*Министерство науки и высшего образования Российской Федерации*

*АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ*

*Факультет математики и информатики*

**Т. А. Галаган**

# ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C++

***Часть 2***

Благовещенск
2022

ББК
Г

*Печатается по решению редакционно-издательского совета факультета математики и информатики Амурского государственного университета*

***Т.А. Галаган***

**Программирование на языке С++. Часть 2. Учебное пособие.** Для студентов направлений подготовки 01.03.02 Прикладная математика и информатика, 09.03.01 Информатика и вычислительная техника, 09.03.02 Информационные системы и технологии, 09.03.04 Программная инженерия очной формы обучения – Благовещенск: Амурский гос. ун-т, 2022.

Во второй части учебного пособия рассмотрены возможности языка программирования С++, позволяющие создавать собственные типы данных, обрабатывать данные, хранимые в файлах, основные принципы объектно-ориентированного подхода и его реализации средствами языка С++. Теоретический материал сопровожден примерами простых для понимания программ. Пособие предназначено для студентов, начинающих изучение программирования с нуля.

*Рецензенты:*

*Е.Ф Алутина, канд. физ.-мат. наук, доцент кафедры информатики и методики преподавания информатики Благовещенского государственного педагогического университета;*

*Е.М Веселова, канд. физ.-мат. наук, доцент кафедры математического анализа и моделирования Амурского государственного университета;*

# ВВЕДЕНИЕ

Учебное пособие посвящено популярному языку программирования C++. Это вторая часть пособия, которая является логическим продолжением его первой части.

Раскрыты и доступны для усвоения такие разделы, как: Типы данных создаваемые пользователем; Работа с файлами, Основы объектно-ориентированного программирования, Стандартная библиотека языка C++.

Каждый раздел включает изложение теоретического материала, сопровожденного примерами кодов программ и перечнем контрольных вопросов.

Пособие предназначено для самостоятельной работы студентов, изучающих курс «Программирование», обучающихся по направлению подготовки бакалавриата «Информатика и вычислительная техника» и по смежным направлениям.

Необходимо отметить, что данное пособие не является учебником языка C++ и не претендует на его полное освещение, хотя и использует C++ как средство обучения программированию. Его материал является подробным изложением лекций дисциплины «Программирование» и рассчитан на первоначальное изучение языка C++.

# Глава 1. Типы данных, определяемые пользователем

## Переименование типов

В C++ можно создавать новые типы данных только на основе уже существующих. Можно приписывать типу данных новое имя, используя ключевое слово typedef. Синтаксис такой процедуры следующий:

typedef тип новое имя типа;

Например,

```
typedef char* str;
```

Введенное таким образом имя можно использовать, как и имя стандартного типа. Тогда объявление вида

```
str name, sign;
```

аналогично объявлению

```
char *name, *sign;
```

На практике задание нового типа используется для облегчения понимания программы.

## Перечисляемые типы

Перечисляемый тип представляет собой набор целых чисел, определенный с помощью ключевого слова enum. Каждому числу набора приписывается имя.

Синтаксис:

enum имя типа { имя константы = целое значение, …};

Константы должны быть целыми и могут инициализироваться обычным способом. Указание инициализирующего значения необязательно. По-умолчанию первой в списке константе присваивается значение нуль. Последующие константы принимают значение на единицу большее предыдущего.

Имя типа также необязательно. Оно задается в том случае, если в программе требуется определить переменную данного типа. Тогда компилятор обеспечивает,

чтобы эти переменные принимали значение только из списка констант.

Примеры

```
enum Computer_Drives { floppy = 1, harddrivers = 2, cd_room = 4 };
enum Com_Number { first=1, second };    // здесь second равно 2
enum Seasson { Fall, Winter, Spring, Summer };
enum  { two = 2, tree, four, ten = 10, eleven, fifty= ten+ 40 };
// tree равно 3,  four - 4, fifty 50
```

С помощью первых трех объявлений можно задавать переменные. В последнем случае просто перечисляется набор констант.

При использовании арифметических операций перечисления преобразуются к целому типу.

