d*:

5 6 2 7 4.

В приведенной программе у функции имеется четыре параметра: число *n* элементов в массивах и три указателя *x*, *y*, *z*. При вызове функции *maxl(5,a,b,d);*, (с помощью оператора, поскольку отсутствует возвращаемое значение) в качестве аргументов используются имена исходных массивов *a* и *b* и результирующего *d*.

### Многомерные массивы

Особенностью языка C++ является несамоопределенность массивов, т. е. по имени массива невозможно узнать его размерность и размеры по каждому измерению. Кроме того, в C++ многомерные массивы не определены. Например, если объявлен массив *float d[3][4][5]*, то это не трехмерный, а одномерный массив *d*, включающий три элемента, каждый из которых имеет тип *float [4][5]*. В свою очередь, каждый из четырех элементов типа *float [5]*. И, соответственно, каждый из этих элементов является массивом из пяти элементов типа *float*. Эти особенности затрудняют использование массивов в качестве параметров функций.

При передаче массивов в качестве параметров через заголовок функции следует учитывать, что передавать массивы можно только с одной неопределенной границей мерности (эта мерность должна быть самой левой).

#### **Пример 6.** Двумерный массив как параметр функции.

Пусть требуется составить программу с функцией, которая подсчитывает сумму элементов матрицы.

```

#include <iostream.h>
float summa(int n,float a[][3])
{
    float s=0;
    for (int i=0;i<n;i++)
        for (int j=0;j<3;j++)
            s=s+a[i][j];
    return s;
}

void main()
{
    float z[4][3] = {0,1,2,3,4,5,6,7,7,6,5,4};
    cout << "\n Сумма элементов матрицы равна " << summa(4,z) ;
}

```

В результате выполнения программы на экран будет выведено сообщение:  
Сумма элементов матрицы равна 50.

В приведенной программе параметром функции *summa()* является матрица *a*[[3], у которой не определено число строк. Число столбцов должно быть заранее определено, оно равно 3. Таким образом, при использовании данной функции вводится существенное ограничение — фиксированное число столбцов. Указанное ограничение можно обойти с помощью использования вспомогательных массивов указателей на массивы.

**Пример 7.** Вспомогательный массив указателей на массив как параметр функции.

Требуется составить программу с функцией, которая возвращает в качестве результата минимальный элемент матрицы *d* размером *m*х*n*.

```
#include <iostream.h>

float min(int m,int n,float *p[]); //прототип

void main()
{
    float d[3][4]={1,2,-2,4,5,0,-3,18,-9,6,7,9};
    float *r[]={ (float*)&d[0], (float*)&d[1], (float*)&d[2]};
    int m=3;
    int n=4;
    cout<<"\n Минимальный элемент матрицы равен "<<min(m,n,r);
}

float min(int m,int n,float *p[])
{
    float x=p[0][0];
    for (int i=0;i<m;i++)
        for (int j=0;j<n;j++)
            if (x>p[i][j]) x=p[i][j];
    return x;
}
```

В результате выполнения программы на экран будет выведено сообщение:

Минимальный элемент матрицы равен -9.

В функции *min()* в качестве параметров используются *int m* — число строк, *int n* — число столбцов и *float \*p[]* — массив указателей на одномерные массивы элементов типа *float*, причем размеры двумерного массива заранее неизвестны.

В теле функции обращение к элементам матрицы осуществляется с помощью двойного индексирования. Здесь *p[i][j]* — значение элемента двумерного массива.

В функции *main()* объявлена и инициализирована матрица *d*[3][4], имеющая фиксированные размеры. Такой размер нельзя использовать непосредственно в качестве аргумента функции, поэтому объявлен дополнительный вспомогательный массив указателей *float \*r[]*. В качестве значений элементам этого массива присваиваются адреса первых элементов строк матрицы, т. е. *&d[0]*, *&d[1]*, *&d[2]*, преобразованные к типу *float \** (указатель на тип *float*).

## Динамические массивы

При использовании в качестве параметра массива в функцию передается указатель на его первый элемент, то есть массив всегда передается по адресу. При этом информация о количестве элементов массива теряется, и следует передавать его размерность через отдельный параметр.

**Пример 8.** Использование одномерного динамического массива. Нахождение суммы элементов массива.

```
#include <iostream.h>
int sum (const int *mas, const int n); //прототип
int const n=10;
int main()
{
    int marks[n] = {3, 4, 5, 4, 4};
    cout << "Сумма элементов массива: " << sum(marks, n);
    return 0;
}
```

```

int sum(const int *mas, const int n)
    // варианты: int sum(int mas[], int n)
    // или       int sum(int mas[n], int n)
    // (величина n должна быть константой)
{
    int s = 0;
    for (int i=0; i<n; i++)
        s += mas[i];
    return s;
}

```

**Пример 9.** Динамические массивы как параметры функций.

Требуется составить программу с использованием функции, которая заполняет элементы матрицы размером *m*×*n* последовательно значениями целых чисел: 0, 1, 2, 3,... Для формирования матрицы в основной программе использованы динамические массивы, так как размеры матрицы заранее не известны.

```

// Матрица как набор одномерных динамических массивов
#include <iostream.h>
void fun(int m, int n, int **uc)
{
    int k=0;
    for (int i=0; i<m; i++)
        for (int j=0; j<n; j++)
            uc[i][j]=k++;
}

void main()
{
    int **pi; // указатель на массив указателей
    int m1;   // число строк в матрице
    cout << "\n Введите число строк матрицы:";
    cin >> m1;
    int n1;   // число столбцов в матрице
    cout << "\n Введите число столбцов матрицы:";
    cin >> n1;
    int i, j;
    // выделение вспомогательного массива указателей
    pi=new int* [m1];
    for (i=0; i<m1; i++)
        // формирование i-й строки матрицы
        pi[i]=new int [n1];
    fun(m1, n1, pi); // обращение к функции
    for (i=0; i<m1; i++) // цикл перебора строк
    {
        cout << "\n строка " << i+1 << ":";
        for (j=0; j<n1; j++)
            cout << "\t" << pi[i][j];
    }
    //
    for (i=0; i<m1; i++)
        delete []pi[i];
    delete []pi;
}

```

Если при вводе задать размерность матрицы 3×4, то в результате выполнения программы на экран будет выдано три следующие строки:

```

строка 1: 0 1 2 3
строка 2: 4 5 6 7
строка 3: 8 9 10 11

```

В функции *fun()* есть три следующих параметра: *int m* — число строк, *int n* — число столбцов матрицы и *int \* \*uc* — указатель на массив указателей.

В функции *main* для определенности значение числа строк *m1* задано 3, а число столбцов матрицы *n1* задано 4. При выполнении оператора присваивания *pi=new int\*[m1]*; операцией *new* выделяется память для вспомогательного массива указателей. В результате выполнения этой операции для массива указателей выделяется требуемое количество памяти, а адрес первого элемента выделенной памяти присваивается указателю *pi*. При выполнении в теле цикла второй операции *new* формируются *m1* строк матрицы.

При обращении к функции происходит присваивание элементам матрицы последовательно значений целых чисел от 0 до 11. В конце программы с помощью *delete* выделенная память освобождается.

Многомерный массив с переменными размерами, который формируется в теле вызываемой функции, невозможно целиком вернуть в качестве результата в вызывающую функцию. Однако возвращаемым значением функции может быть указатель на одномерный массив указателей на одномерные массивы с элементами заданного типа.

**Пример 10.** Формирование многомерного динамического массива в функции.

Пусть требуется составить программу с функцией, формирующую квадратную матрицу порядка *n* с единичными диагональными элементами и остальными нулевыми.

```
// Единичная диагональная матрица порядка n
#include <iostream.h>
#include <process.h>      //для exit ()
// функция формирования матрицы
int **matr(int n)
{
    int **p;                //вспомогательный указатель
    p=new int* [n];         //массив указателей на строки
    if (p==NULL)
    {
        cout << "\n Не создан динамический массив!!";
        exit(1);
    }
    //цикл создания строк (внешний)
    for (int i=0;i<n;i++)
    {
        p[i]=new int [n];   //выделение памяти для i-й строки
        if (p[i]==NULL)
        {
            cout << "\n Не создан динамический массив!";
            exit(1);
        }

        //цикл заполнения строк (вложенный)
        for (int j=0;j<n;j++)
            if (j!=i) p[i][j]=0;
            else p[i][j]=1;
    }
    return p;
}

void main()
{
    int n1;                //порядок матрицы
    cout << "\n Введите порядок матрицы:";
    cin >> n1;
    int **ma;              //указатель для формирования матрицы
    ma=matr(n1);           //обращение к функции
    //печать элементов матрицы
    for (int i=0;i<n1;i++)
    {
        cout << "\n строка " << i+1 << ":";
        for (int j=0;j<n1;j++)
```

```

        cout << "\t" << ma[i][j];
    }
    //освобождение памяти
    for (i=0;i<n1;i++)
        delete []ma[i]; //удаляем содержимое строк
    delete []ma;         //удаляем массив указателей на строки
}

```

При выполнении программы на экране появится подсказка:

Введите порядок матрицы:

Если в ответ на подсказку пользователь введет с клавиатуры порядок матрицы, равный 5, то на экране дисплея он получит сообщение из пяти строк вида:

```

строка 1:    1 0 0 0 0
строка 2:    0 1 0 0 0
строка 3:    0 0 1 0 0
строка 4:    0 0 0 1 0
строка 5:    0 0 0 0 1

```

В данной программе функция *matr()* имеет один параметр *int n* — порядок матрицы. В теле функции формируется квадратная матрица размером *nхn* (создаются *n* одномерных массивов с элементами типа *int* и массив указателей на эти одномерные массивы) и ее элементы заполняются необходимыми значениями. Возвращаемым значением функции *matr()* является значение указателя на сформированную матрицу.

### **Пример 11.** Нахождение суммы элементов двух двумерных массивов.

Внутри функции массив интерпретируется как одномерный, а его индекс пересчитывается в программе. Размерность массива *b* известна на этапе компиляции, под массив *a* память выделяется динамически (программа написана в стиле C).

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <malloc.h>

int sum(const int *a, const int nstr, const int nstb);

int main()
{
    clrscr();
    int b[2][2]= {{2, 2},{4,3}};
    printf ("Сумма элементов b:  %d\n", sum(&b[0][0], 2, 2));
    //имя массива передавать в sum нельзя из-за несоответствия типов
    int i,j, nstr, nstb, *a;
    printf ("Введите количество строк и столбцов: \n");
    scanf("%d%d", &nstr, &nstb);
    a=(int *)malloc(nstr* nstb* sizeof(int)); //1
    printf ("Введите элементы массива: \n");
    for (i=0; i<nstr; i++)
        for (j=0; j<nstb; j++)
            scanf("%d", &a[i * nstb + j]); //2
    printf("Сумма элементов a: %d\n", sum(a, nstr, nstb));
    free(a);
    return 0;
}

int sum(const int *a, const int nstr, const int nstb)
{
    int i,j,s=0;
    for (i=0; i<nstr; i++)
        for (j = 0; j<nstb; j++)
            s += a[i * nstb + j];
    return s;
}

```

Память выделяется сразу под все элементы массива (оператор 1), то есть, по сути двумерный динамический массив занимает сплошной участок памяти. При заполнении массива используется формула для определения индекса соответствующего элемента (2). В конце работы программы массив удаляется.

Можно решить эту задачу по-другому выделяя память (программа написана в стиле C++).

