

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Кафедра численных методов и программирования

ВВЕДЕНИЕ В C++

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО КУРСУ
“МЕТОДЫ ПРОГРАММИРОВАНИЯ”

Для студентов механико-математического факультета

МИНСК
2000

А в т о р ы – с о с т а в и т е л и
И. Н. Блинов, В. С. Романчик

Рекомендовано
Советом механико-математического факультета БГУ
“20” февраля 1999г., протокол № 5

PDF-версия:
Д. П. Глиндзич, А. В. Грызлов

©БГУ, 2000

СОДЕРЖАНИЕ

Введение.....	4
<i>Глава 1. Классы: первое знакомство.....</i>	<i>9</i>
<i>Глава 2. Классы и функции.....</i>	<i>17</i>
<i>Глава 3. Наследование, виртуальные функции и полиморфизм.....</i>	<i>34</i>
<i>Глава 4. Массивы объектов, указатели и ссылки.....</i>	<i>45</i>
<i>Глава 5. Шаблоны и обработка исключительных ситуаций.....</i>	<i>50</i>
<i>Глава 6. Потоки и классы ввода/вывода.....</i>	<i>54</i>
 <i>Приложение 1. Задания для выполнения.....</i>	 <i>58</i>
<i>Приложение 2. Примеры программ.....</i>	<i>59</i>

ВВЕДЕНИЕ

Язык C++ был разработан в Bell Laboratories(США) в 1983г. как расширение языка С. Символы “++” – представляют оператор инкрементации в языке С, что в действительности означает выполнимость С-программ в среде C++. Еще одним источником C++ разработчик языка Б. Страуструп называет язык Simula67, из которого позаимствованы понятия класса, виртуальных функций. Класс соответствует введенному пользователем типу, для которого обеспечивается набор данных, операции над данными, их сокрытие и др. В настоящее время классы рассматриваются как носители таких черт объектно-ориентированного программирования как инкапсуляция, наследование, полиморфизм.

Существует несколько реализаций системы, поддерживающих стандарт C++, из которых можно выделить реализации Visual C++ (Microsoft), Borland C++ (Inprise), а также реализацию в системе UNIX. Отличия относятся, в основном, к используемым библиотекам классов и интегрируемым средам.

В C++ можно использовать потоковый ввод/вывод: оператор << осуществляет вывод в поток *cout*, оператор >> осуществляет ввод из потока *cin*. Такой ввод/вывод поддерживается включением файла <iostream.h>. Например:

```
// вывод строки символов (p01)
#include <iostream.h>
void main()
{
    cout << “Hello, world \n”;
}
```

В программе *p01* использован однострочный комментарий C++, который начинается от символов // и продолжается до конца строки. Наряду с таким комментарием, может быть использован обычный комментарий языка С, ограниченный символами /* и */. Программа выводит строку “Hello,world” в выходной поток *cout*, ‘\n’ – символ перехода к следующей строке.

Отметим, что функция *main()*, как и другие функции имеет возвращаемое значение, если не указан тип *void*. По умолчанию это значение типа *int*.

Язык C++ позволяет объявлять локальные переменные в любом блоке кода, которые будут известны только в этом блоке ниже объявления.

//вычисление факториала (p02)

#include <iostream.h>

main() //по умолчанию возвращает значение типа int

{

int n;

cout<<"введите число:";

cin>>n; //ввод

int fact=1;

for(int j=n;j>=1;j--) fact=fact*j;

cout<<"факториал равен:"<<fact;

return 0; //возвращаемое значение

}

В C++ реализована перегрузка функций, что означает возможность использования функций с одним и тем же именем, но с различными типами либо количеством аргументов.

//перегрузка abs (p03)

#include <iostream.h>

int ABS(int n);

double ABS(double n);

main()

{

//вызов ABS (int)

cout<<"абсолютная величина-10:" << ABS(-10) << "\n";

//вызов ABS(double)

cout<<"абсолютная величина-10.01:"<< ABS(-10.01) << "\n";

return 0;

}

int ABS(int n)

{

```

cout << "В целом ABS()\n";
return n<0 ? -n : n;
}
double ABS(double n)
{
cout << "в double ABS() \n";
return n<0 ? -n : n;
}

```

В C++ используются аргументы функций по умолчанию. При этом один или несколько параметров при всех вызовах могут быть одинаковы, кроме особых случаев. Например:

```

void func(int x=0, int y=1);
//аргументы задаются только при первом объявлении
void main()
{
func(1,2); //значения по умолчанию не используются
func(1); //x=1,y=1
func(); //x=0, y=1.
}
void func(int x,int y)
{
int z=x+y;
}

```

Аргументы по умолчанию должны быть определены справа налево. Например, только левому аргументу присвоить значение нельзя.

В C++ реализован ссылочный тип, позволяющий передавать параметры функций не только по значению, но и по адресу. Например:

```

//функция меняет местами a и b (p04)
#include<iostream.h>
void swap(int &a, int &b)
{int c;
c=a;
a=b;
}

```

```

    b=c;
}
void main()
{int a=3,b=4;
 swap(a,b); // поменять местами
cout<<a<<b<<endl;
}

```

В C++ введены новые операции `::`, `new`, `delete`. Оператор расширения области видимости `::` используется для доступа к членам базового класса (оператор вида *basis::name*), для указания внешней или глобальной области видимости, скрытой локальным контекстом (оператор вида *::global_name*). Оператор *class_name::name* указывает доступ к элементам класса. Например, *basis::func()* означает принадлежность функции *func()* области видимости класса *basis*. Фактически этот оператор используется для указания маршрута к идентификатору. Рассмотрим пример:

```

#include <iostream.h> //p05
#include <conio.h>
int i=1;
class basis
{
public:
int i;
basis(){i=2;}
void func(){
cout<<::i<<" ";//доступ к глобальной компоненте i=1
};
class inner:public basis{
public:
int i;
inner(){i=3;}
void func()
{
cout<<basis::i<<" ";//i=2
basis::func();//i=1
}
}

```

```

};
void main (){
basis ob1;
inner ob2;
    ob1.func();
    ob2.func();
cout<<basis::i<<" ";//i=2
cout<<inner::i;//i=3
while (!kbhit());
}

```

Операторы *new* и *delete* используются для динамического выделения и освобождения памяти. Общая структура оператора *new*/:

new тип [размер] (инициализатор);

Оператор *delete* освобождает память. В следующем примере создается двумерный динамический массив и символьная строка.

```

#include <iostream.h> //p06
#include <conio.h>
void main()
{
int i,j,n,**a;
cin>>n;
cout<<oct<<n<<"\t"<<hex<<n<<"\t"<<dec<<n<<endl;
a=new int*[n]; //создание двумерного массива
for(i=0;i<n;i++){
a[i]=new int[n];
for(j=0;j<n;j++)
cin>>a[i][j];} //ввод значений
for(i=0;i<n;i++){ //вывод значений двумерного массива
for(j=0;j<n;j++)
cout<<a[i][j]; cout<<endl;}
long double *p;
p=new long double(3.1);//инициализация *p=3.1
char *c=new char[80];

```



```

c="String" ;
cout<<"long double *p:"<<*p<<endl;
//выведено : long double *p:3.1
cerr<<"\n stream for errors c:"<<c<<"\n";
//выведено: stream for errors c:String
for(i=0;i<n;i++)
delete [] a[i];
delete p;
while(!kbhit());
}

```

Из других новостей отмечаем два новых типа: *bool* со значениями *true*, *false* и *wchar_t* –шестнадцатибитовые символы в форме UNICODE.

В C++ реализована концепция *объектно-ориентированного программирования*.

Это методология построения программ в виде множества взаимодействующих объектов, структура и поведение которых описана иерархически построенными классами. Классы представляют абстрактные типы данных с открытым интерфейсом и скрытой внутренней реализацией. В классах реализованы такие базовые принципы ООП как:

1. Абстракция данных.
2. Наследование – в производных классах могут быть наследованы члены базового класса.
3. Инкапсуляция – в классах объединяются данные и методы (функции) для работы с этими данными и только через методы возможен доступ к сокрытым данным.
4. Полиморфизм – возможность использования одних и тех же методов для работы с различными объектами.

1. КЛАССЫ. ПЕРВОЕ ЗНАКОМСТВО

Основное отличие языка C++ от C состоит в использовании класса, простейшее определение которого имеет вид:

```

class имя_класса
{ // по умолчанию раздел private – частные члены класса
public:

```

```
//открытые функции и переменные класса
};
```

Имя класса здесь является новым типом данных. Понятие переменной данного типа соответствует понятию объекта класса. Список членов класса включает описание типов и имен данных и функций. Функции, описания которых находятся в описании класса, называются функциями-членами класса. Классы могут также содержать определения функций, тело которых помещается в определение класса (*inline-функции*). В классах используются специальные функции - конструкторы с тем же именем что и класс, и деструкторы, с именем класса, перед которым стоит “~”.