## Структуры

Структуры позволяют определять новые типы данных путем логического группирования переменных различных типов. В отличие от массива, все элементы которого одного типа, структура может содержать элементы разных типов. Элементы структуры являются полями. Описание типов полей содержится в структурном шаблоне. Описание структурного шаблона выглядит следующим образом:

```
struct имя структуры {
                        имя типа1 идентификатор1;
                        имя типа2 идентификатор2;
                        ……..
                      } список переменных;
```

Например,

```
struct book {                  //описание структурного шаблона
  string title;
  string author;
```

```
  int page, year;
};
book b;          // структурная переменная b
book lib1, lib2,*ptb;
book library[100];   // массив структур
```

В примере представлено описание структурного шаблона. Структурный шаблон задает тип. После его определения можно смело объявлять переменные, вновь созданного типа. Правила объявления структурной переменной аналогичны правилам объявления простой переменной, т.е. указывается имя типа, за которым следуют имена переменных, разделяемые запятыми.

Например

```
struct book library;            // описание структурной переменной libry типа book
struct book lib1, lib2,*ptb;    //описание двух структурных переменных и указа-
                                 //теля на структурную переменную
struct book library[100];       //объявлен массив из 10 элементов типа book
```

Теперь каждая из объявленных переменных имеет поля title, author и page, year. При объявлении структурной переменной ключевое слово struct не является обязательным, но очень широко используется.

Можно совмещать в одном объявлении определение структурного шаблона и структурной переменной.

```
struct book{
string title;
string author;
int page, year;
} b1, b2;
```

Объявлены переменные b1 и b2 типа book.

Имя типа структуры можно не задавать. Как правило, это в случае, если предполагается использование структурного шаблона всего один раз. Тогда происходит объединение в один этап объявления структурного шаблона и структурной переменной. Например,

```
struct { float x, y;
} complex;
```

Элемент структуры может иметь базовый тип, либо быть массивом, указателем или, в свою очередь, структурой. Элемент структуры не может быть типом той структуры, в которую он входит, но он может быть объявлен указателем типа структуры, в которой он содержится.

Идентификаторы полей структуры должны различаться между собой, но идентификаторы различных структур могут совпадать.

Доступ к полям структурной переменной осуществляется через операцию выбора (точка). Например,

```
cin>>complex.x;
cout<< complex.y;
```

Как и любую другую переменную, структурную переменную можно инициализировать при объявлении. Тогда значения ее полей перечисляются в фигурных скобках в порядке их описания в шаблоне, через запятую.

```
struct book libry = {“Просто и ясно о Borland C++”, “Бруно Бабэ”, 400, 2005};
```

При инициализации массива структур следует заключать в фигурные скобки каждый элемент массива.

Как и к другим типам, структурам можно давать другое имя с помощью ключа typedef. Например,

```
typedef struct
                { char name[80];
                long anct_Num;
                } custInfo;
```

Объявление переменной этого типа выглядит следующим образом:

```
custInfo newCust;
```

Для переменных одного и того же структурного типа определена операция присваивания. При ее использовании происходит поэлементное копирование.

Пример программы.

Создать структурный шаблон, хранящий сведения о некотором человеке: фамилию, пол, возраст. Создать массив, содержащий сведения о 15 человек. Выбрать из списка фамилии мужчин в возрасте от 20 до 35 лет, причем их фамилия должна начинаться с введенной с клавиатуры буквы.

```
#include <iostream>
using namespace std;
int main( )
{ const int n=15;
 struct man {          //описание структурного шаблона
                       char family[20];
                       char gender;
                       int age;
                 };
 struct man people [n];                    //массив структурных переменных
 int i;
 char symbol;
 for ( i=0; i < n; i+ + )
        {cout<< “введите фамилию: ” ;
        cin>> peole[i].family;
        cout<< “введите возраст: ”;
        cin>> peole[i].age;
        cout<< “введите пол: м – мужской или ж – женский: ”;
        cin>> peole[i].gender;
        }
cout<< “ введите первую букву фамилии: ”;
cin << symbol;
```

```
for ( i=0; i < n; i++)
        { if  ( peole[i].family[0]= = symbol)
                if (peole[i].gender = = ‘м’)
                        if ( peole[i].age > 19)&&( peole[i].age < 36 )
                                cout<< peole[i].family <<endl;
        }
}
```

Полем структуры может быть другая структура. В этом случае структура называется иерархической.

```
struct DateType
{
        int month;
        int day;
        int year;
};
struct Statistic
{
        DateType lastServiced;
        int gurantee;
};
struct CarRec
{
        string descript;
        int number;
        float  cost;
        DateType  dataSale;
        Statistic  history;
};
CarRec Car, Cars[100];
```

Инструкция для обращения к полям внутренних структур строятся слева

направо, начиная с имени структурной переменной. Например, Car.dataSale.day – поле day структуры типа DateType, содержащейся в переменной Car типа CarRec.