```
#include <iostream.h>

int sum(int **a, const int nstr, const int nstb);

int main()
{
    int nstr, nstb;
    cout<<"Введите количество строк и столбцов: \n";
    cin >> nstr >> nstb;
    int i,j, **a;
    a=new int *[nstr];
    for (i=0; i<nstr; i++)
        a[i]=new int [nstb];
    cout<<"Введите элементы массива: \n";
    for (i=0; i<nstr; i++)
        for (j=0; j<nstb; j++)
            cin>>a[i][j];
    cout<<"Сумма элементов a:"<<sum(a, nstr, nstb);
    //освобождение памяти
    for (i=0;i<nstr;i++)
        delete []ma[i]; //удаляем содержимое строк
    delete []ma;          //удаляем массив указателей на строки
    return 0;
}

int sum(int **a, const int nstr, const int nstb)
{
    int i,j,s=0;
    for (i=0; i<nstr; i++)
        for (j = 0; j<nstb; j++)
            s += a[i][j];
    return s;
}
```

В этом случае память сначала выделяется под столбец указателей на строки матрицы, а затем в цикле под каждую строку. Освобождение памяти должно выполняться в обратном порядке.

### Рекурсивные функции

Для реализации рекурсивных алгоритмов в C++ предусмотрена возможность создания рекурсивных функций. **Рекурсивная функция** представляет собой функцию, в теле которой осуществляется вызов этой же функции.

**Пример 1.** Использование рекурсивной функции для вычисления факториала.

Пусть требуется составить программу вычисления факториала произвольного положительного числа.

```
#include <iostream.h>
int fact(int n)
{
    int a;
    if (n<0) return 0;
    if (n==0) return 1;
    a =n * fact(n-1);
    return a;
}
void main()
{
    int m;
    cout << "\nВведите целое число:";
```



```

    cin >> m;
    cout << "\n Факториал числа " << m << " равен " << fact(m);
}

```

Для отрицательного аргумента факториала не существует, поэтому функция в этом случае возвращает нулевое значение. Так как факториал 0 равен 1 по определению, то в теле функции предусмотрен и этот вариант. В случае когда аргумент функции *fact()* отличен от 0 и 1, то вызывается функция *fact()* с уменьшенным на единицу значением параметра и результат умножается на значение текущего параметра. Таким образом, в результате встроенных вызовов функций будет возвращен следующий результат:

$$n * (n-1) * (n-2) * \dots * 2 * 1 * 1$$

Последняя единица при формировании результата принадлежит вызову *fact(0)*.

Поскольку в языке C++ отсутствует операция возведения в степень, то для выполнения возведения в степень вещественного ненулевого числа можно использовать рекурсивную функцию.

**Пример 2.** Использование рекурсивной функции для возведения в степень.

Пусть требуется составить программу возведения в положительную целую степень вещественного ненулевого числа.

```

/* Возведение основания x в степень n */
#include <iostream.h>
float st (float x, int n)
{
    if (n==0) return 1;
    if (x==0) return 0;
    if (n>0) return x*st(x,n-1);
    if (n<0) return st(x,n+1)/x;
}
void main()
{
    int m;
    float a,y;
    cout << "\nВведите основание степени:";
    cin >> a;
    cout << "\nВведите степень числа:";
    cin >> m;
    y=st(a,m);
    cout << "\n Число " << a << " в степени " << m << " равно:" << y;
}

```

В функции предусмотрены четыре ветви выполнения возведения в степень вещественного числа:

- степень числа равна нулю;
- основание равно нулю;
- степень больше нуля;
- степень меньше нуля.

При вызове функции, когда основание степени больше нуля, например, если при вводе заданы значения переменных  $a=2.0$  и  $m=4$ , тогда обращение  $st(a,m)$  приводит к следующему вычислению:

$$2.0 * 2.0 * 2.0 * 2.0 * 1$$

Вызов функции для отрицательной степени, например, когда значения аргументов равны  $a=3.0$  и  $m=-2$  приводит к вычислению следующего выражения:

$$1/3.0/3.0$$

## ООП

### Структура в роли примитивного класса

Структуры в C++ можно рассматривать как примитивные классы, поскольку они могут содержать не только данные, но и функции. Рассмотрим следующую программу:

```
//
// sqroot.cpp
// В этой программе на языке C++ создается структура,
// содержащая, кроме данных, также функции.
//
#include <iostream.h>
#include <math.h>
struct math_operations {
double data_value;
void set_value(double value) {data_value = value; }
double get_square (void) {return (data_value * data_value); }
double get_square_root (void) {return (sqrt(data_value)); }
} math;
main () {
// записываем в структуру число 35.63
math.set_value(35.63);
cout << "Квадрат числа равен " << math.get_square() << endl;
cout << "Корень числа равен " << math.get_square_root() << endl;
return(0);
}
```

В первую очередь обратите внимание на то, что помимо переменной структура содержит также несколько функций. Это первый случай, когда внутри структуры нам встретились описания функций. Такая возможность существует только в C++. Функции-члены структуры могут выполнять операции над переменными этой же структуры. Неявно подразумевается, что все члены структур являются открытыми, как если бы они были помещены в секцию public.

При выполнении программы на экране отобразится следующая информация:

Квадрат числа равен 1269.5 Корень числа равен 5.96909

В структуре math\_operation объявлена единственная переменная-член data\_value и три функции, описание которых дано тут же внутри структуры. Первая функция отвечает за инициализацию переменной data\_value, другие возвращают соответственно квадрат и квадратный корень числа, хранимого в переменной. Обратите внимание, что последние две функции не принимают никаких значений, поскольку переменная data\_value доступна для них как член структуры.

В следующей программе вновь создается структура, содержащая функции, но на этот раз они описаны вне структуры.

```
//
// trigon.cpp
// В этой программе на языке C++ создается структура,
// содержащая тригонометрические функции.
//
#include <iostream.h>
#include <math.h>
const double DEG_TO_RAD = 0.0174532925;
struct degree {
double data_value;
void set_value(double angle);
double get_sine(void) ;
double get_cosine(void);
double get_tangent(void);
double get_cotangent(void);
double get_secant(void);
double get_cosecant(void);
} deg;
void degree::set_value(double angle)
{
data_value = angle;
}
double degree::get_sine(void)
```

```

{
return(sin(DEG_TO_RAD * data_value));
}
double degree::get_cosine(void)
{
return(cos(DEG_TO_RAD * data_value));
}
double degree::get_tangent(void)
{
return(tan(DEG_TO_RAD * data_value));
}
double degree::get_secant(void)
return(1.0 / sin (DEG_TO_RAD * data_value)) ;
}
double degree::get_cosecant(void)
{
return(1.0 / cos(DEG_TO_RAD * data_value));
}
double degree::get_cotangent(void)
{
return(1.0 / tan(DEG_TO_RAD * data_value));
}
main ()
{
// устанавливаем значение угла равным 25 градусов
deg.set_value(25.0);
cout << "Синус угла равен " << deg.get_sine() << endl;
cout << "Косинус угла равен " << deg.get_cosine() << endl;
cout << "Тангенс угла равен " << deg.get_tangent() << endl;
cout << "Секанс угла равен " << deg.get_secant() << endl;
cout << "Косеканс угла равен " << deg.get_cosecant() << endl;
cout << "Котангенс угла равен " << deg.get_cotangent() << endl;
return (0);
}

```

В этой программе структура содержит прототипы семи функций, но сами они описаны отдельно. Тригонометрические функции вычисляют синус, косинус, тангенс, котангенс, секанс и косеканс угла, значение которого в градусах передается в функцию `set_value()`. Здесь следует обратить внимание на синтаксис заголовка функции, например:

```
void degree::set_value(double angle)
```

Имя функции состоит из имени структуры, за которым расположен оператор `::`.

При вызове функции используется оператор точка (`.`), как и при доступе к переменным-членам структуры. Если бы структура была представлена указателем, доступ к функциям необходимо было бы осуществлять с помощью оператора `->`.

Простейший класс

Рассмотрим следующий пример программы:

```

//
// class.cpp
// В этой программе на языке C++ создается простейший
// класс, имеющий открытые и закрытые члены.
//
#include <iostream.h>
#include <math.h>
const double DEG_TO_RAD = 0.0174532925;
class degree {
double data_value;
public:
void set_value(double angle);
double get_sine(void);
double get_cosine(void);
double get_tangent(void);
double get_secant(void) ;

```

```

double get_cosecant(void);
double get_cotangent(void);
} deg;
void degree::set_value(double angle)
{
data_value = angle;
}
double degree::get_sine(void)
{
return(sin(DEG_TO_RAD * data_value));
}
double degree::get_cosine(void)
{
return(cos(DEG_TO_RAD * data_value));
}
double degree::get_tangent(void)
{
return(tan(DEG_TO_RAD * data_value));
}
double degree::get_secant(void)
return(1.0 / sin (DEG_TO_RAD * data_value)) ;
}
double degree::get_cosecant(void)
{
return(1.0 / cos(DEG_TO_RAD * data_value));
}
double degree::get_cotangent(void)
{
return(1.0 / tan(DEG_TO_RAD * data value));
}
main ()
{
// устанавливаем значение угла равным 25.0 градусов
deg.set_value(25.0);
cout << "Синус угла равен " << deg.get_sine() << endl;
cout << "Косинус угла равен " << deg.get_cosine() << endl;
cout << "Тангенс угла равен " << deg.get_tangent () << endl;
cout << "Секанс угла равен " << deg.get_secant() << endl;
cout << "Косеканс угла равен " << deg.get_cosecant() << endl;
cout << "Котангенс угла равен " << deg.get_cotangent() << endl;
return (0);
}

```

Как видите, тело программы сохранилось прежним. Просто описание структуры было преобразовано в описание настоящего класса C++ с открытой и закрытой секциями.

Программа выведет на экран следующую информацию:

```

Синус угла равен 0.422618
Косинус угла равен 0.906308
Тангенс угла равен 0.466308
Секанс угла равен 2.3662
Косеканс угла равен 1.10338
Котангенс угла равен 2.14451

```

### Пример 1. Описание класса.

Рассмотрим описание класса *Sum*, который обеспечивает суммирование двух целых чисел. Компонентами класса являются: два слагаемых *x* и *y*, сумма *s* и методы *getx()*, *gety()*, *summa()*, которые предназначены для инициализации компонентных данных *x* и *y*, а также для получения и вывода на экран компьютера результата.

Для того чтобы в классе *Sum* элементы данных определить *собственными*, а методы *общедоступными*, описание класса можно записать следующим образом:

```

class Sum {
int x,y,s;           // по умолчанию private
public:

```

```

void getx(int x1) {x=x1;} // описание метода
void gety(int y1) {y=y1;} // описание метода
void summa ();          // прототип метода
};
void Sum::summa() {// Описание метода:
s=x+y;
cout << "\n Сумма " << x << " и " << y << " равна:" << s; }

```

В приведенном классе `Sum` компонентные данные `x`, `y` и `s` являются собственными по умолчанию, а методы `getx()`, `gety()` и `summa()` общедоступными.

В описании класса методы `getx()` и `gety()` представлены полностью, а метод `summa()` — своим прототипом. Методы `getx()` и `gety()` обеспечивают ввод компонентных данных `x` и `y` соответственно, так как доступ к элементам данных класса можно обеспечить только с помощью методов класса `Sum`. Другим функциям компонентные данные недоступны, т. к. данные `x` и `y` имеют статус доступа *private*.

Описания методов `getx()` и `gety()` размещены внутри класса. Такая форма описания делает метод встроенным (*inline*) по умолчанию. В этом случае тело метода будет размещено в самом классе в виде макрорасширения. Этим достигается экономия времени реализации метода при вызове функции и выходе из нее. Эту форму описания следует использовать лишь для небольших функций. Второй способ описания метода заключается в том, что внутри класса записывается прототип, а описание метода размещается в произвольном месте программы вне тела класса. В приведенном примере таким образом описан метод `summa()`.

### Пример 2. Создание и использование объектов.

Пусть требуется составить программу, выполняющую суммирование двух произвольных чисел.