Переменные, объявленные в разделе класса по умолчанию как частные (*private*), имеют область видимости в пределах класса. Их можно сделать видимыми вне класса, если поставить перед началом объявления слово *public*:

Классами в C++ являются также структуры (*struct*) и объединения (*union*). Ключевые слова *struct*, *union* не обязательны при объявлении переменных, так как имя структуры или объединения является полноценным типом в C++. Единственным отличием структуры от класса является то, что члены структуры по умолчанию являются открытыми, а не закрытыми, как у класса. Кроме этого объединения не могут наследоваться и наследовать.

Обычно переменные в классе являются *private* – переменными, а видимыми (*public*) являются функции.

В следующей программе определяется простейший класс *Strtype*, членами которого являются массив *str* типа *char* и функции *set()*, *show()*, *get()*.

```
#include <iostream.h> //(p10)
#include <string.h>
#include <conio.h>
class Strtype{
    char str[80];//private
public:
    void set (char *);//задать str
    void show(); //вывести str
```

```

        char* get(); //вернуть str
    };
void Strtype::set(char *s)
    {strcpy(str,s);} //копирование s в str
void Strtype::show()
    {cout<<str<<endl;}
char * Strtype::get()
    {return str;}
void main()
    {Strtype obstr;
    obstr.set("bel.university ");
    obstr.show();
    cout<<obstr.get()<<endl;
    while(!kbhit());
    }
Вывод: bel.university
bel.university

```

Переменная *str* — является частной (*private*), доступ к ней возможен только через члены функции класса. Такое объединение в классе сокрытых данных и открытых функций и есть инкапсуляция. Здесь *obstr* является объектом данного класса (экземпляром класса). Вызов функции осуществляется добавлением к имени функции имени объекта, после имени ставится точка или \rightarrow в случае данного объекта.

Приведем примеры нескольких объявлений объектов.

```

Strtype a; //объект a
Strtype x[100]; //массив объектов
Strtype *p; //указатель на объект
p=new strtype;
Вызов функции:
a.set("строка");
x[i].set("строка");
p->set("строка");

```

Следующая программа демонстрирует создание класса *stack* на основе динамического массива.

Для инициализации объекта класса используется метод *stack()*, а не метода *init()* или *set()* и т.д. Метод, имя которого совпадает с именем класса, является конструктором и вызывается автоматически при объявлении объекта класса *stack*. Деструктор *~stack()* освобождает выделяемую конструктором динамическую память.

```
#include <iostream.h> //(p11)
#define SIZE 10
// объявление класса stack для символов
class stack {
    char *stck; // содержит стек
    int tos; // индекс вершины стека
public:
    stack(); //конструктор
    ~stack(){delete stck;} //деструктор
    void push(char ch); // помещает в стек символ
    char pop(); // выталкивает из стека символ
};
// инициализация стека
stack::stack()
{ stck=new char[SIZE]; //динамический массив
  tos=0;
  cout << "работа конструктора ... \n";
}
// помещение символа в стек
void stack::push(char ch)
{
    if (tos==SIZE) {cout << "стек полон";return; }
    stck[tos]=ch;
    tos++;
}
// выталкивание символа из стека
char stack::pop()
{
    if (tos==0) {cout << "стек пуст";return 0;}
    tos--;
    return stck[tos];
}
```

```

}
main()
{
//образование двух, автоматически инициализируемых стеков
    stack s1, s2; //вызов конструктора для s1 и s2
    int i;
    s1.push('a');s2.push('x');
    s1.push('b');s2.push('y');
    s1.push('c');s2.push('z');
    for(i=0;i<3;i++) cout<<"символ из s1:"<<s1.pop() << "\n";
    for(i=0;i<3;i++) cout<<"символ из s2:"<<s2.pop() << "\n";
    cin>>i;//задержка
    return 0;
}

```

Вывод:

Работа конструктора ...

Работа конструктора ...

Символ из s1:c

Символ из s1:b

Символ из s1:a

Символ из s2:z

Символ из s1:y

Символ из s1:x

Конструктор *stack* вызывается при объявлении объектов *stack s1,s2*; и выполняет инициализацию этих объектов, состоящую из выделения памяти для динамического массива и установки указателя на вершину стека в нуль.

Конструктор может иметь параметры, но не может иметь возвращаемого значения. Как и другие функции, конструктор может быть перегружаемым, то есть класс может иметь несколько конструкторов, отличающихся списком параметров.

Функцией, выполняющей действия, обратные выполняемым конструктором, является деструктор. Деструктор выполняет действия по деинициализации объекта, по освобождению динамической памяти. Деструктор не имеет аргументов, хотя может иметь возвращаемое

значение. Именем деструктора является имя класса, перед которым стоит знак “~” – тильда.

```
#include <iostream.h> //(p12)
#include <conio.h>
struct node{
    int info;
    node* next; };
class list {
    node* top;
public:
    list(){top=0; cout<<"\nkonstructor:\n";};
    ~list(){release(); cout<<"\ndestructor:\n";}
    void push(int);
    void del()
        { node* temp = top;
          top = top->next;
          delete temp; }
    void print();
    void release();
};
void list::push(int i)
{ node* temp = new node;
  temp->info = i;
  temp->next =top;
  top=temp; }
void list::print()
{node* temp=top;
 while (temp!=0)
  {cout<<temp->info<<"->";
   temp=temp->next;}
cout<<endl;}
void list::release()
{while (top!=0)del();}
void main()
{    list *p;
   { list st;
```

```

    int n = 0;
    cout << "Введите целое кроме 999: ";
    do {
        cin >> n; st.push(n);
    }while(n != 999);
    st.print();
    st.del();
    p=&st;
    p->print(); }
    while(!kbhit());
}

```

Важнейшим принципом ООП, реализованным в C++ является *наследование*.

Класс, который наследуется, называется базовым классом, наследуемый класс называется производным классом. В следующем простом примере класс *derived* наследует компоненты класса *base*, точнее компоненты раздела *public* остаются открытыми, компоненты раздела *private* остаются закрытыми и не доступны для порожденного класса. Доступными для порожденного класса являются и компоненты раздела *protected* (защищенный). Объекту базового класса можно присвоить объект производного, также как и соответствующим указателям.

```

#include <iostream.h> //(p12)
#include <conio.h>
// задание базового класса
class base {
    int i;//private
protected:
    static int k;

public:
    void set_i(int n);//установка i
    int get_i();//возврат i
};
int base::k=5;

```

```

//задание производного класса
class derived : public base {
    int j;
public:
    void set_j(int n);
    int mul();//умножение i базового класса на j порожденного
};
//установка значения i в базовом классе
void base::set_i(int n)
{
    i = n;
}
//возврат значения i в базовом классе
int base::get_i()
{
    return i;
}
//установка значения j в производном классе
void derived::set_j(int n)
{
    j = n;
}
//возврат знач. i*k из base умноженного на j из derived
int derived::mul()
{
    /*производный класс может вызывать функции-члены базового
    класса*/
    return j * get_i()*k;//вызов get_i() базового класса
}
main()
{
    derived ob;
    ob.set_i(10); //загрузка i в base
    ob.set_j(4); // загрузка j в derived
    cout << ob.mul(); //вывод числа 40
    base bob=ob;//присваивание объекта базовому
    cout<<bob.get_i();
    while (!kbhit());
}

```



```

    return 0;
}

```

Переменная *i* недоступна в порожденном классе, переменная *k* доступна, поскольку находится в разделе *protected*. Доступна также функция *get_i()* класса *base* из раздела *public*. В результате выводится *200 10*.

2. КЛАССЫ И ФУНКЦИИ

При реализации класса используются функции-члены класса, друзья класса, конструкторы, деструкторы, функции-операторы.

Функции-члены класса объявляются внутри класса, т.е. в описании класса находится прототип функции. Определение функции обычно помещается вне класса. При этом перед именем функции помещается *имя_класса::*.

Таким образом определение функции-члена класса имеет вид:

тип имя_класса:: имя_функции (описание аргументов) { }

Определение функции может находиться и внутри класса вслед за его объявлением. Такие функции называются *inline*-функциями.

```

#include <iostream.h> //(p20)
#include <malloc.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#define SIZE 255
class strtype {
    char *p;
    int len;
public:
    strtype()// инициализация объекта строка, inline конструктор
    {
        p=(char *) malloc(SIZE);
        if(!p) {

```

```

        cout << "ошибка выделения памяти\n";
        exit(1);
    }
    *p='\0';
    len=0;
}

~strtype(); //деструктор
void set(char *ptr);
char *get(){return p;}//inline деструктор
void show();
};
// освобождение памяти при удалении объекта строка
inline strtype::~strtype()//inline деструктор
{
    cout << "освобождение p\n";
    free(p);
}
void strtype::set(char *ptr)
{
    if(strlen(ptr) > SIZE) {
        cout << "строка слишком велика \n";
        return;
    }
    strcpy(p, ptr);
    len = strlen(p);
}
void strtype::show(){
    cout << p << " длина : " << len<< "\n";
}
main(){
    strtype s1,s2;
    s1.set("Это проверка");
    s2.set("I love C++");
    s1.show();
    s2.show();
    cout<<s2.get();
    while (!kbhit());
}

```

```

        return 0;
    }
В результате будет выведено:
Это проверка длина 12
I love C++ длина 10
I love C++
Освобождение p
Освобождение p

```

Деструктор `~strtype()` является *inline* - функцией, хотя и объявляется вне класса. Ключевое слово *inline* содержит указание компилятору создать код функции, подставляемый в точку объявления. При этом время вызова сокращается, хотя код может увеличиться. Отметим, что первый способ организации *inline* –функции является более распространенным. Вызов функций осуществляется одним из двух способов:

```

имя_объекта.имя_функции(аргументы);
указатель_на_объект -> имя_функции(аргументы);

```

Еще раз обращаем внимание на то, что при определении функции члена-класса перед именем функции стоит имя класса, а при вызове - имя объекта класса.