## Указатели на структуру

Возможно объявление указателей на структуру, и тогда для получения доступа к отдельным элементам структуры применяется указатель и оператор –>.

```
struct complex  {
  int Re, Im;
} z1, z2;
complex *p = &z1;
cin >> p -> Re;
```

Рассмотрим еще один пример.

```
#include <stdio.h>
struct person {     int age;        char name[20];   };
 int main( )   {
    struct person kate = {31, "Kate"};
    struct person * p_kate = &kate;
    char * name = p_kate->name;
    //используем разные способы для обращения к элементам структуры
      int age = (*p_kate).age;
     printf("name = %s \t age = %d \n", name, age);
      p_kate->age = 32;   // изменили элемент age в структуре
    printf("name = %s \t age = %d \n", kate.name, kate.age);
    return 0;
}
```

Указатели используются и при создании динамических массивов.

Преимущество динамических массивов заключается в том, что размерность может быть переменной, т.е. объем памяти, выделяемой под массив, определяется на этапе выполнения программы. Доступ к элементам динамического массива

происходит обычным способом.

Аналогичным образом происходит и объявление массива структур, например,

```
struct complex { int x, y; }*ptr;
ptr=new complex[50];
…
delete [ ] ptr;
```

## Передача структур в функцию

Структуры передаются в функцию по значению, т. е. в функции модифицируются лишь копии исходных данных. В прототипе функции следует указать в качестве параметра структурную переменную. Так при объявлении переменной с использованием структурного шаблона из предыдущего примера, возможно объявление прототипа функции в виде:

```
void print (struct boat boat1);
```

Указывать ключевое слово struct необязательно. Если же вместо самой структуры передавать указатель на нее, то это может значительно ускорить выполнение программы.

```
void print (struct boat *boats);
```

Таким же способом происходит и передача массива структур в функцию.