```

// Сумма двух целых чисел
#include <iostream.h>
class Sum
{
int x,y,s;          // по умолчанию private
public:
void getx(int x1) {x=x1;} // описание метода
void gety(int y1) {y=y1;} // описание метода
void summa();        // прототип метода:
};
void Sum::summa()
{
s=x+y;
cout << "\n Сумма " << x << " и " << y << " равна:" << s; }
void main () {
Sum z,*b=&z;
int x2,y2;
cout << "\n Введите первое слагаемое:";
cin >> x2;
cout << "\n Введите второе слагаемое:";
cin >> y2;
z.getx(x2);
z.gety(y2);
b->summa();
// cout << "\n Сумма " << z.x << " и " << z.y << " равна:" << z.s; }

```

В данной программе введен тип `Sum`, компонентами которого являются три элемента данных `x`, `y`, `s` и три метода `putx()`, `puty()`, `summa()`. В функции `main()` объявлены объект `z` типа `Sum` и указатель `b` на объекты типа `Sum`, инициализированный адресом объекта `z`. Обращения к методам объекта `z`: `z.getx(x2)`; и `z.gety(y2)`; присваивают значения `x2` и `y2` элементам данных `x` и `y` объекта `z` соответственно. Вызов `b->summa()`; метода `summa()` выполняет вычисление суммы значений элементов `x` и `y` объекта `z` и вывод результата на экран. В предпоследней строке записан в качестве комментария оператор, позволяющий выводить на экран информацию о значении данных объекта `z` в случае, если доступ к компонентным данным класса `Sum` будет изменен на *public*. В настоящий момент с этими данными могут работать только методы своего класса, т. к. статус доступа у данных *private*.

В следующей программе продемонстрировано создание простейших конструктора и деструктора. В данном случае они лишь сигнализируют соответственно о создании и удалении объекта класса `coins`. Обе функции вызываются автоматически: конструктор — в строке `coins cash_in_cents`; а деструктор — после завершения функции `main()`.

```

//Пример.
//coins.cpp
//В этой программе на языке C++ демонстрируется создание конструктора и
деструктора.
//Данная программа вычисляет, каким набором монет
//достоинством 25, 10, 5 и 1 копейка можно
//представить заданную денежную сумму.
//
#include <iostream.h>
const int QUARTER = 25;
const int DIME = 10;
const int NICKEL = 5;

class coins {
int number;

public:
coins()
{cout << "Начало вычислений.\n"; } //
конструктор
~coins ()
{cout << "\nКонец вычислений."; } // деструктор
void get_cents(int);
int quarter_conversion(void);
int dime_conversion(int);
int nickel_conversion(int);
};

void coins::get_cents(int cents)
{
number = cents;
cout << number << " копеек состоит из таких монет:" << endl;
}

int coins::quarter_conversion ()
{
cout << number / QUARTER << " – достоинством 25, ";
return(number % QUARTER);
}

int coins::dime_conversion(int d)
{
cout << d / DIME << " – достоинством 10, ";
return(d % DIME);
}

int coins::nickel_conversion(int n)
{
cout << n / NICKEL << " – достоинством 5 и ";
return (n % NICKEL);
}

main () {
int c, d, n, p;
cout << "Задайте денежную сумму в копейках: ";
cin >> c;

// создание объекта cash_in_cents класса coins
coins cash_in_cents;

cash_in_cents.get_cents(c);

```

```

d = cash_in_cents.quarter_conversion();
n = cash_in_cents.dime_conversion(d);
p = cash_in_cents.nickel_conversion(n);
cout << p << " - достоинством 1.";
return (0);
}

```

Вот как будут выглядеть результаты работы программы:

Задайте денежную сумму в копейках: 159

Начало вычислений.

159 копеек состоит из таких монет:

6 — достоинством 25,

0 — достоинством 10,

1 — достоинством 5 и

4 — достоинством 1.

Конец вычислений.

В функции `get_cents()` заданная пользователем денежная сумма записывается в переменную `number` класса `coins`. Функция `quarter_conversion()` делит значение `number` на 25 (константа `QUARTER`), вычисляя тем самым, сколько монет достоинством 25 копеек "умещается" в заданной сумме. В программу возвращается остаток от деления, который далее, в функции `dime_conversion()`, делится уже на 10 (константа `DIME`). Снова возвращается остаток, на этот раз он передается в функцию `nickel_conversion()`, где делится на 5 (константа `NICKEL`). Последний остаток определяет количество однокопеечных монет.

### Пример 5. Использование конструктора и деструктора.

Пусть требуется составить программу, реализующую вычисления по следующей формуле:

$$s = axb + cxk + axc.$$

```

#include <iostream.h>
struct Pro
{
private:
int x,y,z;
public:
// Прототипы методов:
Pro (int,int);           //конструктор
int putx();              //доступ к x
int puty();              //доступ к y
int putz();              //доступ к z
void proizv();           //произведение
~Pro();                  //деструктор
};
// Описания методов:
Pro::Pro(int x1,int y1) {x=x1; y=y1;}
int Pro::putx() {return x;}
int Pro::puty() {return y;}
int Pro::putz() {return z;}
void Pro::proizv() {z=x*y;}
Pro::~~Pro() { }
void main() {
int s,a,b,c,k;
cout << "\n Введите a,b,c и k \n";
cin >> a >> b >> c >> k;
Pro D = Pro(a,b);        //создание и инициализация объекта D
Pro E(c,k);              //создание и инициализация объекта E
Pro F(a,c);              //создание и инициализация объекта F
D.proizv();              //получение произведения a*b
E.proizv();              //получение произведения c*k
F.proizv();              //получение произведения a*c
cout << "\n D.a=" << D.putx();
cout << "\n D.b=" << D.puty();
cout << "\n D.z=" << D.putz();
s=D.putz()+E.putz()+F.putz();
cout << "\n s=" << s;
}

```

```

F.Pro::~~Pro();           //уничтожение объекта F
E.Pro::~~Pro();           //уничтожение объекта E
D.Pro::~~Pro();           //уничтожение объекта D
}

```

В примере для получения произведения создан класс *Pro*, компонентными данными которого являются сомножители  $x$  и  $y$ , а также элемент  $z$ , предназначенный для хранения произведения. Для доступа к компонентным данным используются три метода *putx()*, *puty()* и *putz()*, а для получения произведения *proizv()*. Кроме того, для класса определены конструктор *Pro()* и деструктор *~Pro()* — в классе содержатся их прототипы.

Описания всех методов, в том числе конструктора и деструктора, вынесены за пределы описания класса. В результате трех явных вызовов конструктора созданы и инициализированы объекты *D*, *E* и *F*. В результате работы метода *proizv()* получены три значения произведений для своих объектов. Результатом работы программы является получение суммы этих значений. В конце программы три раза вызван деструктор для уничтожения созданных ранее объектов.

#### Пример 6.

```

class string_operation
{
char *stringl;
int string_len;
public:
string_operation (char*)
{ stringl = new char[string_len]; }
~string_operation()
{ delete stringl; }
void input_data (char*);
void output_data (char*);
};

```

Память, выделенная для указателя *stringl* с помощью оператора *new*, может быть освобождена только оператором *delete*. Поэтому если конструктор класса выделяет память для какой-нибудь переменной, важно проследить, чтобы в деструкторе эта память обязательно освобождалась.

Области памяти, занятые данными базовых типов, таких как *int* или *float*, освобождаются системой автоматически и не требуют помощи конструктора и деструктора.

#### Пример создания и использования класса

Рассмотрим класс *Point*, который позволяет сформировать точку на экране компьютера. Поместим описание класса в файл с именем *point.h*

```

//Файл point.h
#ifndef POINT_H
#define POINT_H 1
class Point          //класс для определения точки на экране
{
protected:         //защищенный статус доступа к элементам данных
int x;              //координата x точки
int y;              //координата y точки
// Прототипы методов:
public:              //общедоступный статус доступа
Point (int, int);   //конструктор
int putx();          //доступ к x
int puty();          //доступ к y
void show();         //изобразить точку на экране
void move (int,int); //переместить точку
private:             //собственный статус доступа
void hide();         //убрать изображение точки
};
#endif

```

Поскольку описание класса *Point* планируется использовать при описании других классов, то для предотвращения недопустимого дублирования описания класса в текст включены три директивы препроцессора *#ifndef POINT\_H*, *#define POINT\_H 1* и *#endif*.



Компонентами класса *Point* являются два элемента данных *x* и *y* с защищенным статусом доступа, пять общедоступных методов и один метод с собственным статусом доступа. Методы в описании класса представлены своими прототипами.

Выполним внешнее описание методов класса, разместив описания в файле *point.cpp*:

```
//Файл point.cpp - описание методов
#ifndef POINT_CPP
#define POINT_CPP 1
#include <graphics.h> // прототипы функций графической библиотеки
#include "point.h" // описание класса Point
Point::Point (int x1=0, int y1=0) // Конструктор:
{
    x=x1; y=y1;
}
//Метод доступа к x:
int Point::putx()
{return x;}
//Метод доступ к y:
int Point::puty()
{return y;}
//Метод изображения точки на экране:
void Point::show(void)
{putpixel(x,y,getcolor())};
//Метод удаления точки с экрана:
void Point::hide(void)
{putpixel(x,y,getbkcolor())};
//Метод перемещения точки в новое место экрана:
void Point::move(int xn=0, int yn=0)
{
    hide();
    x=xn;
    y=yn;
    show();
}
#endif
```

**Достоинство** внешнего описания методов класса состоит в том, что оно позволяет при необходимости модифицировать содержание методов, причем эти изменения останутся незамеченными для остальных частей программы.

Программа, содержащая в своем составе главную функцию, будет выглядеть следующим образом:

```
// Точка на экране
#include <iostream.h>
#include <graphics.h> //прототипы графических функций
#include <conio.h> //прототип функции getch()
#include "point.cpp" //описание класса Point

void main ()
{
    Point t(100,150); //создана невидимая точка t(x=100,y=150)
    Point t1(200,200); //создана невидимая точка t1(X=200,Y=200)
    //Инициализация графики
    int a=DETECT,b;
    initgraph(&a,&b,"");
    t.show(); //показать точку t
    cout << "\n коор-ты точки t: x=" << t.putx() << ",y=" << t.puty(); getch();
    //ждать нажатия клавиши
    t1.show(); //показать точку t1
    cout<<"\n координаты точки t1: x="<<t1.putx()<<",y="<<t1.puty();
    getch ();
    t.move(150,300); //переместить точку t(x=150,y=300)
    cout<<"\n новые координаты t: x="<<t.putx()<<",y="<<t.puty();
    getch();
    closegraph (); //закрыть графический режим
```

}

В программе используется функция *getch()*, которая позволяет остановить выполнение программы до нажатия на клавиатуре любой клавиши. Прототип этой функции находится в файле *conio.h*.

## НАСЛЕДОВАНИЕ

### Статусы доступа производных классов

Статус доступа в базовом классе	Модификатор доступа	Статус доступа в производном классе	
		struct	class
public	-	public	private
protected	-	protected	private
private	-	недоступны	недоступны
public	public	public	public
protected	public	protected	protected
private	public	недоступны	недоступны
public	protected	protected	protected
protected	protected	protected	protected
private	protected	недоступны	недоступны
public	private	private	private
protected	private	private	private
private	private	недоступны	недоступны

В следующей программе показан пример создания открытых производных классов. Родительский класс *consumer* содержит общие данные о клиенте: имя, адрес, город, штат и почтовый индекс. От базового класса порождаются два производных класса: *airline* и *rental\_car*. Первый из них хранит сведения о расстоянии, покрытом клиентом на арендованном самолете, а второй — сведения о расстоянии, покрытом на арендованном автомобиле.