Указатель this

Обращение к данным объекта осуществляется с помощью функций, которые в свою очередь могут вызываться из различных объектов. Возникает вопрос: откуда функция “знает”, какой объект его вызвал, и какие данные при этом должны быть использованы? Например, в программе *p20* вызовы функции *s1.show()*; и *s2.show()*; приводят к выводу различных для *s1* и *s2* значений строк *p*.

Ответ: каждый объект, например *s1* и *s2*, имеет скрытый указатель *this* на этот объект, объявляемый неявно. Значением этого указателя является адрес начала объекта, который автоматически передается функциям-членам класса. В результате функция *show()* в реальности содержит инструкцию:

```

cout << this->p << " длина : " << this->len<< "\n";

```

При этом **this* представляет сам объект, *this->имя_члена* -ссылка на данные объекта класса. Рассмотрим пример:

```
#include<iostream.h> //p21
#include<conio.h>
class my_class {
    int x,y;
    public:
    void set(int xx, int yy)//устанавливает данные
    {x=xx;
    y=yy;} //равносильно this->x=xx;this->y=yy;
    my_class f(my_class &);//возвращает объект
    my_class *ff()//возвращает указатель на вызвавший объект
    {x=y=100;
    return this;}
    void display(){
    cout<<x<<'\\t'<<y<<endl;}
    };
    my_class my_class::f(my_class &M)
    {x+=M.x; y+=M.y;
    return *this;
    }
    void main(){
    my_class A,B;
    A.set(10,20);
    B.ff()->display();//результат 100,100
    B.display(); //результат 100,100
    A.display(); //результат 10,20
    A.f(B).display(); //результат 110,120
    A.display(); //результат 110,120
    while (!kbhit()); }
```

Так как функция *B.ff()* возвращает указатель *this* на объект *B*, то затем можно вызвать функцию *display()* как *this->display()*. При этом выводятся поля данных объекта *B(100,100)*. При вызове: *A.f(B).display()* функция *f(B)* возвращает значение вызвавшего ее объекта *A* как **this*.

Затем вызывается функция *display()* как *A.display()* и выводятся поля объекта *A*.

Отметим, что в функцию лучше передавать не значение объекта, а ссылку на него. Это дает экономию стековой памяти, так как объект не копируется и более безопасно, чем передача указателей на динамические объекты в качестве аргументов. Уничтожение копий аргументов при выходе из функции может разрушить динамические объекты.

Функции-“друзья” класса, объявляемые со спецификатором *friend*, указатель *this* не содержат. Объекты должны передаваться в качестве их аргументов.

Конструкторы и деструкторы

Конструктор – это функция-член класса, которая вызывается автоматически для создания и инициализации экземпляра класса. Известно, что экземпляры структур можно инициализировать при объявлении:

```
struct student  
{ int semesterhours; //public по умолчанию  
char subj;  
student s={0}; //объявление и инициализация
```

При использовании класса приложение не имеет доступа к защищенным элементам класса, поэтому подобную инициализацию выполнить нельзя.

Можно написать специальную функцию-член класса, инициализирующую экземпляр класса и указанные переменные при создании объекта. Подобная функция вызывается всегда при создании объекта класса. Это и есть конструктор – функция-член класса, которая имеет то же имя, что и класс. Конструктор может быть подставляемой (*inline*) и неподставляемой функцией. Рассмотрим пример:

```
#include<iostream.h> //p22  
class student{  
int semhours;  
char subj;
```

```

public:
student() //inline конструктор1
{ semhours=0; subj='A';}
  student(int ,char); //объявление конструктора2
};
student::student(int hours,char g)//коструктор2
{semhours=hours;
 subj=g;
}
void main(){
  student s(100,'A'); //конструктор2
  student s1[5]; //конструктор1 вызовется 5 раз
}

```

Там, где находится объявление *s* и *s1*, компилятор помещает вызов конструктора *s.student()*.

Конструктор не имеет типа возвращаемого значения, хотя может иметь аргументы и быть перегружаемым. Конструктор вызывается автоматически при создании объекта, при выполнении оператора *new* или при копировании объекта. Если конструктор отсутствует в классе, компилятор C++ генерирует конструктор по умолчанию.

Деструктор вызывается автоматически при уничтожении объекта. Деструктор имеет то же имя, что и класс, но перед ним стоит “~”. Деструктор можно вызывать явно, в отличие от конструктора. Конструкторы и деструкторы не наследуются, хотя производный класс может вызывать конструктор базового класса.

В следующем примере используется конструктор копирования, который необходимо вводить из-за проблемы, связанной с динамической памятью. Когда объект передается в функцию по значению, создается его копия. При этом конструктор не вызывается, т.к. это приводит к инициализации. При выходе из функции объект уничтожается, и вызывается его деструктор, освобождающий динамическую память. При этом исходный объект, использующий ту же динамическую память, что и копия, может быть поврежден. Аналогично, если функция возвращает объект, содержащий динамические переменные, после выхода из функции копия этого объекта разрушается вызовом деструктора. При этом возвращаемый объект тоже может быть разрушен, при

освобождении деструктором динамической памяти. То же происходит и при инициализации объявляемого объекта вида:

Class_type B=A;

При этом происходит копирование объекта *A* в объект *B*, содержащий динамический массив. Если затем объект *A* разрушается вызовом деструктора, освобождающего динамическую память, то объект *B* тоже разрушается. Поэтому при таком объявлении должен вызваться конструктор копирования.

Общая форма конструктора копирования имеет вид:

```
имя_класса (const имя_класса &ob,...) {  
//тело конструктора - выделение памяти и копирование в нее ob  
}
```

Здесь *ob* является ссылкой на объект в правой части инициализации.

Рассмотрим пример использования конструктора копирования при создании безопасного динамического массива.

/*создается класс "безопасный" массив. Когда один объект-массив используется для инициализации другого, для выделения памяти вызывается конструктор копирования*/

```
#include <iostream.h> //p23  
#include <stdlib.h>  
#include <conio.h>  
class array {  
    int *p;  
    int size;  
public:  
    array (int sz) { cout << "constructor1 \n";  
        p=new int[sz];  
        if(!p) exit(1);  
        size=sz;  
    }  
    ~ array() {  
        delete [] p;
```

```

        cout<<"destructor\n";
    }
    array(const array &a); //конструктор копирования
    void put(int i, int j) {
        if(i>=0 && i<size) p[i]=j;
    }
    int get(int i) {
        return p[i]; }
};

/* конструктор копирования. Память выделяется специально для
копии, и адрес этой памяти передается p.*/
array::array(const array &a) {
    p=new int[a.size]; //выделение памяти для копии
    if(!p) exit(1);
    for(int i=0;i<a.size;i++)
        p[i]=a.p[i]; //копирование содержимого в память для копии
    cout << "constructorcopy2\n";}

void main()
{
    {   array num(10);           //вызов обычного конструктора
        int i;
        for(i=0;i<10;i++)
        {
            num.put(i, i); //помещение в массив нескольких значений
            cout << num.get(i); //вывод на экран num
        }
        cout << "\n";
        array x=num; //создание массива x и инициализация, вызов
//конструктора копирования.
// Другой способ вызова - объявление: array x(num);
        num=x; //конструктор не вызывается !!!
    }
    while(!kbhit());
}

```

Результат:

constructor1
0123456789
constructorcpy2
destructor
destructor

Объект *num* используется при инициализации объекта *x* с помощью конструктора копирования. При этом выделяется динамическая память, адрес которой помещается в *x.p*, и производится копирование *num* в *x*. После этого *num* и *x* не разделяют одну и ту же динамическую память. Если присваивание происходит не при инициализации, конструктор копирования не вызывается, а происходит побитовое копирование объекта. В результате используется та же динамическая память. Так происходит при присваивании *num=x* в примере.

В следующем примере конструктор копирования вызывается при передаче объекта – строки в качестве аргумента функции *show()*.

```
#include <iostream.h> //p24
#include <stdlib.h>
#include <conio.h>
class Strtype {
    char *p;
    int l;
public:
    Strtype(char *s='\0')           //конструктор
    {   l=strlen(s);
        p=new char[l+1];
        if(!p) {   cout << "ошибка памяти\n";
                    exit(1); }
        strcpy(p, s);
        cout<<"constr1\n";
    }
    Strtype(const Strtype &o)//конструктор копирования
    {   l=strlen(o.p);
        p=new char[l+1];//выделение памяти для новой копии
        if(!p) {cout << "ошибка памяти\n";
                    exit(1);}
    }
```

```

        strcpy(p, o.p);          //передача строки в копию
    cout<<"constncpy\n";
}
    ~Strtype() {//деструктор
        delete [] p;
        cout<<"destructor\n";
    }
    char *get() {return p;}
};
void show(Strtype x)
{
    char *s;
    s=x.get();
    cout << s << "\n";}
void main(){
    Strtype a("Hello"), b("There");//constr1,constr1
    show(a);//конструктор копирования constncpy,Hello,destr
    show(b);//конструктор копирования constncpy,There,destr
    while(!kbhit());
}

```

Когда функция *show()* завершается и *x* выходит из области видимости, то освобождается память *x.p*, однако это не та память которая используется передаваемым в функцию объектом.