Пример

```
#include <iostream>
#include <ctime>
#include <stdlib.h>
#include <iomanip>
#include <math.h>
using namespace std;
struct D //структурный шаблон хранения даты
{ int day, month,  year;
```

```
};

struct Pogoda //структурный шаблон хранения сведений о погоде
{ D date;
int dt,  nt,  osad, dav;
};
void input(Pogoda *p, int &b); //прототип функции ввода данных
void output(Pogoda *p, int b); // прототип функции вывода данных
void star( );
void leto(Pogoda *p, int b);
void zimden(Pogoda *p, int b);

int main( ) {
int z = 0, k = 0, i;
const int n = 256;
Pogoda* p = new Pogoda[n]; //динамический массив структур
do {              // организация меню
        system("cls");
        cout << "Select menu item:" << endl << "1-Input data" << endl;
        cout << "2-Output all data" <<endl << "3-Dannie o pogode v letnie mesyaci" <<endl;
        cout << "4-Samii teplii zimnii den'"<<endl;
        cout<< "5-Exit" << endl;
        cin >> k;
        switch(k)  {
        case 1: input(p,i); break;
        case 2: output(p,i); break;
        case 3: leto(p,i); break;
        case 4: zimden(p,i); break;
        case 5: z = 1; break;
        default: system("cls");
```

```
        }
} while(z != 1);
system("cls");
delete[ ] p;  //очищение динамической памяти
return 0;
}

void input(Pogoda *p, int&b) //функция ввода
{       system("cls");
do{
cout << "vvedite znachenie v diapazone ot 1 do 31"<<endl<<"Day: ";
cin >> p[b].date.day;
} while(p[b].date.day>31 || p[b].date.day<1);
do{
cout << "vvedite znachenie v diapazone ot 1 do 12"<<endl<<"Month: ";
cin >> p[b].date.month;
} while(p[b].date.month>12 || p[b].date.month<1);
do{
cout << "vvedite znachenie v diapazone ot 1970 do 2023"<<endl<<"Year: ";
cin >> p[b].date.year;
} while(p[b].date.year>2023 || p[b].date.year<1970);
do{
cout << "vvedite znachenie v diapazone ot -25 do 50"<<endl<<"Day Temp: ";
cin >> p[b].dt;
} while (p[b].dt>50 || p[b].dt<-25);
do{
cout << "vvedite znachenie v diapazone ot -40 do 25"<<endl<<"Night Temp: ";
cin >> p[b].nt;
} while (p[b].nt>25 || p[b].nt<-40);
do{
```

```
cout << "0-net osadkov 1-est' osadki"<<endl<<"Siege: ";
cin >> p[b].osad;
} while (p[b].osad>1 || p[b].osad<0);
do{
cout<<"vvedite znachenie v diapazone ot 709 do 800"<<endl<<"Pressure: ";
cin>>p[b].dav;
} while (p[b].dav>800 || p[b].dav<709);
b++;
}
void output(Pogoda p[], int b) //функция вывода
{ if(b==0) { system("cls"); cout << "No Data" << endl;     }
  else { int k; system("cls");
         cout<<"*   Data     *  Day Temp  * Night Temp  * Siege  * Pressure  *"<<endl;
         star( );
         for (int j=0;j<b; j++)
         {
         cout<<"* "<<setw(3)<<p[j].date.day <<"."<<p[j].date.month<<"."<<p[j].date.year <<"
         *"<<setw(8)<<p[j].dt <<"    *"<<setw(6)<<p[j].nt<<"       *"<<setw(6)<<p[j].osad<<"  *
         "<< setw(6)<< p[j].dav <<"    *  "<<endl;
         star( );
         }
       }
system("pause");
}
void leto(Pogoda *p, int b)
{        Pogoda* temp = new Pogoda[b];
         int max;
         if(b==0){ system("cls"); cout << "No Data" << endl; }
         else { for(int i=0; i < b-1; i++)
                  {   max=i;
```

```
                    for(int j=i+1; j < b; j++)
                    {  if(p[j].dt>p[max].dt)        max=j;
                       temp[i]=p[i]; p[i]=p[max]; p[max]=temp[i];
                    }
                }
                  int k;  system("cls");
        cout<<"*   Data     *  Day Temp  * Night Temp  * Siege  * Pressure  *"<<endl;
        star( );
        for (int j=0;j<b; j++)
        { if (p[j].date.month>=5 && p[j].date.month<=8)
        cout<<"* "<<setw(3)<<p[j].date.day <<"."<<p[j].date.month<<"."<<p[j].date.year <<"
         *"<<setw(8)<<p[j].dt <<"    *"<<setw(6)<<p[j].nt<<"       *"<<setw(6)<<p[j].osad<<"  *
         "<< setw(6)<< p[j].dav <<"    *  "<<endl;
        star( );
        }
}
system("pause");
delete [ ] temp;
}
void zimden(Pogoda *p, int b)
{       int max = p[0].dt ;
        int index = 0;
        if(b==0) { system("cls");       cout << "No Data" << endl;  }
           else { int k;            system("cls");
                cout<<"*   Data     *"<<endl;
                for (int j=1;j<b; j++)
                        if (p[j].date.month==12 || p[j].date.month==1 || p[j].date.month==2)
                         if (max<p[j].dt) { max = p[j].dt; index = j;
                  }
        cout<<"* "<<setw(3)<<p[index].date.day <<"."<<p[index].date.month <<"."<<
```

```
p[index].date.year <<"  *"<<endl<<"***************"<<endl;
}
system("pause");
}
void star()         //функция для вывода разделителей строк
{
        for (int i=0;i<62;i++)
        cout<<"*";
        cout<<endl;
}
```

Пример демонстрирует возможность создания меню в консольном приложении, передачу массива структур в качестве параметров функции, создания и удаления динамической памяти для массива структур.

## Объединения

Объединение позволяет запоминать данные различных типов в одном и том же месте памяти. В каждый момент времени объединение может хранить значение только одного типа из набора. Память, которая выделяется переменной типа объединение, определяется размером наиболее длинного из элементов объединения. Все элементы объединения размещаются в одной и той же области памяти с одного и того же адреса. Значение текущего элемента теряется, когда другому элементу объединения присваивается значение.

Синтаксис определения объединения аналогичен определению структуры, с использованием ключевого слова union.

```
union sign { int svar;
             unsigned uvar;
           }number;          //Знаковое или беззнаковое целое

union { char *a, b;    float f[20];
        }  jack;
```

Память, выделенная для хранения переменой jack, равна памяти необходимой массиву из 20 элементов типа float.