```
//Пример 7. derclass.cpp
// Эта программа на языке C++ демонстрирует использование производных классов.
//
#include <iostream.h>
#include <string.h>
char newline;
class consumer
{
    char name[60],
        street[60],
        city[20],
        state[15],
        zip [10];
    public:
    void data_output(void);
    void data_input(void) ;
};
void consumer::data_output()
{
    cout << "Имя:"<< name << endl;
    cout << "Улица: "<< street << endl;
    cout << "Город: "<< city << endl;
    cout << "Штат: "<< state << endl;
    cout << "Индекс: "<< zip << endl;
    void consumer::data_input()
    {
```

```

cout << "Введите полное имя клиента: ";
cin.get (name, 59, '\n');
cin.get (newline); // пропуск символа новой строки
cout << "Введите адрес: ";
cin.get (street, 59, '\n'); cin.get (newline);
cout << "Введите город: ";
cin.get (city, 19, '\n'); cin.get (newline);
cout << "Введите название штата: " ;
cin.get (state, 14, '\n'); cin.get (newline);
cout << "Введите почтовый индекс: ";
cin.get (zip, 9, '\n'); cin.get (newline);
}
class airline : public consumer
{
char airline_type [20] ;
float acc_air_miles;
public:
void airline_consumer();
void disp_air_mileage();
}
void airline::airline_consumer()
{
data_input();
cout << "Введите тип авиалиний:";
cin.get (airline_type, 19, '\n'); cin.get (newline);
cout << "Введите расстояние, покрытое в авиарейсах:";
cin >> acc_air_miles; cin.get (newline);
}
void airline: :disp_air_mileage ()
{
data_output();
cout << "Тип авиалиний:" << airline_type << endl;
cout << "Суммарное расстояние: " << acc_air_miles << endl;
}
class rental_car : public consumer
{
char rental_car_type[20];
float acc_road_miles;
public:
void rental_car_consumer();
void disp_road_mileage();
};
void rental_car::rental_car_consumer()
{
data_input();
cout << "Введите марку автомобиля:";
cin.get (rental_car_type, 19, '\n'); cin.get (newline);
cout << "Введите суммарный автопробег:";
cin >> acc_road_miles; cin.get (newline);
}
void rental_car::disp_road_mileage()
{
data_output();
cout << "Марка автомобиля:" << rental_car_type << endl;
cout << "Суммарный автопробег: " << acc__road_mile << endl;
}
int main()
{
airline cons1;
rental_car cons2;
cout << "\n--Аренда самолета --\n";

```

```

cons1.airline_consumer();
cout << "\n--Аренда автомобиля--\n" ;
cons2.rental_car_consumer() ;
cout << "\n--Аренда самолета --\n";
cons1.disp_air_mileage();
cout << "\n--Аренда автомобиля--\n";
cons2.disp_road_mileage();
return (0) ;
}

```

Переменные родительского класса *consumer* недоступны дочерним классам, так как являются закрытыми. Поэтому для работы с ними созданы функции *data\_input()* и *data\_output()*, которые помещены в открытую часть класса и могут быть вызваны в производных классах.

### Пример. Одиночное наследование классов.

Требуется составить программу, которая позволяет получить на экране окружность. Для иллюстрации механизма наследования на основе класса *Point* построим производный класс *Circle* (окружность). Для производного класса из класса *Point* выберем следующие элементы данных и методы:

- *int x* – координата *x* точки;
- *int y* – координата *y* точки;
- *int putx()* – доступ к *x*;
- *int puty()* – доступ к *y*.

Дополнительно для класса *Circle* введем следующие компоненты:

- *int radius* – радиус окружности;
- *int vis* – индикатор видимости окружности на экране;
- *void show()* – изобразить окружность на экране;
- *void hide()* – убрать изображение окружности;
- *void move()* – переместить окружность на новое место экрана;
- *void vary()* – изменить размер окружности;
- *int putradius()* – обеспечить доступ к радиусу окружности.

В соответствии с изложенным создадим класс *Circle* и его описание поместим в файл *circle.h*:

```

// Файл circle.h – описание производного класса
#include "point.cpp" // описание класса Point
class Circle:public Point//класс для определения окружности
{ // модификатор publicпозволяет сделать элементы x и y класса
// Point в описании класса Circle защищенными (protected) protected:
    //защищенный статус доступа к элементам данных
    int radius; //радиус окружности
    int vis; //видимость окружности на экране
//Прототипы методов:
public: //общедоступный статус доступа
    Circle(int,int,int); //конструктор
    ~Circle(); //деструктор
    void show(); //изобразить окружность на экране
    void hide(); //убрать изображение окружности
    void move (int,int); //переместить окружность
    void vary(int); //изменить размер окружности
    int putradius(); // доступ к радиусу

```

В созданном классе *Circle* явно определены конструктор *Circle()* и деструктор *~Circle()*. Из класса *Point* наследуются два метода: *putx()* и *puty()*, так как методы *show()* и *move()* заменяются одноименными методами класса *Circle*, а метод *hide()* не наследуется, поскольку имеет статус доступа *private*.

Используем внешние описания методов, поместив их в отдельный файл с именем *circle.cpp*:

```

//Файл circle.cpp
#include <graphics.h> //прототипы функций графической библиотеки #include
"circle.h" // описание класса Circle

//Описания методов класса Circle::

//Конструктор
Circle::Circle (int xc,int yc,int radc=0) : Point(xc,yc)
{

```

```

radius=radc;
vis=1;
}

//Изобразить окружность на экране
void Circle::show(void)
{
circle(x,y,radius); //рисование окружности
}

//Убрать окружность с экрана
void Circle::hide(void)
{
unsigned int col;          //объявление переменной для текущего цвета
if (vis==0) return;        //окружности нет
col=getcolor();            //запоминание текущего цвета setcolor(getbkcolor());
    //выявление текущего цвета фона
circle(x,y,radius);        //рисование окружности цветом фона
vis=0;
setcolor(col);             //восстановление текущего цвета
}

//Переместить окружность в новое место экрана
void Circle::move(int xn, int yn)
{
hide();                    //стирание старой окружности
x=xn;
y=yn;
show();                    //рисование новой окружности
}

//Изменить размер окружности
void Circle::vary(int d)
{
hide();                    //стирание старой окружности
radius+=d;                 //изменение радиуса
if(radius<0) radius=0;
show();                    //рисование новой окружности
}

// Доступ к радиусу
int Circle::putradius()
{
return radius;
}

//Деструктор
Circle::~Circle()
{
hide();                    //стирание окружности
}

```

Конструктор *Circle()* имеет три параметра: координаты центра (*xc*, *yc*) и радиус окружности *radc*. При создании объекта класса *Circle* вначале вызывается конструктор класса *Point*, который по значениям фактических параметров определяет центр окружности. Эта точка создается как объект класса *Point* без имени. Затем выполняется тело конструктора *Circle()*.

Деструктор *~Circle()* уничтожает окружность и вызывает деструктор базового класса *~Point()*, который, несмотря на отсутствие его описания в классе, формируется компилятором по умолчанию.

Программа для работы с объектами класса *Circle*.

```

#include <iostream.h>
#include <graphics.h>
#include <conio.h>
#include "circle.cpp"          //описание класса Circle

```

```

void main()
//Инициализация графики
int a=ДЕТЕКТ,b;
initgraph(&a,&b,"");
Circle A(150,250,30);           //создана невидимая окружность A
Circle B(300,100,50);          //создана невидимая окружность B
cout<<"\n A:x="<<A.putx()<<" ,y="<<A.puty()<<" ,рад="<<A.putradius();
cout<<"\n B:x="<<B.putx()<<" ,y="<<B.puty()<<" ,рад="<<B.putradius();
getch();                       //ждать нажатия клавиши
A.show();                      //показать на экране окружность A
getch();                       //ждать нажатия клавиши
B.show();                      //показать на экране окружность B
getch();                       //ждать нажатия клавиши
Point C(300,100);              //создана точка C
C.show();                      //показать на экране точку C
getch();                       //ждать нажатия клавиши
A.move(150,150);               //переместить окружность A
cout<<"\n A:x="<<A.putx()<<" ,y="<<A.puty()<<" ,рад="<<A.putradius();
getch();
B.vary(20);                    //изменить размеры окружности B
cout<<"\n B:x="<<B.putx()<<" ,y="<<B.puty()<<" ,рад="<<B.putradius();
getch();
A.Circle::~~Circle();          //уничтожение объекта A
B.Circle::~~Circle();          //уничтожение объекта B
closegraph();                  //закрыть графический режим
}

```

В результате выполнения программы вначале конструктором *Circle()* создаются два невидимых объекта: окружность *A* с координатами (150, 250) и радиусом 30 и окружность *B* с координатами (300, 100) и радиусом 50. В результате двукратной работы метода *show()* объекты *A* и *B* становятся видимыми. При явном вызове конструктора *Point()* создается объект *C* с координатами центра окружности *B*. В результате работы метода *show()* класса *Point* точка *C* становится видимой. Метод *move()* стирает старую окружность *A*, изменяет в объекте *A* значения защищенных координат объекта (150,150) (функции, которые не принадлежат к классам *Point* и *Circle*, этого сделать не могут) и рисует окружность прежнего радиуса на новом месте экрана. Метод *vary()* стирает окружность объекта *B* с экрана (при этом объект *B* не уничтожается) и после изменения значения радиуса рисует окружность с другим радиусом.

В качестве примера множественного наследования рассмотрим производный класс *Reccircle* — "прямоугольник и окружность". Для описания класса *Reccircle* создадим новый непосредственный базовый класс *Rectangle* — "прямоугольник" и используем ранее созданный класс *Circle* — "окружность". Описание класса *Rectangle* поместим в отдельный файл *rectang.h*:

```

//Файл rectang.h - описание класса прямоугольник
#include <graphics.h>
#include "point.cpp"           //описание класса Point
class Rectangle:public Point   //класс для определения прямоугольника
{
protected:                   //защищенный статус доступа к элементам данных
int lx;                       //длина по горизонтали
int ly;                       //длина по вертикали
// Прототипы методов:
void risrec();                //прорисовка прямоугольника
public:                       //общедоступный статус доступа
Rectangle(int,int,int,int);   //конструктор
void show(void);              //прорисовка прямоугольника
void hide(void);              //убрать изображение прямоугольника с экрана
};

```

Непосредственным базовым классом для класса *Rectangle* является класс *Point*. Кроме наследуемых элементов данных *x* и *y* (координаты центра прямоугольника) класс *Rectangle* в своем составе имеет два элемента данных *lx* — длина по горизонтали и *ly* — длина по вертикали. Четыре метода класса позволяют создавать объекты, их инициализировать, получать изображение на экране и убирать его в случае необходимости. Описания методов класса *Rectangle* разместим в отдельном файле *rectang.cpp*:

```

//Файл rectang.cpp - описания методов класса "прямоугольник"
#include <graphics.h>

```

```

#include "rectang.h"                                //описание класса Point

//Описания методов:
void Rectangle::risrec()                            //прорисовка прямоугольника
{
    int g=lx/2;
    int v=ly/2;
    line(x-g,y-v,x+g,y-v);
    line(x-g,y+v,x+g,y+v);
    line(x-g,y-v,x-g,y+v);
    line(x+g,y-v,x+g,y+v);
}

// Конструктор:
Rectangle::Rectangle(int xi,int yi,int lxi=0,int lyi=0):Point(xi,yi)
{
    lx=lxi;
    ly=lyi;
}

//Изобразить прямоугольник на экране:
void Rectangle::show(void)
{
    risrec();                                        //прорисовка прямоугольника
}

//Убрать изображение прямоугольника с экрана:
void Rectangle::hide(void)
{
    int b,c;
    b=getbkcolor();                                //запоминание текущего цвета фона
    c=getcolor();                                  //запоминание цвета изображения
    setcolor(b);                                    //установить цвет изображения
    risrec();                                        //нарисовать прямоугольник цветом фона
    setcolor(c);                                    //восстановление цвета изображения
}

```

Теперь можно создать класс "прямоугольник и окружность" на основании двух непосредственных базовых классов. Описание класса *Reccircle* поместим в отдельный файл *reccirc.h*:

```

//Файл reccirc.cpp - описание класса "прямоугольник и окружность"
#include "rectang.cpp"                                // описание класса Rectangle
#include "circle.cpp"                                // описание класса Circle
//Класс "прямоугольник и окружность":
class Reccircle:public Rectangle, public Circle
{
public:

//Конструктор:
Reccircle(int xi,int yi,int ri,int lxi,int lyi) : Rectangle(xi,yi,lxi,lyi),
Circle(xi,yi,ri)
{ }                                                    //Явный вызов конструкторов базовых классов

//Изобразить прямоугольник и окружность на экране:
void show(void)
{
    Rectangle::show();                                //изобразить прямоугольник
    Circle::show();                                    //изобразить окружность
}

//Убрать изображение с экрана:
void hide(void)
{

```

```

Rectangle::hide();           //убрать изображение прямоугольника
Circle::hide();             //убрать изображение окружности
}
};

```

В классе описано три метода: конструктор, который явным образом вызывает конструкторы непосредственных базовых классов, и методы изображения на экране и стирания с экрана прямоугольника и окружности.

Теперь можно составить программу, которая позволит работать с объектами класса *Reccircle*.