Организация внешнего доступа к компонентам класса. Функции “друзья” класса.

В языке C++ одна и та же функция не может быть компонентом двух разных классов. Можно определить обычную функцию языка C и предоставить ей право доступа к элементам класса типа *private, protected*. Для этого надо в описании класса поместить заголовок функции, перед которым поставить ключевое слово *friend*. Например:

```

//функция same_color сравнивает цвет прямоугольника и линии
class line;
class box {
    int color; // цвет прямоугольника
    int upx, upy; // левый верхний угол

```

```

    int lowx, lowy; //правый нижний угол
public:
    friend int same_color(line l, box b);
    void set_color(int c);
    void def_box(int x1, int y1, int x2, int y2);
};
class line {
    int color;
    int startx, starty;
    int endx, endy;
public:
    friend int same_color(line l, box b);
    void set_color(int c);
    void defline(int x, int y, int l);
};
int same_color(line l, box b){
    if(l.color==b.color) return 1;
return 0;
}

```

Если не использовать *friend* функцию, то необходимо сначала вернуть цвета, а затем написать функцию их сравнения. В приведенном примере класс *box* использует класс *line*, объявленный позже. Поэтому перед объявлением класса *box* необходимо поставить предварительную ссылку *class имя_класса*.

Хотя дружественная функция “знает” элементы класса, доступ к ним возможен только через объект класса, т.к. указатель *this* ей не передается. В следующем примере *friend*–функция использует указатель на класс в качестве параметра и для доступа к элементам *i* и *k* через указатель. Прямой доступ к ним невозможен.

```

class X{int i,k;
    public:
    void mem_func(int);
    friend void fr_func(X*,int);
};
void X::mem_func(int a)

```

```

    {i=a;k=a;}//прямой доступ
void fr_func(X *xptr,int a) {
xptr->i=a; //доступ к i через указатель
k=a;}// ошибка k=a
void main(){
    X xobj;//объявление объекта
fr_func(&xobj,6);//вызов дружественной функции
xobj.mem_func(6);//вызов члена класса
}

```

Можно объявить функции-члены класса дружественными для другого класса. В следующем примере функция *mem_f()* получает доступ к защищенным полям класса *y*.

```

class X{//...
    void mem_f(void);    };
class y{int i;
    friend void X::mem_f(void);
    };

```

Операторы-функции

Операторы-функции используются для введения операций, связываемых с символами:

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

Оператор-функция является членом класса или дружественным (*friend*) классу. Общая форма оператор-функции члена класса:

```

возвращаемый_тип имя_класса::operator#(список_аргум)
{
//выполняемые действия
}

```

После этого вместо *operator#(a,b)* можно писать *a#b*. Здесь # представляет один из введенных выше символов. Примерами являются операторы *>>*, *<<* перегружаемые для ввода/вывода. Отметим, что при перегрузке нельзя менять приоритет операторов и число операндов. Если оператор-функция член класса перегружает бинарный оператор, у

функции будет только один параметр-объект, стоящий справа от знака оператора. Объект слева вызывает оператор-функцию и передается неявно с помощью указателя *this*. Например:

```
// перегрузка +, = и ++ для класса Coord (p25)
#include <iostream.h>
class Coord {
    int x, y; // значения координат
public:
    Coord(int i=0, int j=0) { x = i; y = j; }
    void get_xy(int &i, int &j) { i = x; j = y; }
    Coord operator+(Coord ob2);
    Coord operator=(Coord ob2);
    Coord operator++();
};
Coord Coord::operator+(Coord ob2) //перегрузка +
{
    Coord temp;
    temp.x = x + ob2.x; //temp.x=this->x+obr.x
    temp.y = y + ob2.y; //temp.y=this->y+obr.y
    return temp;
}
Coord Coord::operator=(Coord ob2) //перегрузка =
{
    x = ob2.x; //this->x=obr.x
    y = ob2.y; //this->y=obr.y
    return *this; //возвращение объекта, которому присвоено значение
}
Coord Coord::operator++() //перегрузка ++, унарный оператор
{
    x++; y++;
    return *this;
}

void main()
{
    Coord o1(10, 10), o2(5, 3), o3;
    int x, y;
    o3 = o1 + o2; //сложение двух объектов - вызов operator+()
    o3.get_xy(x, y);
    cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";
}
```

```

    o3 = o1; //присваивание объектов
    o3.get_xy(x, y);
    cout << "(o3 = o1) X: " << x << ", Y: " << y << "\n";
    ++o1; //инкрементация объекта
    o1.get_xy(x, y);
    cout << "(++o1 ) X: " << x << ", Y: " << y << "\n";
    cin>>x;
}

```

Результат: (o1+o2) X:15 ,y:13
 (o3=o1) X:10, y:10
 (++o1) X:11, Y:11

При перегрузке унарного оператора ++. Параметр-объект передается через указатель *this*.

В следующем примере вводится класс “множество” и операции && - пересечения, << - вывод множества.

```

// перегрузка && и << для класса Set(множество) (p26)
#include <iostream.h>
#include <conio.h>
class Set{
    char *pi;//указатель на множество элементов типа char
public:
    Set(char *pl)//конструктор
    { pi=new char[strlen(pl)+1];
      strcpy(pi,pl);
    }
    Set &operator &&(Set &); //перегрузка &&-пересечение
    friend ostream &operator<<(ostream &stream,Set &ob);//перегрузка<<
    ~Set(){delete pi;}//деструктор
};
Set& Set::operator &&(Set &s)//пересечение
{int l=0;
  for (int j=0;pi[j]!=0;j++)
    for (int k=0;s.pi[k]!=0;k++)
      if (pi[j]==s.pi[k])
        {pi[l]=pi[j];

```

```

l++;
break;}
    pi[l]=0;
return *this;
}
ostream &operator<<(ostream &stream, Set &ob)
{
    stream << ob.pi << '\n';/*перезгрузка вывода, cout-stream,ob-
объект вывода*/
    return stream;
}
void main()
{ Set s1="1f2bg5e6",s2="abcdef";
  Set s3=s2;
  cout<<(s1&& s2)<<endl;//результат fbe
  cout<<s3<<endl;//результат abcdef
    while(!kbhit());
}

```

Оператор присваивания

Если одному объекту присваивается значение другого объекта, происходит его копирование. Если побитовое копирование нежелательно из-за того, что копии объектов ссылаются на одну и ту же динамическую память, можно использовать собственную функцию копирования *operator=()*. Рассмотрим пример:

```

Strtype &Strtype::operator=(Strtype &ob)
{    //выяснение необходимости дополнительной памяти
if(l < ob.l) {//требуется выделение дополнительной памяти
    delete[] p;
    p = new char [ob.l+1];
    if(!p) {
cout << "ошибка выделения памяти \n";exit(1);}
    }
    l = ob.l;
    strcpy(p, ob.p);
}

```

```

        return *this;//
    }

```

Здесь *operator=()* возвращает ссылку на объект. Параметром также является ссылка, что необходимо для запрещения создания копии объекта, стоящего справа от операции присваивания. Добавление этого оператора в класс, объявленный в примере *p24* позволяет копировать объекты с помощью оператора присваивания.

Использование дружественных оператор-функций

В дружественную функцию не передается оператор *this*, поэтому для унарного оператора передается один параметр, для бинарного – два параметра.

Оператор присваивания не может быть дружественной функцией а только членом класса. В следующем примере рассматривается использование дружественных оператор – функций `+`, `*`, `/`, `<<`, `>>` для класса *complex*.

```

// friend-функции для класса Complex (p27)
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
class complex {
    double re, im;
public:
    complex (double r=0,double i=0)//конструктор
    {re=r;im=i;}
    friend complex operator+(complex,complex);
    friend complex operator*(complex,complex);
    friend complex operator/(complex,complex);
    friend ostream & operator<<(ostream &stream, complex ob);
    friend istream & operator>>(istream &stream, complex ob);
};
ostream &operator<<(ostream &stream, complex ob)
{
    stream << "(" << ob.re << ", " << ob.im << ")" << '\n';
}

```



```

        return stream;
    }
istream &operator>>(istream &stream, complex &ob)
{
    stream >> ob.re >> ob.im;
    return stream;
}
complex operator+(complex a1, complex a2){
    return complex(a1.re+a2.re,a1.im+a2.im);
}
complex operator*(complex a1, complex a2){
    return complex(a1.re*a2.re-a1.im*a2.im, a1.re*a2.im+a1.im*a2.re);
}
complex operator/(complex a1, complex a2){
    double r=a2.re,i=a2.im;
    double tr=r*r+i*i;
    return complex((a1.re*r+a2.im*i)/tr,(a1.im*r-a1.re*i)/tr);
}
void main()
{ complex a(1.,2.),b(2.,2.),c;
  c=a+b;
  cout<<c<<a*b<<a/b<<endl;
  cout<<"Enter complex number:";
  cin>>c;
  cout<<c;
  while(!kbhit());
}
//Вывод:(3,4) (-2,6) (0.75,0.25)
//Enter complex number:(5,7)<ввод>
//(5,7)

```

Операторы-функции, объявленные как *friend* внутри класса *complex*, не являются членами класса, но подобно членам они имеют доступ к скрытым элементам объекта типа *complex*. Объект класса при этом должен быть параметром, иначе функция не знает, с каким объектом работать (*this* не передается). Определенный в файле *complex.h* класс

complex представляет тип данных, включающий и другие операции над комплексными числами.