Объединения используются для экономии памяти в тех случаях, когда известно, что больше одного значения поля одновременно не требуется. Объединения часто используются в качестве поля структуры, при этом в структуру удобно включить дополнительное поле, определяющее, какой именно элемент объединения используется в каждый момент. Тогда имя объединения можно не указывать, что позволяет обращаться непосредственно к его полям. Например,

```
#include <iostream>
using namespace std;
int main ( ) {
        enum ptype {Card, Check};
        struct {
            ptype type;
            union {
                char card [25];
                long check;
                };
            } info;
// присваивание значения info
switch ( info.type ) {
    case Card: cout<< “оплата по карте”<< info.card; break ;
    case Check: cout<< “оплата чеком”<< info.check; break ; }
}
```

### *Контрольные вопросы*

1. Какого типа константы могут содержать перечисляемый тип?
2. В каком случае имя структурного шаблона не является обязательным?
3. Какая операция используются для доступа к полям структурной переменной?

4. Какого типа могут быть поля структуры?
5. Что такое иерархическая структура?
6. В чем заключается отличие между структурой и объединением?
7. Как инициализировать структурную переменную?
8. Каким образом передается структура в функцию?

# Глава 2. РАБОТА С ФАЙЛАМИ

Во всех примерах, рассмотренных ранее предполагалось, что ввод в программу осуществляется с клавиатуры, а вывод направляется на экран монитора.

В случаях использования большого объема вводимых и выводимых данных используют файлы.

Файлом называется именованная область внешней памяти, в которой содержится некоторая информация. Хранение информации в файле позволяет не вводить заново данные при повторном запуске программы, а также просматривать данные сколько угодно раз. Содержимое файла может быть просто просмотрено на экране или напечатано на принтере.

Язык C++ позволяет работать с файлами несколькими способами.

## Текстовые и бинарные файлы

Первый из предлагаемых способов применялся в языке С и поддерживается в С++. Язык С выделяет 2 вида файлов: текстовые и двоичные (бинарные). Текстовым считается файл, в котором информация запоминается в виде символов кода ASCII (или аналогичном). Он отличается от бинарного файла, который обычно используется для запоминания кодов машинного языка. Таким образом, содержимое текстового файла можно просмотреть и изменить в любом текстовом редакторе. Для чтения бинарного файла необходима специальная программа.

Для работы с любым видом файла описываем указатель на переменную типа file (иногда используют термин файловая переменная):

```
file *in;
```

Открытие файла происходит с помощью функции fopen( ). Данной функцией управляют три основных параметра.

Имя файла, который следует открыть, является первым аргументом.

Второй аргумент, указывающий режим использования файла, может принимать три значения: “r” – файл используется для чтения, “a” – для дополнения, “w” – для записи (при применении “r” используется существующий файл, для двух других, если файл не существует, то он будет создан; если используется “w”

для уже существующего файла, старая версия файла затирается);

Третий параметр является указателем на файл, и это значение возвращается функцией. Например,

```
file *in;
in = fopen( “test”, ”r” );
```

Теперь in является указателем на файл, хранящийся на диске под именем test. С этого момента программа будет работать с файлом через in, а не по имени test. Если fopen( ) не способна открыть требуемый файл, то она возвращает значение ’NULL’ (определенное в файле stdio.h как 0).

Рекомендуется использовать проверять существование файл перед его открытием, например, строкой вида

```
if ( (in = fopen( “test”, ”r” ) ) != NULL )
```

Если файл создать невозможно, то указателю in присваивается значение NULL, и условие становится ложным. В этом случае нужно вывести на экран предупреждающее сообщение и завершить программу.

Закрыть файл проще. Для этого используют функцию fclose( ), которая имеет только один аргумент – указатель на файл.

```
fclose(in) ;
```

Ввод текстового файла осуществляется с помощью функций getc( ) или getch( ), работающих с одним символом. Различие этих функций в том, что первая работает с целым типом, а вторая – с типом char. Функции putc( ) и putch( ), соответственно записывают один символ. Прототипы эти функций содержатся в заголовочном файле stdio.h.

```
//Программа печатает содержимое файла на экран и определяет количество точек
#include < stdio.h >
int main( )
{
file *in;// описание указателя на файл
char ch, point = ‘ . ’;
int count=0;
```

```
if ( ( in = fopen( “test”, “rt” )) != NULL )
        {  while ( (ch = getch(in) ) != EOF)  // получает символ из in
              { putch(ch, stdout);      //посылает на stdout (стандартный вывод - экран)
                if (ch = = point) count++;
              }
        fclose(in);
        }
else printf(“‘файл не открывается /n“);
}
```