```

// Пример множественного наследования
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
#include "reccirc.cpp"          //описание класса Reccircle
void main () {
// Инициализация графики
int a=DETECT,b;
initgraph(&a,&b,"");
Reccircle A(250,100,100,50,50); //создан невидимый объект A
Reccircle B(400,300,50,120,140); //создан невидимый объект B
cout<<"\n A: x="<<A.Rectangle::putx()<<" y="<<A.Rectangle::puty();
cout<<"\n B: x="<<B.Circle::putx()<<" y="<<B.Circle::puty();
getch();
A.show();                      //показать на экране объект A
getch();
B.show();                      //показать на экране объект B
getch();
A.hide();                     //стереть объект A
getch();
A.Circle::show();             //показать на экране окружность объекта A
getch();
B.hide();                     //стереть объект B
getch();
B.Rectangle::show();          //показать на экране прямоугольник объекта B
getch();
closegraph();                 //закрыть графический режим
}

```

В программе посредством вызова конструктора формируются два объекта *A* и *B* с координатами центров (250,100) и (400,300) соответственно. В результате работы методов *A.show()* и *B.show()* на экране высвечиваются объекты *A* и *B*, каждый из которых состоит из прямоугольника и окружности. Размеры прямоугольника для объекта *A*: 50 — по горизонтали, 50 — по вертикали, а для *B*: 120 — по горизонтали, 140 — по вертикали. Радиус окружности объекта *A* — 100, а *B* — 50. После стирания с экрана объекта *A* с помощью оператора *A.Circle::show()*; на экране высвечивается окружность объекта *A*. Подобным же образом высвечивается на экран прямоугольник объекта *B*.

### Класс в классе

Компонентами класса может быть класс. Если *class1* содержит компонент класса *class2*, то в классе *class1* должен быть объявлен конструктор по умолчанию *class2::class2()*, чтобы компонент класса *class2* мог быть построен.

```

//класс в классе
#include<iostream.h>
class A //первый класс
{
    int a;
public:
    int b;
    A(){} //обязательно для поддержки класса в классе
    A(int a1,int b1){a=a1;b=b1;} //инициализация с конструктором
    ~A(){} //можно не объявлять
    void init(int a1,int b1){a=a1;b=b1;} //инициализация без конструктора
    void geta(){cout<<"a="<<a<<endl;}
    void getb();
};

```



```

class B //второй класс, содержащий в себе первый класс
{
public:
    int c;
    class A s; //можно писать без слова class, т.е. так: A s
    B(int); //конструктор
};
void A::getb(){cout<<"b="<<b<<endl;}
B::B(int c1) {c=c1;
//так A s(0,0); нельзя (безрезультатно)
//вызов конструктора внутреннего класса не проходит
    s.init(0,0); //нужно так
}

void main()
{
cout<<endl<<"start"<<endl;
A a(1,2); //создаем объект а класса А с параметрами 1 и 2
a.geta();
a.getb();
B b(4), *z=&b; //создаем объект b класса В и указатель на объект z
b.c=0;
b.s.b=9;
b.s.geta();
z->s.getb();
cout<<"stop"<<endl;
}
Класс в классе может быть реализован и с помощью struct.
#include<iostream.h>
struct A
{
    int a;
public:
    int b;
//A(int a1,int b1){a=a1;b=b1;}
    ~A(){}
    void geta(){cout<<"a="<<a;}
    void getb();
}qq;

struct B
{
public:
    int c;
    struct A s;
}zz;
void A::getb(){cout<<"b="<<b;}
void main()
{
A a;
a.geta();
a.getb();
B b;
b.c=0;
qq.a=5;
zz.c=7;
zz.s.b=9;
}

```

## Дружественные функции

Переменные-члены классов, как правило, являются закрытыми, и доступ к ним можно получить только посредством функций-членов своего же класса. Может показаться странным, но существует категория функций, создаваемых специально для того, чтобы преодолеть это ограничение. Эти функции, называемые дружественными и объявляемые в описании класса с помощью ключевого слова `friend`, получают доступ к переменным-членам класса, сами не будучи его членами.

```
//
//friend.cpp
//Эта программа на языке С++ демонстрирует использование
//дружественных функций. Программа получает от системы
//информацию о текущей дате и времени и вычисляет
//количество секунд, прошедших после полуночи.
//
#include <iostream.h>
#include <time.h> // содержит прототипы функций time(),
//localtime(), asctime(), а также описание структур tm и time_t

class time_class {
long secs;
friend long present_time(time_class);
//дружественная функция
public:
time_class(tm *) ;
};
time_class::time_class(tm* timer)
{
secs = timer->tm_hour*3600 + timer->tm_min*60 + timer->tm_sec;
}
long present_time(time_class); // прототип
int main ()
{
// получение данных о дате и времени от системы
time_t ltime;
tm *ptr;
time(&ltime);
ptr = localtime(&ltime);
time_class tz(ptr);
cout<<"Текущие дата и время:"<<asctime(ptr)<<endl;
cout<<"Число секунд после полуночи:"<<present_time(tz)<<endl;
return(0);
}
long present_time(time_class tz)
{
return(tz.secs); }
```

Рассмотрим подробнее процесс получения данных о дате и времени.

```
time_t ltime;
tm *ptr;
time (&ltime);
ptr = localtime(&ltime);
```

Сначала создается переменная `ltime` типа `time_t`, которая инициализируется значением, возвращаемым функцией `time ()`. Эта функция вычисляет количество секунд, прошедших с даты 1 января 1970 г. 00:00:00 по Гринвичу, в соответствии с показаниями системных часов. Тип данных `time_t` специально предназначен для хранения подобного рода значений. Он лучше всего подходит для выполнения такой операции, как сравнение дат, поскольку содержит, по сути, единственное число, которое легко сравнивать с другим аналогичным числом.

Далее создается указатель на структуру `tm`, предназначенную для хранения даты в понятном для человека виде:

```
struct tm {
int tm_sec;           //секунды (0–59)
int tm_min;           //минуты (0–59)
int tm_hour;          //часы (0–23)
```

```

int tm_mday;           //день месяца (1–31)
int tm_mon;           //номер месяца (0–11; январь = 0)
int tm_year;          //год (текущий год минус 1900)
int tm_wday;          //номер дня недели (0–6; воскресенье = 0)
int tm_yday;          //день года (0–365; 1-е января = 0)
int tm_isdst;         //положительное число, если осуществлен переход
//на летнее время;
//0, если летнее время еще не введено;
//отрицательное число, если информация
//о летнем времени отсутствует
};

```

Функция `localtime ()`, в качестве аргумента принимающая адрес значения типа `time_t`, возвращает указатель на структуру `tm`, содержащую информацию о текущем времени в том часовом поясе, который установлен в системе. Имеется схожая с ней функция `gmtime ()`, вычисляющая текущее время по Гринвичу.

Полученная структура передается объекту `tz` класса `time_class`, а точнее — конструктору этого класса, вызываемому в строке

```
time_class tz(ptr);
```

В конструкторе из структуры `tm` извлекаются поля `tm_sec`, `tm_min` и `tm_hour` и по несложной формуле вычисляется количество секунд, прошедших после полуночи.

```
secs = timer->tm_hour*3600 + timer->tm_min*60 + timer->tm_sec;
```

Функция `asctime ()` преобразует структуру `tm` в строку вида  
*день месяц число часы:минуты:секунды год\n\0*

Например:

```
Mon Aug 10 13:12:21 1998
```

Дружественная классу `time_class` функция `present_time ()` получает доступ к переменной `secs` этого класса и возвращает ее в программу.

Программа выводит на экран две строки следующего вида:

```
Текущие дата и время: Mon Aug 10 09:31:14 1998
```

```
Число секунд, прошедших после полуночи: 34274
```

Еще один пример дружественных функций: функция, позволяющая получать доступ к собственным компонентам класса и изменять их.

```

#include<iostream.h>
class A
{
    int a;
    friend void my(A&);
public:
    A(int a1){a=a1;}
    void print(){cout<<a<<endl;}
};

```

```
void my(A& c) {c.a=5;} //изменяем другим функциям и вызовам недоступное поле
```

```

void main()
{
    A a=A(1);           //создаем объект a на класс A
    a.print();          //печатаем собственный элемент данных a.a (1)
    my(a);              //изменяем недоступное поле объекта через вызов //дружественной
функции
    a.print();          //печатаем a.a (5)
}

```

Перегрузка функций – членов класса

//Пример 1.

//absolute.cpp

//Эта программа на языке C++ демонстрирует использование перегруженных функций-членов класса. Программа вычисляет абсолютное значение чисел типа `int` и `double`.

//

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

```
#include <math.h> // содержит прототипы функций abs() и fabs()
```

```

class absolute_value
{
public:
int number(int);
double number(double);
};
int absolute_value::number(int test_data)
{
return(abs(test_data));
}
double absolute_value::number(double test_data)
{
return(fabs(test_data));
}
int main ()
{
absolute_value neg_number;
cout<<"|-583|="<<neg_number.number(-583)<< endl;
cout<<"|-583.1749|="<<neg_number.number(-583.1749) << endl;
return(0);
}
Вот какими будут результаты работы программы:
|-583|=583
|-583.1749|=583.175

```

В приведенной ниже программе в функцию `trig_calc ()` передается значение угла в одном из двух форматов: числовом или строковом. Программа вычисляет синус, косинус и тангенс угла.

```

//Пример 2.
//overload.cpp
//Эта программе на языке C++ содержит пример перегруженной функции,
//принимающей значение угла как в числовом виде, так и в формате
//градусы/минуты/секунды.
//
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
const double DEG_TO_RAD=0.0174532925;
class trigonometric
{
double angle;
public:
void trig_calc(double);
void trig_calc(char *);
};
void trigonometric::trig_calc(double degrees)
{
angle = degrees;
cout << "\n Для угла " << angle << " градусов:" << endl;
cout << "синус равен " << sin(angle * DEG_TO_RAD)<< endl;
cout << "косинус равен " << cos (angle * DEG_TO_RAD)<< endl;
cout << "тангенс равен " << tan(angle * DEG_TO_RAD)<< endl;
}
void trigonometric::trig_calc(char *dat)
{
char *deg, *min, *sec;
deg = strtok(dat, "d");
min = strtok(0, "m");
sec = strtok(0, "s");
angle = atof(deg) + atof(min)/60.0 + atof (sec)/360.0;
cout << "\n Для угла " << angle << " градусов:" << endl;
cout << "синус равен " << sin (angle * DEG_TO_RAD)<< endl;

```

```

cout << "косинус равен " << cos(angle * DEG_TO_RAD)<< endl;
cout << "тангенс равен " << tan (angle * DEG_TO_RAD)<< endl;
}
int main ()
{
    trigonometric data;
    data.trig_calc(75.0);
    char str1[] = "35d 75m 20s";
    data.trig_calc(str1);
    data.trig_calc(145.72);
    char str2[] = "65d 45m 30s";
    data.trig_calc(str2);
    return(0);
}

```

В программе используется библиотечная функция `strtok()`, прототип которой находится в файле `STRING.H`. Эта функция сканирует строку, разбивая ее на лексемы, признаком конца которых служит один из символов, перечисленных во втором аргументе. Длина каждой лексемы может быть произвольной. Функция возвращает указатель на первую обнаруженную лексему. Лексемы можно читать одну за другой путем последовательного вызова функции `strtok()`. Дело в том, что после каждой лексемы она вставляет в строку символ `\0` вместо символа-разделителя. Если при следующем вызове в качестве первого аргумента указать `0`, функция продолжит чтение строки с этого места. Когда в строке больше нет лексем, возвращается нулевой указатель.

Рассмотренная программа позволяет задавать значение угла в виде строки с указанием градусов, минут и секунд. Признаком конца первой лексемы, содержащей количество градусов, служит буква `d`, второй лексемы - `m`, третьей - `s`. Каждая извлеченная лексема преобразуется в число с помощью стандартной библиотечной функции `atof()` из файла `stdlib.h`.

Ниже представлены результаты работы программы:

```

Для угла 75 градусов:
синус равен 0.965926
косинус равен 0.258819
тангенс равен 3.73205
Для угла 36.3056 градусов:
синус равен 0.592091
косинус равен 0.805871
тангенс равен 0.734722
Для угла 14 5.72 градусов:
синус равен 0.563238
косинус равен -0.826295
тангенс равен -0.681642
Для угла 65.8333 градусов:
синус равен 0.912358
косинус равен 0.4 09392
тангенс равен 2.22857

```

### Перегрузка стандартных операций

*Перегрузку стандартных операций* можно рассматривать как разновидность перегрузки

Переопределение, и как следствие, перегрузка может выполняться для следующих стандартных операций и встроенных функциональных вызовов:

+	-	*	/	%	^	!	=	<	>	+=	-=	^=
&=	=	<<	>>	<<=	<=	>=	>>=	&&		++	--	()
[]	new	delete	&		~	*=	/=	%=	==	!=	,	->

->\*

---

Не могут быть переопределены следующие операции: `., .*, ?., ::, sizeof`.