3. НАСЛЕДОВАНИЕ, ВИРТУАЛЬНЫЕ ФУНКЦИИ И ПОЛИМОРФИЗМ

Все члены класса C++ относятся к одному из разделов: *public*, *private*, *protected*. Члены раздела *protected* являются частными для базового и производного классов. Спецификаторы доступа могут располагаться в любом порядке и в любом количестве. При наследовании базового класса также может объявляться спецификатор доступа к членам базового класса.

```
class имя_класса: доступ имя_базового_класса
{
//члены класса
}
```

Спецификатор *доступ* базового класса при наследовании может принимать одно из трех значений: *public* (по умолчанию), *private*, *protected*. Если спецификатор принимает значение *public*, то все члены разделов *public* и *protected* базового класса становятся членами разделов *public* и *protected* производного класса. Члены раздела *private* (частные) недоступны для производного класса. Если доступ имеет значение *private*, то все члены разделов *public* и *protected* становятся членами раздела *private* производного класса. Если доступ имеет значение *protected*, то все члены разделов *public*, *protected* становятся членами раздела *protected* производного класса. Рассмотрим пример:

```
// поведение protected-атрибутов при наследовании (p28)
#include <iostream.h>
#include <conio.h>
class X {
protected://обязательно для наследования
    int i,j;
public:
    void set_ij() {
```

```

    cout << "Введите два числа: ";
    cin >> i >> j;
}
void put_ij() { cout << i << " " << j << "\n"; }
};

class Y : public X { // в классе Y, i и j из X - защищенные данные
    int k;
public:
    int get_k() { return k; }
    void make_k() { k = i*j; }
};
/*Z имеет доступ к i и j из X, но не к частному k класса Y */
class Z : public Y {
public:
    void f()
    { i = 2; j = 3; } // i и j доступны отсюда
};

void main()
{ Y v1;
  Z v2;
  v1.set_ij();
  v1.put_ij();
  v1.make_k();
  cout << v1.get_k() << "\n";
  v2.f();
  v2.put_ij();
  while(!kbhit());
}
Вывод: Введите два числа:5 6
      5 6
      30
      2 3

```

Конструкторы и деструкторы производных классов

Если у базового и производного классов имеются конструкторы и деструкторы, то конструкторы выполняются в порядке наследования, а деструкторы в обратном порядке. При необходимости передачи аргументов конструктору базового класса при использовании производного класса используется следующий синтаксис:

```
конструктор_производного(арг):конструктор_базового(арг)  
{  
    //тело конструктора производного класса  
}
```

Рассмотрим пример:

```
// передача параметров конструктором в базовый класс (p29)  
#include <iostream.h>  
#include <conio.h>  
class base {  
    int i;  
public:  
    base(int n) {  
        cout << "constructor1 \n";  
        i = n;  
    }  
    ~base() {cout<<"destructor1 \n"; }  
    void showi() { cout << i << '\n'; }  
};  
class derived : public base {  
    int j;  
public:  
    derived(int n):base(n+5)//передача аргумента в базовый класс  
        {cout << "constructor2 \n";  
            j = n;  
        }  
    ~derived(){cout<<"destructor2\n"; }  
        void showj() { cout << j << '\n'; }  
};  
main()
```

```

{
    derived o(3);
    o.showi();
    o.showj();
    while(!kbhit());
    return 0;
}
Вывод:constructor1
      constructor2
          8
          3
      destructor2
      destructor1

```

Множественное наследование

Один класс может наследовать атрибуты двух и более базовых классов, которые перечисляются после двоеточия через запятую. Если базовые классы содержат конструкторы, то они вызываются поочередно в порядке перечисления.

Пример:

```

// множественное наследование (p30)
#include <iostream.h>
class X {
protected:
    int a;
public:
    X() {
        a = 10;
        cout << "инициализация X\n";
    }
};
class Y {
protected:
    int b;
public:

```

```

Y() {
    cout << "инициализация Y\n";
    b = 20;
}
};
// Z наследует как от X, так и от Y
class Z : public X, public Y {
public:
    Z() { cout << "инициализация Z\n"; }
    int make_ab() { return a*b; }
};
int main()
{
    Z i;
    cout << i.make_ab();
    return 0;
}

```

Результат:
инициализация X
инициализация Y
инициализация Z
200

В C++ указатели и ссылки на базовый класс могут быть использованы для ссылок на объект производного класса. Если параметр функции является ссылкой на базовый класс, то аргументом функции может быть как объект базового класса, так и объект производного класса.

Виртуальные функции и полиморфизм

Механизм виртуальных функций в ООП используется для реализации полиморфизма: создания метода, предназначенного для работы с различными объектами за счет механизма позднего связывания (*late binding*). Виртуальные функции объявляются в базовом и производных классах с ключевым словом **virtual**. При этом каждый объект класса, управляемого из базового класса с виртуальными функциями, содержит указатель на *vmtbl* (*virtual method table*), содержащую адреса

виртуальных функций. Эти адреса устанавливаются в адреса нужных для данного объекта функций во время выполнения.

В С++ эффективно работает механизм перегрузки функций: функции с одним и тем же именем, но с различными типами аргументов считаются разными. В отличие от перегружаемых функций виртуальные функции объявляются в порожденных классах с тем же именем, возвращаемым значением и типом аргументов. Если различны типы аргументов, виртуальный механизм игнорируется. Тип возвращаемого значения переопределить нельзя.

Основная идея в использовании виртуальных функций состоит в следующем: виртуальная функция может быть объявлена в базовом классе, а затем переопределена в каждом производном классе. При этом доступ через указатель на объект базового класса осуществляется к этой функции из базового класса, а доступ через указатель на объект производного класса осуществляется к этой функции из производного класса. То же происходит при передаче функции объекта производного класса, если аргумент объявлен как базовый класс. Оба варианта рассмотрены в следующем примере:

```
/*указатель на базовый класс и ссылка используются для доступа  
к виртуальной функции. (p31)*/  
#include <iostream.h>  
#include <conio.h>  
class Base {  
public:  
    virtual void name() { // определение виртуальной функции  
        cout << "База\n";  
    }  
};  
class first_d : public Base {  
public:  
    void name() { // определение name() относительно first_d  
        cout << "Первое дифференцирование\n";  
    }  
};  
class second_d : public Base {
```

```

public:
    void name() { // определение name() относительно second_d
        cout << "Второе дифференцирование\n";
    }
};
// использует в качестве параметра ссылку на объект базового
// класса.
void show_name(Base &r) {
    r.name();
}
int main()
{
    Base base_obj;
    Base* pb;
    pb=&base_obj;
    pb->name();//base_obj.name()
    first_d first_obj;
    pb=&first_obj;
    pb->name();//first_obj.name()
    second_d second_obj;
    pb=&second_obj;
    pb->name(); //second_obj.name()
    show_name(base_obj); // доступ к Base's name()
    show_name(first_obj); // доступ к first_d's name()
    show_name(second_obj); // доступ к second_d's name()
    while(!kbhit());
    return 0;
}

```

Вывод: База

Первое дифференцирование

Второе дифференцирование

База

Первое дифференцирование

Второе дифференцирование

В следующем примере рассматривается реализация стека и очереди на основе связанного списка, при этом очередь и стек являются потомками класса *list*:

```
//классы, наследование и виртуальные функции  
//создание класса родовой список для целых(p32)  
#include <iostream.h>  
#include <stdlib.h>  
class list {  
public:  
list *head; //указатель на начало списка  
list *tail; //указатель на конец списка  
list *next; //указатель на следующий элемент  
int num; //число для хранения  
list () { head = tail = next = NULL; }  
virtual void store(int i) = 0; /*абстрактная базовая виртуальная  
функция*/  
virtual int retrieve() = 0; /*абстрактная базовая виртуальная  
функция*/  
};  
//создание типа очередь на базе списка  
class queue : public list {  
public:  
void store(int i);  
int retrieve();  
queue operator+(int i) { store(i); return *this; }  
int operator --(int unused) { return retrieve(); }  
};  
void queue::store(int i)  
{  
list *item;  
item = new queue;  
if(!item) {  
cout << "ошибка выделения памяти\n"; exit(1); }  
item -> num = i;  
//добавление в конец списка  
if(tail) tail -> next = item;
```

```

        tail = item;
        item -> next = NULL;
        if(!head) head = tail;
    }
    int queue::retrieve()
    {
        int i;
        list *p;
        if(!head) {cout << "список пуст\n";return 0;    }
        //удаление из начала списка
        i = head -> num;
        p = head;
        head = head -> next;
        delete p;
        return i;
    }
    //создание типа стек на базе списка
    class stack : public list {
    public:
        void store(int i);
        int retrieve();
        stack operator+(int i) { store(i); return *this; }
        int operator --(int unused) { return retrieve(); }
    };
    void stack::store(int i)
    {
        list *item;
        item = new stack;
        if(!item) {
            cout << "ошибка выделения памяти\n";
            exit(1);}
        item -> num = i;
        //добавление в начало списка, как в стеке
        if(head) item -> next = head;
        head = item;
        if(!tail) tail = head;
    }