При работе с файлом используются понятия начало и конец файла. Когда файл открывается, система помещает файловый указатель на начало файла – его первый элемент. При чтении информации в систему передается первый элемент, и происходит сдвиг указателя на следующий элемент. Таким образом, работа с файлом походит на работу с массивом, размер которого заранее не известен. Указатель перемещается до тех пор, пока не будет достигнут конец файла. Конец файла помечается константой EOF (End of File), которая автоматически формируется при закрытии файла, после записи в него информации. Данная константа возвращается функцией feof( ) в случае достижения конца файла. Ее параметром служит указатель на файл. Следовательно, задание цикла чтения до конца файла может быть записано как:

```
while ( !feof( in) ) { … }
```

Для указания, что файл открыт в двоичном режиме необходимо добавить букву b во второй аргумент функции fopen() (для текстового добавляется t , но это не является обязательным).

Для работы с бинарными файлами существует ряд функций:

fread ( buffer, size, count,stream ) содержит 4 аргумента. Данная функция читает count элементов длины size из входного потока ( файла ) stream ( FILE * stream ) и помещает в заданный массив buffer. При этом значение указателя файла увеличивается на число действительно прочитанных байтов.

fwrite ( ) позволяет записывать в файл и имеет такие же аргументы. Функ-

ция дописывает count записей по size байтов каждый из области buffer в выходной поток stream.

fseek (stream, offset, origin) перемещает внутренний указатель файла, связанный с потоком stream, на новое место в файле, которое вычисляется по смещению offset и указанию направления отсчета origin. После использования fseek( ) следующая операция ввода/вывода с указанным потоком stream будет выполнена, начиная с той позиции, на которую произведено перемещение. Аргумент offset должен быть типа long, а origin может принимать значение одной из следующих целочисленных констант:

SEEK_SET (значение 0) – начало файла,

SEEK_CUR (значение 1) – текущая позиция указателя файла,

SEEK_END (значение 2) – конец файла.

Функция fseek( ) возвращает целое значение.

Для текстового режима работы с файлом можно также использовать fseek(), но значение аргумента offset должно быть получено с помощью функции ftell() или равняться нулю.

long ftell (stream);

file *stream;

Функция позволяет получить текущую позицию указателя файла, связанного с потоком stream. Позиция задается как смещение относительно начала файла.

Запись и чтения файла осуществляется поэлементно. Элементом файла может быть символ, или целое число, или даже структурная переменная.

Если текстовый файл можно создать с помощью текстового редактора, например WordPad или среды Borland C++, то бинарный файл нужно создавать программно. Так для хранения координат точек на плоскости можно создать файл следующим образом:

```
#include<stdio.h>
#include<stdlib.h>
int main ( )
{
```

```
struct point              //шаблон и переменная для хранения координаты
{       int x, y;
        }z;
FILE *f;
f = fopen( “point .bbb”, ”wb”);
randomize( );
int k=1;
while ( !k )     { printf (“Задать координаты точки? Если да - 1, иначе 0 \n”);
                   z. x= random(100) – 50;
                   z. y = random(100) – 50;
                   fwrite (&z, sizeof(point), 1, f);
        }
fclose(f);
}
```

Таким образом, на диске в текущей директории создан файл с именем point.bbb, и теперь обрабатывать хранящиеся в нем данные. Требуется отметить, что в отличие от массива при работе с файлом нужно помнить, что количество элементов файла неизвестно. Например, в заданном файле определим координат точек, принадлежащих прямой, заданной уравнением $y = kx + b$. Значения констант $k, b$ задаются с клавиатуры.

```
#include<stdio.h>
int main ( )
{
struct point
{       int x, y;
        }z;
int k, b, count=0;
printf (“Введите параметры уравнения”);
scanf ( “%d   %d  ”, &k, &b);
FILE *f;
```

```
f = fopen( “point .bbb” ,”rb” );
while ( !feof(f) ) {
                    fread (&z, sizeof(point), 1, f);
                    if ( z.y = = k*z.x + b )   { count++;
                                                 printf (“точка (%d, %d) лежит на прямой, z.x, z.y);
                                                }
                  }
fclose(f);
printf ( “количество найденных точек  %d, count);
}
```