```

//Пример 1. Перегрузка операций.
//Определим класс комплексных данных и для этого класса переопределим
//стандартные операции +, ==, <, >.
#include<iostream.h>

```

```

class Complex
{
double real;
double image;
public:
Complex(){real=0;image=0;} //первый конструктор
Complex(double r){real=r;image=0;} //второй конструктор
Complex(double r,double i){real=r;image=i;} //третий конструктор
Complex operator+(Complex&); //перегрузка операции
int operator == (Complex&); //прототип операторной функции ==
int operator > (Complex&); //прототип операторной функции >
int operator < (Complex&); //прототип операторной функции <
void print(){cout<<"real="<<real<<" image="<<image<<"\n";}
~Complex(){} //пустой деструктор
};

//первый вариант перегрузки операции + (плохой вариант, т.к. меняется
значение первого слагаемого)
/*
Complex Complex::operator+(Complex& c)
{real=real+c.real; image=image+c.image; return *this;}
*/
//второй вариант перегрузки операции + (хороший вариант, т.к. значения
слагаемых не изменяются)
Complex Complex::operator+(Complex &c)
{
Complex d;
d.real=real+c.real;
d.image=image+c.image;return d;
}

int Complex::operator==(Complex &c) //определение операторной функции =
{
return ((sqrt(real*real+image*image)== sqrt(c.real*c.real+c.image*c.image))
? TRUE:FALSE);
}
int Complex::operator>(Complex &c) //определение операторной функции >
{
return((sqrt(real*real+image*image)> sqrt(c.real*c.real+c.image*c.image)) ?
TRUE:FALSE);
}
int Complex::operator<(Complex &c) //определение операторной функции <
{
return((sqrt(real*real+image*image)< sqrt(C.real*C.real+C.image*C.image)) ?
TRUE:FALSE);
}

void main()
{
cout<<"Start of test\n";
//создаем три статических объекта типа Complex
Complex c1(1); //использование второго конструктора (1,0)
Complex c2(2,2); //использование третьего конструктора (2,2)
Complex c3; //использование первого конструктора (0,0)
//контрольная печать значений
cout<<"контрольная печать значений\n";
c1.print(); c2.print(); c3.print();
//использование перегруженной операции +
c3=c1+c2; //сложение комплексных чисел
cout<<"печать результата c3=c1+c2\n";
c3.print(); //печать результата
}

```

```
//контрольная печать значений
cout<<"контрольная печать значений\n";
c1.print();c2.print();c3.print();
if (c1==c2) cout<<"c1 equal c2"<<endl;
if (c1<c3) cout<<"c1 < c2"<<endl;
if (c3>c2) cout<<"c3 > c2"<<endl;
//создаем динамический объект типа Complex
Complex *p;           //использование первого конструктора
p=new Complex(17,4);   //создание динамического объекта и его
//инициализация
*p=c2;                //присвоение значений объекта c2
cout<<"контрольная печать значений динамического объекта\n";
p->print();           //печать значений
p->Complex::~~Complex(); //вызов деструктора
delete p;             //уничтожаем объект в памяти
//вызов деструкторов (необязательно)
c1.~Complex(); c2.~Complex(); c3.~Complex();
}
```

В приведенном примере выполнено переопределение стандартных операций  $+$ ,  $==$ ,  $<$  и  $>$  (стандартно используемых для численных данных), что обеспечивает возможность их применения для комплексных данных. Вызов требуемых экземпляров операций будет осуществляться в зависимости от типа операндов (в нашем примере операнды являются комплексными).

Использование знака операции представляет собой *сокращенную форму записи* вызова операторной функции, которая может вызываться как любая другая функция. Например:

```
c1+c2;                //сокращенная форма вызова
c1.operator+(c2);     //полная форма вызова функции-операции
или
c1==c2;               //сокращенная форма вызова
c1.operator==(c2);    //полная форма вызова функции-операции
```

На перегрузку операторов накладываются следующие ограничения:

- невозможно изменить приоритет оператора и порядок группировки его операндов;
- невозможно изменить синтаксис оператора: если, например, оператор унарный, т.е. принимает только один аргумент, то он не может стать бинарным;
- перегруженный оператор является членом своего класса и, следовательно, может участвовать только в выражениях с объектами этого класса;
- перегруженная операция не может изменить поведение применительно к базовым типам данных;
- невозможно создавать новые операторы;
- не могут быть перегружены следующие символы препроцессора:  $\#$  и  $\##$ ;

### Пример 2. Использование перегрузки операций.

Пример, иллюстрирующий преимущества объектно-ориентированного подхода для вычислительных задач, основанных на матричных преобразованиях. Определим класс `matrix` (динамический двумерный массив). Для указанного класса перегрузим стандартные операции умножения, деления, сложения и вычитания. Такая перегрузка операторов позволит в дальнейшем писать программы с матричными вычислениями на C++ подобно тому, как пишутся программы для обработки простых числовых данных. Пример использования класса `matrix` содержится в конце программы. В программе для вывода сообщения об ошибке используется `cerr` – небуферизованный поток для стандартного вывода сообщений об ошибках (один из четырех стандартных потоков ввода-вывода, которые автоматически создаются в программе в начале ее выполнения при подключении файла `iostream.h`).

```
//Файл matrix.cpp
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
const int ERR_EXIT = -1;
const double IS_ZERO = 1E-5;           //если число меньше IS_ZERO,
                                         //оно считается нулевым

class matrix
{
private:
```

```

unsigned int
ROWS,                                //число строк
COLS;                                //число столбцов
float **point; //собственно данные матрицы
protected:
float Det2x2(); //особый случай - матрица 2x2
public:
matrix (unsigned int rows,unsigned int cols); //конструктор
matrix(const matrix &REF);                  //конструктор копирования
~matrix();                                  //деструктор
matrix operator + (matrix &B);
matrix operator - (matrix &B);
matrix operator * (matrix &B);
matrix operator ! ();                      //транспонирование
matrix operator =(matrix &B);              //присваивание
inline matrix operator * (float a); //Умножение на скаляр
inline matrix operator * (int a);
inline matrix operator /(float a); //Деление на скаляр
inline matrix operator /(int a) ;
inline matrix set_ZERO();                 //обнуление
inline matrix swap_rows(unsigned int row1,unsigned int row2);
inline matrix swap_cols(unsigned int coll,unsigned int col2);
//вычисление минора
matrix Menor (unsigned int row,unsigned int col);
void Out();                               //Вывод содержимого
matrix input();                           //ввод значений
public:
friend float Det(matrix &B);              //определитель
friend matrix Adj(matrix &B);             //приведение
friend matrix Inv(matrix SB);             //инвертирование
};

//конструктор
matrix::matrix(unsigned int rows,unsigned int cols)
{
unsigned int i;
ROWS=rows;
COLS=cols;
if (!(point = new float*[ROWS] ) )
{
cerr << "Невозможно разместить матрицу в памяти";
exit(ERR_EXIT);
}
for (i=0;i<ROWS;i++)
if ( !(point [i]=new float[COLS] ) )
{
cerr << "Невозможно разместить матрицу в памяти";
exit(ERR_EXIT);
}
}

//Конструктор копирования
matrix::matrix(const matrix &REF)
{
unsigned int i, j; //Размещаем новую матрицу
ROWS=REF.ROWS;COLS=REF.COLS;
if ( !( point = new float*[ROWS] ) )
{
cerr << "Невозможно разместить матрицу в памяти";
exit(ERR_EXIT);
}
}

```



```

for (i=0;i<ROWS;i++)
if ( !( point[i]=new float[COLS] ) )
{
cerr << "Невозможно разместить матрицу в памяти";
exit(ERR_EXIT);
}
//Копируем матрицу в новое расположение
for (j=0;j<COLS;j++)
for (i=0;i<ROWS;i++)
point[i][j]=REF.point[i][j];
}

//Присваивание (=)
matrix matrix::operator=(matrix &B)
{
unsigned int i,j;
for (i=0;i<COLS;i++)
delete(point[i]);
delete (point);
ROWS=B.ROWS; COLS=B.COLS;
if ( !( point = new float*[ROWS] ) )
{
cerr << "Невозможно разместить матрицу в памяти";
exit(ERR_EXIT);
}
for (i=0;i<ROWS;i++)
if ( !( point [i]=new float[COLS] ) )
{
cerr << "Невозможно разместить матрицу в памяти";
exit(ERR_EXIT);
}
for (j=0;j<B.ROWS;j++)
for (i=0;i<B.COLS;i++)
point[i][j]=B.point[i][j];
return *this;
}

//Деструктор
matrix::~~matrix()
{
unsigned int i;
for (i=0;i<ROWS;i++)
delete(point[i]);
delete (point);
}

//Транспонирование (!)
matrix matrix::operator !()
{
unsigned int i,j;
matrix TEMP(ROWS,COLS);
for (j=0;j<COLS;j++)
for (i=0;i<ROWS;i++)
TEMP.point[j][i]=point[i][j];
return TEMP;
}

//Обнуление
inline matrix matrix::set_ZERO ()
{
unsigned int i, j;

```

```

for (j=0;j<COLS;j++)
for (i=0;i<ROWS;i++)
point[i][j]=0;
return *this;
}

//Операция вычитания (-)
matrix matrix::operator -(matrix &B)
{
unsigned int i,j;
if ((COLS!=B.COLS) || (ROWS!=B.ROWS))
{
cerr << "Матрица несовместима с операцией -.";
exit (ERR_EXIT);
}
matrix C(ROWS,COLS);
for (j=0;j<COLS;j++)
for (i=0;i<ROWS;i++)
{
C.point[i][j]=point[i][j]-B.point[i][j];
}
return C;
}

//Операция сложения (+)
matrix matrix::operator +(matrix &B)
{
unsigned int i,j;
if ((COLS!=B.COLS) || (ROWS!=B.ROWS))
{
cerr << "Матрица несовместима с операцией +.";
exit (ERR_EXIT);
}
matrix C(ROWS,COLS);
for (j=0;j<COLS;j++)
for (i=0;i<ROWS;i++)
{
C.point[i][j]=B.point[i][j]+point[i][j];
}
return C;
}

//Операция умножения (*)
matrix matrix::operator * (matrix &B)
{
unsigned int i,j,k;
if ((COLS!=B.ROWS)
{
cerr << "Матрица несовместима с операцией *.";
exit (ERR_EXIT);
}
matrix M(ROWS,COLS);
M.set_ZERO(); //обнуляем M
//умножение:
for (j=0;j<B.COLS;j++)
for (i=0;i<ROWS;i++)
for (k=0;k<B.ROWS;k++)
M.point[i][j]=M.point[i][j]+point[i][k]*B.point[k][j];
return M;
}

```

```

//Инвертирование Inv(M)
matrix Inv(matrix &B)
{
    unsigned int i,j;
    float det;
    matrix InvM(B.ROWS,B.COLS);
    det=Det(B);
    if (det==0)
    {
        cerr<<"Матрица не имеет обратной";
        exit(ERR_EXIT);
    }
    //инвертируем
    InvM=(Adj(!B))/det;
    return InvM;
}