```

```

int stack::retrieve()
{
    int i;
    list *p;
    if(!head) {cout << "список пуст\n";return 0; }
    //удаление из начала списка
    i = head -> num;
    p = head;
    head = head -> next;
    delete p;
    return i;
}
main()
{
    list *p;
    //демонстрация очереди
    queue q_ob;
    p = &q_ob; //указывает на очередь
    q_ob + 1;
    q_ob + 2;
    q_ob + 3;
    cout << "очередь : ";
    cout << q_ob --;
    cout << q_ob --;
    cout << q_ob --;
    cout << '\n';
    //демонстрация стека
    stack s_ob;
    p = &s_ob; //указывает на стек
    s_ob + 1;
    s_ob + 2;
    s_ob + 3;
    cout << "стек: ";
    cout << s_ob --;
    cout << s_ob --;
    cout << s_ob --;
    cout << '\n';
}

```

```

        return 0;
    }

```

Абстрактные классы и чисто абстрактные виртуальные функции

Абстрактные классы – это классы, содержащие чисто абстрактные виртуальные функции. Чисто абстрактные виртуальные функции при объявлении в классе приравниваются к нулю. Абстрактные классы используются только для наследования, так как объекты таких классов не могут быть определены.

```

// объявление абстрактного класса (p33)
#include<iostream.h>
class shape { //абстрактный
    int xb,yb,xc,yc;
public:
    shape(int hb,int vb,int hc,int vc)://конструктор
    xb(hb),yb(vb),xc(hc),yc(vc){}
    virtual void move(int,int)=0; //абстрактный
    virtual void copy(int,int)=0; //абстрактный
    virtual void rotate(int)=0; //абстрактный
};

```

Абстрактные базовые классы используются для создания общего для множества иерархических классов интерфейса. В следующем примере рассматриваются два класса с общим интерфейсом. Вызов функций осуществляется через общий интерфейс с помощью указателей на соответствующую *vtbl* - таблицу виртуальных методов.

```

// классы с общим интерфейсом (p34)
#include <iostream.h>
#define interface struct
interface IX {
    virtual void _stdcall FX1()=0;
    virtual void _stdcall FX2()=0;
};
class CA:public IX{

```

```

public:
virtual void _stdcall FX1(){cout<<"CA::FX1"<<endl;}
virtual void _stdcall FX2(){cout<<"CA::FX2"<<endl;}
};
class CB:public IX{
public:
virtual void _stdcall FX1(){cout<<"CB::FX1"<<endl;}
virtual void _stdcall FX2(){cout<<"CB::FX2"<<endl;}
};
void ff(IX* pix)
{pix->FX1();
pix->FX2();}
void main()
{CA* pA=new CA;//создание экземпляра CA
CB* pB=new CB;//создание экземпляра CB
IX* pix=pA;
ff(pix);//вызов методов CA
pix=pB;
ff(pix);//вызов методов CB
}

```

Подобный интерфейс используется в компонентной *COM* – технологии, если кроме этого наследует интерфейсе *IUnknown* и поддерживает три функции, выдающую информацию об интерфейсе:

```

interface IUnknown{
virtual HRESULT _stdcall QueryInterface(const IID& id,void **pv)=0;
virtual ULONG _stdcall Addref()=0;
virtual ULONG _stdcall Release()=0;
};

```

Здесь *_stdcall* означает поддержку соглашения о вызовах *Pascal*.

4. МАССИВЫ ОБЪЕКТОВ, УКАЗАТЕЛИ И ССЫЛКИ

Массивы объектов создаются так же, как и массивы переменных. Если класс содержит конструктор, массив может быть инициализирован,

причем конструктор вызывается столько раз, сколько задается элементов массива.

```
// создание и вывод массива объектов (p35)
#include<iostream.h>
class samp{
    int a;
    public:
        samp(int n){a=n;cout<<"constructor\n";}
        int get_a(){return a;}
};
main()
{
    samp ob[4]={1,2,3,4};
    samp *pob=ob;
    int i;
    for(i=0;i<4;i++) cout<<ob[i].get_a()<<' ';
    cout<<"\n";
    cout<<pob->get_a();
    return 0;
}
```

Программа выведет “1,2,3,4” , конструктор вызывается четыре раза, затем еще раз для указателя. Список инициализации – это сокращение общей конструкции:

```
samp ob[4]={samp(1),samp(2),samp(3),samp(4)};
```

Такая конструкция становится основной, если конструктор имеет два и более аргумента. Например:

```
samp ob[4]={samp(1,2),samp(3,4),samp(5,6),samp(7,8)};
```

При создании динамических объектов используется оператор *new*, который вызывает конструктор и производит инициализацию. Для разрушения динамического объекта используется оператор *delete*, который может помещаться в деструкторе. Например:

```

// динамический массив объектов (р36)
#include <iostream.h>
class samp {
    int i, j;
public:
    samp(){cout<<"конструктор2\n";}
    samp(int a,int b){
        i=a;
        j=b;
        cout<<"конструктор3\n";}
    void set_ij(int a, int b) { i = a; j = b; }
    ~samp() { cout << "удаление...\n"; }
    int get() { return i*j; }
};
main()
{
    samp *p01;
    int i;
    p01 = new samp(6,5);
    samp *p02;
    p02 = new samp[3];
    if(!p01||!p02) {cout << "ошибка выделения памяти\n";    return
1;}
    for(i=0; i<3; i++)
    {
        p02[i].set_ij(i, i);
        cout << "p02[" << i << "]= "<<p02[i].get()<<"\n";
    }
    cout << p01->get() << "\n";
    delete p01;
    delete [] p02;
    return 0;
}

```

результат:
конструктор1
конструктор2

```
конструктор2
конструктор2
p02[0]=0
p02[1]=1
p02[2]=4
30
удаление...
удаление...
удаление...
удаление...
```

Деструктор вызывается 4 раза, по одному разу на каждый элемент массива и один раз для объекта *p01*.

Ссылки

Ссылка является скрытым указателем и работает как другое имя переменной. Ссылку можно передать в функцию и вернуть из функции. При передаче объекта через ссылку в функцию передается адрес объекта и не делается его копия. Это уменьшает вероятность ошибок, связанных с выделением динамической памяти и вызовом деструктора.

При передаче функции параметра объекта может возникнуть ошибка из-за разрушения деструктором копии объекта, которая должна быть исправлена созданием конструктора копирования. В этой ситуации лучше использовать функцию, возвращающую ссылку на объект. Функция может возвращать ссылку на объект. Например:

```
// пример защищенного двумерного массива (p37)
#include <iostream.h>
#include <stdlib.h>
class array {
    int isize, jsize;
    int *p;
public:
    array::array(int i, int j)
    {
```



```

    p = new int [ i * j ];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);    }
    isize = i;    jsize = j;
}

int &put(int i, int j);
int get(int i, int j);
};

// Запись значения в массив
int &array::put(int i, int j)
{
    if(i<0 || i>=isize || j<0 || j>=jsize) {
        cout << "Ошибка, нарушены границы массива!!!\n";
        exit(1);
    }
    return p[i * jsize + j]; // возврат ссылки на p[ i ]
}

// Получение значения из массива
int array::get(int i, int j)
{
    if(i<0 || i>=isize || j<0 || j>=jsize) {
        cout << "Ошибка, нарушены границы массива!!!\n";
        exit(1);
    }
    return p[i * jsize + j]; // возврат символа
}

int main()
{
    array a(2, 3);
    int i, j;
    for(i=0; i<2; i++)
        for(j=0; j<3; j++)
            a.put(i, j) = i + j;
    for(i=0; i<2; i++)
        for(j=0; j<3; j++)
            cout << a.get(i, j) << ' ';
}

```

```

        // генерация ошибки нарушения границ массива
        a.put(10, 10);
        return 0;
    }

```

5. ШАБЛОНЫ И ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

С помощью шаблонов можно создавать родовые (*generic*) функции и классы. Родовая функция определяет базовый набор операций, которые будут применяться к разным типам данных, получаемых функцией в качестве параметра.