//Детерминант матрицы 2x2 (Det2x2)
float matrix::Det2x2()
{ //особый случай
    float det;
    det=point[0][0]*point[1][1]-point[0][1]*point[1][0];
    return det;
}

//Детерминант матрицы Det(M)
float Det(matrix &B)
{
    unsigned int n;
    int signo;
    float det=0;
    if (B.ROWS!=B.COLS)
    {
        cerr<<"Матрица должна быть квадратной!";
        exit(ERR_EXIT);
    }
    else
    if (B.ROWS==1)
        return B.point[0][0];
    else
    if (B.ROWS==2)
        return B.Det2x2();
    else
        for (n=0;n<B.COLS;n++)
        {
            //проверка на четность, для четных столбцов
            //знак +, нечетных знак -
            (n&1)==0 ? (signo=1) : (signo=-1);
            det=det+signo*B.point[0][n]*Det(B.Menor(0,n));
        }
        return det;
    }

//Умножение матрицы на скаляр
matrix matrix::operator * (float a)
{
    unsigned int i,j;
    for (j=0;j<COLS;j++)
    for (i=0;i<ROWS;i++)
        point[i][j]=a*point[i][j];
    return *this;
}

```

```

}

//Умножение матрицы на целочисленный скаляр
matrix matrix::operator *(int a)
{
    unsigned int i,j;
    for (j=0;j<COLS;j++)
    for (i=0;i<ROWS;i++)
    point[i][j]=a*point[i][j];
    return *this;
}

//Деление матрицы на скаляр
matrix matrix::operator /(float a)
{
    unsigned int i,j;
    for (j=0;j<COLS;j++)
    for (i=0;i<ROWS;i++)
    point[i][j]=point[i][j]/a;
    return *this;
}

//Деление матрицы на целочисленный скаляр
matrix matrix::operator /(int a)
{
    unsigned int i,j;
    for (j=0;j<COLS;j++)
    for (i=0;i<ROWS;i++)
    point[i][j]=point[i][j]/a;
    return *this;
}

//Вычисление минора матрицы
matrix matrix::Menor(unsigned int a,unsigned int b)
{
    unsigned int i,j,p,q;
    matrix MEN (ROWS-1, COLS-1);
    for (j=0,g=0;q<MEN.COLS;j++,q++)
    for (i=0,p=0;p<MEN.ROWS;i++,p++)
    {
        if (i==a) i++;
        if (j==b) j++;
        MEN.point[p][q]=point[i][j];
    }
    return MEN;
}

//Приведение матрицы
matrix Adj (matrix &B)
{
    unsigned int i,j;
    float signo;
    matrix ADJ(B.ROWS,B.COLS);
    for (j=0;j<B.COLS;j++)
    for (i=0;i<B.ROWS;i++)
    {
        //проверка на четность, для четных знак +, нечетных знак-
        ((i+j)&1)==0 ? (signo=1) : (signo=-1);
        ADJ.point[i][j]=signo*Det(B.Menor(i,j));
    }
    return ADJ;
}

```

```

}

//Ввод данных
matrix matrix::input()
{
    unsigned int i,j;
    cout << "Введите матрицу";
    cout << ROWS << 'X'<< COLS << ":"<<'\n';
    flush(cout);
    for (i=0;i<ROWS;i++)
    for (j=0;j<COLS;j++)
    {
        cin>>point[i][j];
    }
    return *this;
}

//Вывод
void matrix::Out()
{
    unsigned int i,j;
    cout << endl;
    for (i=0;i<ROWS;i++)
    {
        for (j=0;j<COLS;j++)
        cout<< point[i][j] << " ";
        cout<< endl;
    }
    flush(cout);
}

//Перестановка строк и столбцов
matrix matrix::swap_rows(unsigned int row1,unsigned int row2)
{
    unsigned int j;
    float *FILA=new (float);
    if ( (row1>ROWS)|| (row2>ROWS) )
    {
        cerr<<"Невозможно переставить несуществующие строки!";
        exit(ERR_EXIT);
    }
    FILA=point[row1];
    point[row1]=point[row2];
    point[row2]=FILA;
    return *this;
}

matrix matrix::swap_cols(unsigned int col1,unsigned int col2)
{
    unsigned int i;
    float COLUMN;
    if ( (col1>COLS)|| (col2>COLS) )
    {
        cerr<<"Невозможно переставить несуществующие столбцы!";
        exit(ERR_EXIT) ;
    }
    for (i=0;i<ROWS;i++)
    {
        COLUMN=point[i][col1];
        point[i][col1]=point[i][col2];

```

```

point[i][col2]=COLUMN;
}
return *this;
}

//Пример работы программы
# include "matrix.cpp"
#include <stdlib.h>
void main ()
{
unsigned int i, j;
matrix A(4,4) ,B(4,4) ,C(4,4);
A.input ();
cout<<"A = \n";
A.Out ();
B=Inv(A);
cout<<"B = \n";
B.Out ();
C=A*B;
cout<<"Результат умножения A*B:\n";
C.Out ();
cout<<"Введите множитель i = \n";
cin >> i;
C = C * i ;
cout<<"Результат умножения: \n";
C.Out ();
C = !C ;
cout<<"Результат транспонирования: \n";
C.Out ();
cout<<"Детерминант A: "<<Det (A) <<"\n";
}

```

Следующая программа наглядно иллюстрирует концепцию перегрузки операторов, осуществляя суммирование углов, заданных с помощью строк формата градусыd минутm секундыs

Выделение из строки значащих частей осуществляет функция strtok(). Признаками конца лексем служат буквы d, m и s.

```

//Пример 3
//angles.cpp
//Эта программа на языке C++ содержит пример перегрузки оператора +.
//
#include <iostream.h>
#include <string.h>
#include <stdlib.h> //функция atoi()
class angle_value
{
int degrees, minutes, seconds;
public:
angle_value () //стандартный конструктор
{
degrees = 0,
minutes = 0,
seconds =0;
}
angle_value(char *); //конструктор
int get_degrees();
int get_minutes();
int get_seconds();
angle_value operator +(angle_value); //перегруженный оператор
};

```

//выделение значащих частей строки и преобразование их в целые числа

```

angle_value::angle_value(char *angle_sum)
{
degrees=atoi(strtok(angle_sum, "d"))
minutes=atoi (strtok(0, "m"));
seconds=atoi (strtok(0, "s"));
}

int angle_value::get_degrees()
{
return degrees;
}

int angle_value::get_minutes()
{
return minutes;
}

int angle_value::get_seconds()
{
return seconds;
}

angle_value angle_value::operator+(angle_value angle_sum)
{
//Раздельное суммирование градусов, минут и секунд.
//Здесь применяется еще не перегруженный оператор +
angle_value ang;
ang.seconds=(seconds + angle_sum.seconds) % 60;
ang.minutes=((seconds+angle_sum.seconds)/60+
minutes+angle_sum.minutes)%60;
ang.degrees=((seconds+angle_sum.seconds)/60+
minutes+angle_sum.minutes)/60;
ang.degrees+=degrees+angle_sum.degrees;
return ang;
//когда сумма секунд или минут превышает 60, должен осуществляться
//перенос соответствующего числа разрядов. Поэтому каждая предыдущая
//операция повторяется в следующей, только оператор деления по модулю
//(%) заменяется оператором деления без остатка (/).
}

int main ()
{
char str1[] = "37d 15m 56s'
angle_value angle1(str1);
char str2[] = "10d 44m 44s";
angle_value angle2(str2);
char str3[] = "75d 17m 59s";
angle_value angle3(str3);
char str4[] = "130d 32m 54s";
angle_value angle4(str4);
angle_value sum_of_angles;
sum_of_angles = angle1 + angle2 + angle3 + angle4;
cout<<"Сумма углов равна"<<sum_of_angles.get_degrees()<<"d";
cout<<sum_of_angles.get_minutes()<<"m"<<sum_of_angles.get_seconds();
cout<<"s"<<endl;
return (0);
}

```

Программа выдаст на экран такую строку:  
Сумма углов равна 253d 51m 33s

### Виртуальные функции

#### Пример 1. Разница между виртуальными и обычными функциями.

```

#include <iostream.h>
class parent                                //объявление базового класса
{
public: int k;
parent(int a)                              //конструктор класса
{
k = a;
}
virtual void fv()                          //виртуальная функция
{
cout << "call fv() of base class:";
cout << k << endl;
}
void f()                                    //обычная функция
{
cout << "call f() of base class: ";
cout << k*k << endl;
}
};
class child1:public parent
{
public:
child1(int x):parent(x) {}
void fv()
{
cout<<"call fv() of derived class child1:";
cout<<(k + 1)<<endl;
}
void f ()                                  //обычная функция
{
cout<<"call f () of derived class child1:";
cout<<(k-1)<<endl;
}
};
int main ()
{
parent *p;                                //указатель на базовый класс
parent ez(5);                             //объявление объекта базового класса child1 chez1(15);
                                           //объявление объекта производного класса
p=&ez;                                     // указатель ссылается на объект базового класса
p->fv();                                   //вызов функции fv() базового класса
p->f();                                    //вызов функции f() базового класса
p=Schez1;                                 //указатель ссылается на объект производного класса
p->fv();                                   //вызов виртуальной функции fv() производного класса
p->f();                                    // вызов обычной функции f() производного класса
return (0);
}

```

При выполнении приведенной программы на экран будут выданы следующие результаты:

```

call    fv() of base class: 5
call    f() of base class: 25
call    fv() of derived class child1: 16
call    f() of base class: 225

```

Полученные результаты показывают, что при обращении к виртуальной функции *fv()* производного класса происходит вызов функции производного класса. Обращение к обычной функции *f()* производного класса приводит к вызову одноименной функции базового класса, а не производного.

Пример 2. Реализация механизма множественного наследования классов с использованием виртуальных функций на примере графических классов Point (базовый), Circle, Rect (производные от Point, одиночное наследование), RecCircle (производный от Circle, Rect, множественное наследование)

```

//
#include<conio.h>
#include<graphics.h>

```



```

#include<stdlib.h>
//определение класса Point
class Point
{
void hide(); //собственный метод "спрятать точку"
protected: //защищенные данные (но будут //доступны при
наследовании)
    int x; int y;
public: //общедоступные методы
    Point(int,int); //конструктор
    int putx(); //для доступа к x
    int puty(); //для доступа к y
virtual void show(); //виртуальный метод "показать точку"
    void move(int,int); //метод "переместить точку"
//деструктор для класса Point описывать не будем
}; //end of class Point

//реализация методов класса Point
Point::Point (int x1=0,int y1=0) {x=x1;y=y1;}
int Point::putx() {return x;}
int Point::puty() {return y;}
void Point::show() {putpixel(x,y,14);}
void Point::hide() {putpixel(x,y,0);}
void Point::move(int x1=0,int y1=0) {hide();x=x1;y=y1; show();}

//определение нового класса Circle на основе класса Point
//применяется механизм одиночного наследования
class Circle:public Point
{ //описание компонентов класса
protected: //защищенные данные (но будут доступны
//при наследовании)
    int radius;
    int vis;
public:
    Circle(int,int,int); //constructor
    ~Circle(); //destructor
//методы putx(),puty() наследуются из Point
    void show(); //это виртуальный метод
    void hide(); //не наследуется из Point, т.к. метод //Point::hide() является
private
    void move(int,int);
    void vary(int); //change radius
    int putradius();
}; //end of class Circle

//реализация методов класса Circle
//constructor
Circle::Circle(int xc,int yc,int radc):Point(xc,yc) //выполняем вызов
//конструктора базового класса
{radius=radc; vis=1;} //дополнительные инициализации в конструкторе

//переопределение виртуальной функции
void Circle::show() {circle(x,y,radius);}

void Circle::hide()
{
    unsigned int col;
    if (vis==0) return;
    col=getcolor();
    setcolor(getbkcolor());
    circle(x,y,radius);
}

```