Определение функции с ключевым словом *template* (шаблон) имеет вид:

```

template<class Ttype>тип имя_функции(список аргументов)
{//тело функции}

```

Здесь *Ttype* – фиктивный тип, который используется при объявлении аргументов и локальных переменных функции. Компилятор заменит этот фиктивный тип на один из реальных типов и создаст соответственно несколько перегружаемых функций. При этом перегружаемые функции являются ограниченными, поскольку выполняют одни и те же действия. Например:

```

// применение класса-шаблона (p38)
#include <iostream.h>
#include <string.h>
template<class X> int find(X object, X *list, int size)
{
    int i;
    for(i=0; i<size; i++)
        if(object == list[i]) return i;
    return -1;
}
int main()
{

```

```

        int a[]={1, 2, 3, 4};
        char *c="это проверка";
        double d[]={1.1, 2.2, 3.3};
        cout << find(3, a, 4) << endl;
        cout << find('a', c, (int) strlen(c))<< endl;
        cout << find(0.0, d, 3);
    return 0;
}

```

Компилятор автоматически создает три перегружаемые функции *find()*, соответствующие типом передаваемых аргументов.

С помощью родового класса можно создать класс, реализующий стек, очередь, дерево и т.д. для любых типов данных. Компилятор будет генерировать правильный тип объекта на основе типа, задаваемого при создании объекта. Общая форма объявления родового класса:

```

template <class Ttype> class имя_класса {
//имена класса, поля класса
}

```

Здесь *Ttype* – фиктивное имя типа, которое заменится. Например:

```

// простой родовой связанный список(р39)
#include <iostream.h>
template <class data_t> class list {
    data_t data;
    list *next;
public:
    list (data_t d);
    void add(list *node) {node->next = this; next=0;}
    list *getnext() {return next;}
    data_t getdata() {return data;}
};
template <class data_t> list<data_t>::list(data_t d)
{
    data=d;
    next=0;
}

```

```

main()
{
list<char> start('a');//создается объект с реальным типом данных
    list<char> *p, *last;
    int i;
    // создание списка
    last=&start;
    for(i=1; i<26; i++) {
        p=new list<char> ('a'+i);
        p->add(last);
        last=p;
    }
    // вывод списка
    p=&start;
    while(p) {
cout << p->getdata();
        p=p->getnext();
    }
    return 0;
}

```

С помощью простого объявления можно создать другой объект, например, для хранения целых.

```
list <int> int_start(1);
```

В список можно поместить структуры или другие объекты. Класс-шаблон может иметь больше одного родового типа данных.

```
template<class Type1,class Type2> class m {Type1 a;Ttype2 b;}
```

Обработка исключительных ситуаций

В программах C++ следует использовать механизм обработки исключительных ситуаций (ошибок). Операторы программы, во время выполнения которых обеспечивается обработка исключительных ситуаций, располагаются в блоке *try*. Перехватывается и обрабатывается

исключительная ситуация в блоке *catch*. Форма операторов *try/catch* следующая:

```
try {  
    //блок try  
}  
catch(type1 arg){  
    //блок catch  
}
```

С блоком *try* связывается несколько блоков *catch*. Выполняется тот блок *catch*, для которого тип данных соответствует типу возникшей исключительной ситуации. При этом ее значение присваивается аргументу *catch*. Если ошибка имеет место внутри блока *try*, она может генерироваться с помощью *throw*.

```
// генерация и перехват ошибки (p40)  
#include <iostream.h>  
main()  
{  
    try  
    {    // начало блока try  
        cout << "Внутри блока try\n";  
        throw 10;    // генерация ошибки 10  
        cout << "Это выполнено не будет\n";  
    }  
    catch (int i)  
    { // перехват ошибки  
        cout << "Перехвачена ошибка номер: "<< i << "\n";  
    }  
    return 0;  
}  
Будет выведено:  
Внутри блока try  
Перехвачена ошибка: 10
```

Оператор **throw** генерирует ошибку, после чего управление передано блоку **catch**. Если заменить тип ошибки **int** на **double**, ошибка перехвачена не будет. Если надо перехватить все исключительные ситуации независимо от типа, то используется

```
catch(...)  
{  
//тело  
}
```

Многоточие соответствует любому типу данных. Для функций, вызываемых из блока **try** можно указать число типов исключительных ситуаций, которые будет генерировать функция:

```
тип имя (список аргументов)  
throw(список типов)  
{  
//тело  
}
```

6.ПОТОКИ И КЛАССЫ ВВОДА/ВЫВОДА

В С++ ввод и вывод осуществляется через потоки - логические устройства, которые передают и принимают данные и связываются с физическими устройствами с помощью системы ввода/вывода. При запуске программы автоматически открываются стандартные потоки **cin**, **cout**, **cerr**, **clog**. Последние два потока используются для вывода сообщения об ошибках. В файле **iostream.h** заданы классы: базовый **streambuf**, порожденный класс **ios**, обеспечивающий контроль ошибок и информацию о потоках; порожденные классы ввода – **istream**, вывода – **ostream**, ввода/вывода – **iostream**. Потоки являются объектами соответствующих классов.

Операторы ввода/вывода определены для основных встроенных типов как << и >>. Для оператора вывода (вставки) можно использовать манипуляторы (форматы), например

```
cout<<hex<<100<<oct<<100dec<<100;
```

Сам оператор << можно перезагрузить, как в следующем примере:

```
/*создание недружественной функции-вставки для объектов coord
(p41)*/
#include <iostream.h>
class coord{
    public:
        int x,y;//должны быть открытыми
coord() {x=0;y=0;}
coord(int i,int j) {x=i;y=j;}
};
//функция вставки для объектов класса coord
ostream &operator<<(ostream &stream ,coord ob){
    stream <<ob.x<<","<<ob.y<<'\n';
    return stream;
}
main(){
    coord a(1,1), b(10,23);
    cout <<a<<b;
    return 0;
}
```

Оператор ввода (извлечения) также можно перезагрузить

```
istream &operator>>(istream &stream, имя_класса ob)
{
//тело функций ввода
return stream;
}
```

Для реализации файлового ввода/вывода необходимо включить файл *fstream.h*, содержащий производные от *istream* и *ostream* классы *ifstream*, *ofstream* и *fstream* и объявить соответствующие объекты. Например:

```
ifstream in;//ВВОД
ofstream out;//ВЫВОД
fstream io;//ВВОД/ВЫВОД
```

После объявления потоков производится открытие файла, связывающее его с потоком с помощью функции *open()*. Файл закрывается с помощью функции *close()*. Метод *int eof()* возвращает ненулевое значение, если достигнут конец файла при вводе, иначе возвращает 0. Например:

```
// соединение с файлов и чтение его содержимого (p42)
#include <iostream.h>
#include <fstream.h>
main()
{
    ofstream fout("test"); /* создание обычного файла вывода с
помощью конструктора */
    if(!fout) { cout << "Файл открыть невозможно\n";
return 1; }
    fout << "Привет!\n";
    fout << 100 << ' ' << hex << 100 << endl;
    fout.close();
    ifstream fin("test"); // открытие обычного файла ввода
    if(!fin) {cout << "Файл открыть невозможно\n";return 1;}
    char str[80];
    int i;
    fin >> str >> i;
    cout << str << ' ' << i << endl;
    fin.close();
    return 0;
}
```

Для чтения/записи здесь можно использовать перегружаемые оператор-функции >> и <<, основанные на вызове *fprint()* и *fscan()*. При работе с бинарными файлами используются функции ввода/вывода одного символа:

```
istream &get(char &ch);
ostream &put(char ch);
```

Прототип функции *open()*:
void open (char *filename,int mode,int access)

где *filename* – имя файла ,включающее путь; *mode* – режим открытия файла,

mode:

ios::in – открытие файла для чтения.

ios::out – открытие для записи.

ios::binari – открытие файла в двоичном режиме, по умолчанию в текстовом;

access: 0 – файл со свободным доступом.

1 –только для чтения.

8 – архивный файл.

Пример:

ofstream out;

out.open("test.dat",ios::out,0);

Параметры можно задать по умолчанию.

out.open("test.dat");//будет тоже самое

Пример открытия файла с использованием аргументов командной строки:

// чтение произвольного файла (p43)

#include <iostream.h>

#include <fstream.h>

main(int argc, char *argv[])

{

char ch;

if(argc!=2) {cout << "Использование:WRITE<имя_файла>\n";return 1;}

ofstream out(argv[1]);

if(!out) {cout << "Файл открыть невозможно\n";return 1;}

cout << "Для остановки введите символ \$\n";

do {

cout << ": ";

cin.get(ch);

out.put(ch);

} while (ch!='\$');

out.close();

```

    return 0;
}

```

Для записи и считывания блоков двоичных данных используются функции, которые считывают или записывают *num* байт в буфер или из буфера.

```

istream &read(unsigned char *buf, int num)
ostream &write(const unsigned char *buf, int num)

```

Пример:

```

// (p44)
#include <iostream.h>
#include <fstream.h>
#include <string.h>
main()
{
    ofstream out("test");
    if(!out) {
        cout << "Файл открыть невозможно\n";
        return 1;}
    double num = 100.45;
    char str[ ] = "Это проверка";
    out.write((char *) &num, sizeof(double));
    out.write(str, strlen(str));
    out.close();
    return 0;
}

```

ПРИЛОЖЕНИЕ 1. Задания для выполнения

I

Создать класс Integer для чисел из *n* цифр (*n*>100). Реализовать операции умножения, деления, сложения, вычитания, присваивания.
 Создать класс Fraction (дроби). Реализовать операции.
 Создать класс Complex. Реализовать операции.
 Создать класс многочлен, наследующий одночлен.