```

    vis=0;
    setcolor(col);
}

//другие методы класса Circle
void Circle::move(int x1=0,int y1=0)
{
hide();x=x1;y=y1;show();
}
void Circle::vary(int d)
{
hide();
radius+=d;
if (radius<0) radius=0;
show();
}

int Circle::putradius() {return radius;}

Circle::~Circle() {hide();} //деструктор

//определение нового класса Rect на основе класса Point
//применяется механизм одиночного наследования
class Rect:public Point
{
//описание компонентов класса
protected:
int dx,dy; //защищенные данные (будут //доступны при наследовании)
public:
Rect(int,int,int,int); //constructor
~Rect(); //destructor
//методы putx(),puty() наследуются из Point
void show(); //виртуальный метод
void hide(); //не наследуется из Point
}; //end of class Rect

//реализация методов класса Rect
//constructor
Rect::Rect(int x2,int y2,int dx2,int dy2):Point(x2,y2) //вызов
//конструктора //базового класса
{dx=dx2; dy=dy2;} //дополнительные инициализации в конструкторе

//переопределение виртуальной функции
void Rect::show()
{
int a=dx/2,b=dy/2;
rectangle(x-a,y-b,x+a,y+b);
}

void Rect::hide()
{
int a,b;
a=getcolor();
b=getbkcolor();
setcolor(b);
rectangle(x-a,y-b,x+a,y+b);
setcolor(a);
}

Rect::~Rect() {hide();} //деструктор

//определение нового класса RecCircle на основе классов Rect и Circle

```

```

//применяется механизм множественного наследования

class RecCircle:virtual public Rect,virtual public Circle
//virtual чтобы работать с указателем z на базовый тип Point
{
//описание компонентов класса
public:
RecCircle(int,int,int,int,int);          //constructor
~RecCircle();                            //destructor
//методы putx(),puty() наследуются из Point
void show();                             //виртуальная функция
void hide();
};

//end of class RectCircle

//реализация методов класса RecCircle
//constructor
RecCircle::RecCircle(int x2,int y2,int dx2,int dy2,int r2):
Rect(x2,y2,dx2,dy2), Circle(x2,y2,r2){}
//destructor
RecCircle::~~RecCircle(){};
//переопределение виртуальной функции
void RecCircle::show()
{Rect::show();Circle::show();}
void RecCircle::hide()
{Rect::hide();Circle::hide();}

//main program
void main()
{
char buf[30];
int a=DETECT,b;
initgraph(&a,&b,"");
RecCircle A(100,100,50,50,30),*zz;
RecCircle B(250,250,30,10,50);
Point C(200,200);          //C.show();
Circle D(200,200,50);      //D.show();
Rect E(200,200,50,50);     //E.show();
Point *z;
z=&C;z->show();
getch();
z=&D;
z->show();
getch();
z=&E;
z->show();
getch();
z=&A;
z->show();          //A.show();
getch();
z=&B;
z->show();          //B.show();
getch();
itoa(z->putx(),buf,10);    //нельзя z->Circle::putx()
outtextxy(10,10,buf);
zz=&B;
setcolor(10);
zz->Circle::show();
getch();
itoa(zz->Circle::putx(),buf,10); //нельзя zz->putx()
outtextxy(10,20,buf);
setcolor(11);
B.Circle::show();
}

```

```

getch();
setcolor(12);
z->show(); //нельзя z->Rect::show();
getch();
setcolor(14);
B.Rect::show();
getch();
void* zzz=&A; //zz=(RecCircle*) zzz;
setcolor(1);
((RecCircle*) zzz)->Circle::show();
getch();
setcolor(2);
((RecCircle*) zzz)->show();
getch();
closegraph();
}

```

### ПАРАМЕТРИЗОВАННЫЕ ФУНКЦИИ

Рассмотрим функцию  $\max(x, y)$ , которая возвращает больший из двух аргументов;  $x$  и  $y$  могут принадлежать к любому типу.

Первый способ решения этой проблемы — использовать макрос:

```
#define max(x,y) ((x) > (y)) ? (x) : (y)
```

Однако использование *#define* обходит механизм контроля типов. Фактически такое использование макроопределений почти вышло из употребления в C++. Естественно требовать, чтобы функция  $\max(x, y)$  сравнивала только совместимые типы. К сожалению, использование макроса допускает любое сравнение, например, между целочисленным значением и структурой, которые несовместимы.

Второй недостаток использования макроподстановки заключается в том, что подстановка будет выполнена не там, где необходимо. Используя взамен макроса шаблон, можно определить заготовку для семейства связанных перегруженных функций или классов, при этом тип данных передавать как параметр:

```

template <class T> T max(T x, T y)
{
    return (x > y) ? x : y;
};

```

Здесь тип данных представлен аргументом шаблона *<class T>*, где  $T$  — фиктивное имя типа данных, которое программист выбирает по своему усмотрению. При использовании его в приложении компилятор сгенерирует соответствующий код для функции  $\max$  согласно действительному типу данных, использованному при вызове функции.

**Пример 1.** Параметризованная функция для произвольного типа данных.

```

#include<iostream.h>
class MyClass
{
public: float r;
    MyClass(float r1){r=r1;}
    //перегрузка оператора сравнения > чтобы иметь возможность //сравнивать
    //объекты типа MyClass
    int operator > (MyClass& l){return r>l.r;}
};
template <class T> T max(T x,T y)
{return (x>y)?x:y;};
int max(int,int); //для исключения ошибки
void f(char c,int i)
{
    max(i,i);
    max(c,c);
    max(i,c);
    max(c,i);
}
void main()
{
    float a=1.0,b=3.4,c;
    c=max(a,b); //вещественные аргументы
}

```

```

    cout<<"c="<<c<<endl;
    int i=10;
    int j=max(i,5);          //целые аргументы
    cout<<"j="<<j<<endl;
    MyClass x(1.5),y(2.1);
    MyClass z=max(x,y);      //аргументы типа MyClass
    cout<<"z="<<z.r<<endl;
}

```

В результате выполнения программа выдает следующее:

```

c = 3.4
j= 10
z=2.1

```

Любой тип данных (не только класс) может использоваться в подстановке `<class T>`. Компилятор сам позаботится о вызове соответствующего оператора сравнения `operator>()`, так что возможно использовать определенную таким образом функцию с аргументами любого типа, для которого определен оператор сравнения (в нашем примере это вызов функций с целыми и вещественными аргументами).

При использовании шаблонов следует проявлять осторожность. Дело в том, что логика работы созданной параметризованной функции для определенных типов данных может отличаться от предполагаемой. Приведем пример, иллюстрирующий вариант такого использования шаблона.

**Пример 2.** Некорректное использование шаблона.

```

#include <string.h>
#include <iostream.h>
template <class T> T max (T x, T y)
{return (x > y) ? x : y;};
int main () {
    int a = 5, b=30, c=15, d = 0;
    float pi = 3.141596, e = 2.172;
    d = max(a,max(b,c)) ;
    cout << " Наибольшее из трех целых:" << d << endl;
    cout << " Наибольшее из двух целых:" << max(d, a) << endl;
    cout << " max (pi, e) = " << max (pi, e) << endl;
    char m[] = "Мама", p[] = "Папа", *s;
    s= max(p,m);                //Внимание! Функция max сравнит два
                                //указателя, а не содержимое строк!
    cout << "Кто в доме хозяин? :" << s << endl;
}

```

В результате выполнения программа выдаст следующее:

```

Наибольшее из трех целых: 30
Наибольшее из двух целых: 30
Кто в доме хозяин? : Мама

```

Предполагалось, что в последнем случае программа будет посимвольно сравнивать две строки, а в кодировке ASCII, справедливо отношение "Папа" > "Мама". В действительности результаты такого сравнения непредсказуемы и зависят от того, в какой последовательности компилятор распределяет в памяти указанные строки символов. Для приведенной программы компилятор автоматически сгенерирует по шаблону следующие три перегруженные функции:

```

int max (int x, int y)
{return (x > y) ? x : y;}
float max (float x, float y)
{return (x > y) ? x : y;}
char* max (char * x, char * y)
{return (x > y) ? x : y;}

```

Экземпляр функции создается каждый раз при вызове функции с аргументами, для которых определения не найдено. Чтобы не проводилось бессмысленное сравнение двух указателей, для строк можно явно перегрузить функцию `max` следующим образом:

```

char *max(char *x, char *y)
{return (strcmp(x, y) > 0) ? x : y;
}

```

В этом случае компилятор не будет производить генерацию функции по шаблону для аргументов типа `char*`, а воспользуется уже готовой. Другое решение проблемы — использовать классы, для которых определена операция ">", например, `class String`.

Следует иметь в виду, что для параметризованных функций преобразования аргументов по умолчанию не выполняются. Кроме того, параметризованная функция должна использовать все типы формальных аргументов шаблона, в противном случае компилятор не сможет определить фактические типы для генерации тела.

Принимая решение о перегрузке операндов, компилятор игнорирует функции, которые сгенерированы компилятором неявно.

**Пример 3.** Преобразование типов аргументов при использовании шаблонов.

```
template <class T> T min(T a, T b)
{ return (a < b) ? a : b; };
void f(int i, char c)
{
    min(i, i);           //вызов min(int, int)
    min(c, c);           //вызов min(char, char)
    min(i, c);           //нет совпадений для min(int, char) !
    min(c, i);           //нет совпадений для min(char, int) !
}
```

Для того чтобы вызов функции *min()* не приводил к сообщениям об ошибках, в последних двух строках данного примера, необходимо явно объявить формат, позволяющий произвести преобразование типов. Так, если добавить в указанную программу только объявление (тело функции будет создано шаблоном):

```
int min(int, int);
```

то сообщения об ошибках выдаваться не будут.

### Параметризованные классы

Рассмотрим пример класса *Vector* (одномерный массив). Независимо от того это есть вектор целых, вещественных, комплексных чисел или любого другого типа, основные операции, которые он позволяет выполнять — те же (инициализация, получение элемента по индексу и т. п.). Естественное решение для определения такого класса — использовать шаблон. Аналогично тому, как компилятор производил генерацию функций, при подстановке фактического типа в шаблон класса *Vector*, система создает класс автоматически.

**Пример.** Использование шаблона для определения класса *Vector*.

```
#include<alloc.h>           //прототип coreleft()
#include<iostream.h>
template <class T> class Vector
{
    T* data;
    int size;
public:
    Vector(int);
    ~Vector(){delete [] data;}
    T& operator [] (int i) {return data[i];}
    void get_size();
};
template <class T> Vector<T>::Vector(int n)           //конструктор
{
    data=new T[n];
    size=n;
};
template <class T> void Vector<T>::get_size()
{ cout<<"size="<<size<<endl; }

void main()
{
    cout<<"start"<<endl;
    cout<<coreleft()<<endl;
    Vector<int> x(10); //создаем вектор из 10 элементов типа int
    x.get_size();
    for (int i=0;i<10;++i)
        x[i]=i;           //присваиваем значения
    cout<<coreleft()<<endl;
    Vector<char> c(5); //создаем вектор из 5 элементов типа char
    c.Vector<char>::get_size();
    for (char j=0;j<5;++j)
```

```

        c[j]=j+'a';
        cout<<coreleft()<<endl;
        //присваиваем значения
        //print
    for (i=0;i<10;i++)
        cout<<x[i]<<" ";
    cout<<endl;
    for (j=0;j<5;++j)
        cout<<c[j];
    cout<<endl;
    x.Vector<int>::~~Vector();
    cout<<coreleft()<<endl;
    cout<<"stop"<<endl;
    cout<<coreleft()<<endl;
}

```

В этом примере используется функция *coreleft()*, которая возвращает размер не используемой оперативной памяти. Выводятся различные значения в зависимости от модели памяти. В малых моделях памяти возвращает значение равное количеству неиспользуемой памяти между вершиной heap и стеком. В больших моделях памяти, она возвращает объем памяти между самым верхним выделенным блоком и концом оперативной памяти.

```

#include<iostream.h>
#include<alloc.h>

```

```

int main(void)
{
    cout<<"Разница между самым верхним выделенным блоком и \n";
    cout<<"вершиной heap равна <<(unsigned long)coreleft()<<байт";
    return 0;
}

```