Создать класс рациональный многочлен, наследующий многочлен.
Создать класс массив (максимальной размерн. и динамический) и производный класс - двумерный массив.

Создать класс Vector.

Создать класс Matrica.

Создать класс Stack.

Создать класс Очередь.

Создать класс Дерево.

Создать класс String. Создать производный класс Text. Реализовать операции ввода и вывода.

Создать класс Boolean

Создать класс Set (множество)

Создать классы Picture, Line, Circle, Rectangle

II

Создать те же классы, используя шаблоны классов, Exception и виртуальные функции

III

Создать Классы для Windows 95 (Окно, Кнопка, Меню, Диалог, Графический объект)

IV

Создать классы для Java

ПРИЛОЖЕНИЕ 2. Примеры программ

1_Stack:

//пример виртуальной функции print(), печатающей стек или

//список

//list2.h : список целых чисел

```
const max_elem=10;
```

```
class List {
```

```
protected:
```

```
int *list;//массив целых
```

```

    int nmax;//размер массива
    int nelem;//число элементов в массиве
public:
    List(int n=max_elem)//конструктор
    { list=new int[n];//создание массива list
      nmax=n;
      nelem=0;
    }
    ~List(){ delete list; }//деструктор, уничтожение list
    int put_elem(int,int);
    int get_elem(int &,int);
    void setn(int n) { nelem=n; }
    int getn() {return nelem;} //! Не исп.
    void incn() { if (nelem<nmax) ++nelem; } //! Не исп.
    int getmax() { return nmax; }
    virtual void print();
};

```

```

// list.cpp :реализация класса list
#include "list2.h"
#include <stdio.h>

```

```

int List::put_elem(int elem, int pos) {
    if(0<=pos && pos<nmax) {
        list[pos]=elem;
        return 0;
    }
    else return -1;
}

```

```

int List::get_elem(int &elem, int pos) {
    if(0<=pos && pos<nmax) {
        elem=list[pos];
        return 0;
    }
    else return -1;
}

```

```

void List::print() {
    for (int i=0;i<nelem;++i)
        printf("%d\n",list[i]);
}

```

```

//stack2.h : класс stack порождается из List
//#include "list.cpp"

```

```

class stack: public List {
    int top;
public:
    stack() {top=0; }//конструктор
    int push (int elem);
// int pop(int &elem);
    void print();
};

```

```

//stack.cpp : реализация класса stack
//#include <stdio.h>
//#include "stack2.h"

```

```

void stack::print() {
    for(int i=top-1;i>=0;--i)
        printf("%d\n", list[i]);
//другие функции
}

```

```

int stack::push (int elem) {
    if (put_elem(elem, top)==0) {
        top++;
        return 0;
    } else return -1;
}

```

```

//использование виртуальной функции print()
//#include <stdio.h>

```

```

#include <conio.h>
//#include "stack.cpp"

main() {
    stack s;//инициализация стека ! s(5) ???
    List l, *lp;
    int i=0;
    //занесение в стек чисел 1..5 !на самом деле 1..10
    while(s.push(i+1)==0)
        ++i;
    //занесение элементов в список
    l.put_elem(1,0);
    l.put_elem(2,1);
    l.setn(2);
    printf("stack:\n");
    lp=&s;
    lp->print();
    printf("list:\n");

    lp=&l;
    lp->print();
    while(!kbhit());
}
//на экране 5 элементов стека и 2 элемента списка

```

2_BD:

```

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

class B {
    char name[80];
public:
    void put_name(char *s) { strcpy(name, s); }
    void show_name() {cout << name << " "; }
};

```

```

class D : public B {
    char phone_num[80];
public:
    void put_phone(char *num) { strcpy(phone_num, num); }
    void show_phone() { cout << phone_num << "\n"; }
};

int main()
{
    B *pb;
    B B_ob;
    D *pd;
    D D_ob;
    pb = &B_ob;
    // доступ к B_class через указатель
    pb->put_name("Tomas Edison");
    // доступ к D_class через указатель
    pb = &D_ob;
    pb->put_name("Albert Einstein");
    // показать каждое имя соответствующего объекта
    B_ob.show_name();
    D_ob.show_name();
    cout << "\n";
    /*
    * поскольку put_phone и show_phone не являются частью базового
    класса,
    * они не доступны через указатель на базовый класс и доступ должен
    осуществляться
    * напрямую, или, как показано ниже, через указатель на
    порожденный класс
    */
    pd = &D_ob;
    pd->put_phone("555 555-1234");
    pb->show_name(); // в данной строке могут использоваться
                    // или pb, или pd
    pd->show_phone();

```

```

while(!kbhit());
return 0;
}

```

3_P:

```

//передача указателей и вызов через указатели
//виртуальной функции
#include <iostream.h>
#include <conio.h>

```

```

class A {
    int a;
public:
    A(int aa):a(aa) { }//конструктор
    virtual void display() { cout<<endl<<a; }
};

```

```

class B:public A{
    int b;//производный класс
public:
    B(int aa,int bb):b(bb),A(aa){}
    void display() {
        A::display();
        cout<<'\\t'<<b;
    }
};

```

```

void f(A *x) {
    x->display();
}

```

```

void main() {
    A a1(33),a2(44);
    B b1(1,2),b2(11,12);
    f(&a1);//33
    f(&a2);//44
}

```



```

f(&b1);//1 2
f(&b2);//11 12
while(!kbhit());
}

```

4_Templ:

/* Программа использует класс-шаблон и создаёт два класса: int и char;

Ввод и вывод происходит с использованием функций-членов класса и перегружаемых

операторов + и cout<<;

функция cout<< - дружественная.

***/**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

template <class X> class BaseStack //базовый класс-шаблон

{

protected:

X inf;

};

template <class X> class Stack : public BaseStack<X> //наследование класса-шаблона

{

Stack* top;

Stack* next;

public:

Stack();

~Stack();

void Add(X value);

void operator + (X value);

friend ostream &operator <<(ostream &stream,Stack<X>* p);

void Look();

```
};
```

```
template <class X> Stack<X>::Stack()//конструктор
{
    top=NULL;
    cout<<"Stack Initialized\n";
};
```

```
template <class X> Stack<X>::~~Stack()//деструктор
{ cout<<"Stack Destroyed\n";
}
```

```
template <class X> void Stack<X>::Add(X value)//добавление в стек-
//функция-член класса
{ Stack* newel;
  newel=(Stack<X>*)malloc(1);
  newel->next=top;
  newel->inf=value;
  top=newel;
};
```

```
template <class X> void Stack<X>::Look()//вывод стека-
//функция-член класса
{ Stack<X> *p;
  p=top;
  cout<<"Working of the function-member of class: ";
  while(p!=NULL) {
    cout<<p->inf<<" ";
    p=p->next;
  }
  cout<<"\n";
};
```

```
template <class X> /*Stack<X>*/ void Stack<X>::operator +(X value)
/*добавление в стек-перегружаемый оператор + */
{ Stack* temp;
  temp=(Stack<X>*)malloc(1);
```

```

temp->next=top;
temp->inf=value;
top=temp;
};

```

```

template <class X> ostream &operator << (ostream &stream, Stack<X>*
p)
/* вывод стека-перегружаемый оператор cout<< ,
  функция-друг класса */
{
p=p->top;
stream<<"Working of the function-friend of class: ";
while(p!=NULL) {
stream<<p->inf<<" ";
p=p->next;
}
stream<<"\n";
return stream;
};

```

```

void main()
{
Stack<int> a1,a2;//создание классов a1 и a2 с типом int
Stack<char> b1,b2;//создание классов b1 и b2 с типом char
int i;
char j;
for(i=0;i<=10;i++)
a1.Add(i);//добавление в стек a1 (функция-член класса)
for(i=10;i<=20;i++)
a2+i; //добавление в стек a2 (перегружаемый оператор +)
a2+57;
for(j='a';j<='m';j++)
b1.Add(j);//добавление в стек b1 (функция-член класса)
for(j='m';j<='p';j++)
b2+j; //добавление в стек b2 (перегружаемый оператор +)
a1.Look(); //вывод стеков a1 и b1 (функция-член класса)
b1.Look();
}

```

```
cout<<"\n";  
cout<<&a2; //вывод стеков a2 и b2 (перегружаемый  
//оператор cout<< - друг класса)  
cout<<&b2;  
getch();  
}
```

Учебное издание

Блинов Игорь Николаевич
Романчик Валерий Станиславович

ВВЕДЕНИЕ В C++

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО КУРСУ
“МЕТОДЫ ПРОГРАММИРОВАНИЯ”

Для студентов механико-математического факультета

В авторской редакции

Подписано в печать 6.03.2000. Формат 60х84/16. Бумага офсетная.
Печать офсетная. Усл.печ.л.2,79. Уч.-изд.л.1,59. Тираж 100 экз. Зак.

Белорусский государственный университет
Лицензия ЛВ №315 от 14.07.98.
220050, Минск, пр. Ф.Скорины, 4.

Отпечатано в Издательском центре БГУ,
220030, Минск, ул. Красноармейская, 6.