В следующем примере рассмотрена обработка чисел, хранящийся в файле. Первоначальное содержимое файла создается в программе с возможностью 2 режимов; генерации чисел случайным образом или вводом с клавиатуры. Во второй программе происходит считывание чисел из файла и их обработка.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main( ) {
FILE *number = fopen("number.cpp", "wb");
srand(time(NULL));
printf("1 - Avto | 2 - Handle\nChose:");
int chose, count;
scanf("%d", &chose);
switch(chose) { // Выбор способа ввода
case 1:
        count = rand()%100+16; // Автоматический ввод
        for(int i = 0, rnd; i < count; i++){
        rnd = rand()%100;
        fwrite(&rnd, 1, sizeof(int), number);
```

```
        }  break;
case 2:
        printf("Count of number:"); // Ручной ввод
        scanf("%d", &count);
        if(count < 16){
        printf("Count need > 15");
        return 0;
        }
        for(int i = 0, rnd; i < count; i++){
        scanf("%d", &rnd);
        fwrite(&rnd, 1, sizeof(int), number);
        } break;
default:
        printf("Error of number chose!!!");
        return 0;
        }

fclose(number);
return 0;
}

#include <stdio.h>
int main( ) {
FILE *number = fopen("number.cpp", "rb");
fseek(number, -16*sizeof(int), SEEK_END); // Установка указателя на 16 число с конца
int X[4][4], summa = 0;
for ( int i = 0, num; i < 4; i++){ // Заполнение массива 4 на 4 числами
        for(int j = 0; j < 4; j++) {
        fread(&num, sizeof(int), 1, number);
        if (i < j ) summa += X[i][j];
```

```
            X[i][j] = num;
            printf(" %d ", num);
                    }
                    printf("\n");
            }
    printf("\nSumma = %d\n", summa);
    fclose(number);
    return 0;
    }
```

Работа с данными, организованными в файлы, позволяет хранить результаты работы программы после ее завершения, а также не тратить время на ввод данных при каждом запуске программы.

### *Контрольные вопросы*

1. Что такое файл?
2. Как объявить файловую переменную?
3. В чем заключается отличие работы с файлом и массивом данных?
4. Как определяется конец файла?
5. Можно ли редактировать бинарный файл в текстовом редакторе?
6. Перечислите функции чтения и записи в текстовый файл?
7. Каким способом можно обратиться к определенной записи файла, например десятой?

# Глава 3. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

В основе объектно-ориентированного программирования (ООП) лежит идея разбиения задачи на группу объектов. ООП – это парадигма разработки программных систем, в которой приложения состоят из объектов.

Объекты – это сущности, которые имеют характеристики и поведение.

Объекты являются экземплярами какого-нибудь класса. Классом называют тип данных, объединяющий в себе данные различных типов и стандартные подпрограммы(функции), обрабатывающие эти данные. При необходимости можно получить доступ к данным класса, но при этом используются только подпрограмма данного класса.

К достоинствам объектно-ориентированных программ относятся:

хорошая структурированность, что облегчает понимание алгоритма программы;

возможность их разбиения на небольшие компоненты и тестирования отдельных компонент;

простота расширения.

## Принципы объектно-ориентированного программирования

Три ключевых принципа определяют объектно-ориентированное программирование: инкапсуляция, наследование, полиморфизм.

*Инкапсуляция* (сокрытие информации) – определение пользователем новых типов данных. Каждый такой тип содержит определение набора значений и операций, которые могут быть выполнены над этими значениями, и образует так называемый абстрактный тип данных. При этом никакие другие функции, кроме набора определенных в классе операций, не должны иметь доступа к значениям этого типа. В языке C++ инкапсуляция данных реализуется с помощью механизма классов. Переменные класса являются объектами, из которых строится про-

грамма.

*Наследование* – механизм, позволяющий строить иерархию типов. Это предполагает определение базового типа (или прародителя), а затем использование этого типа для построения производных типов (потомков). Причем каждый производный тип наследует все свойства базового типа, включая как данные, так и набор операций (функции). Новые типы данных могут также иметь свои дополнительные свойства. Таким образом, они не дублируют свойства базового класса, а только имеют возможность их использовать.

*Полиморфизм* в переводе с греческого означает «много форм», заключается в обозначении общего действия одним именем (функцией), которое используется во всей иерархии типов. Каждый тип в этой иерархии реализует это действие своим собственным, пригодным для него способом. В языке C++ полиморфизм имеет две формы: перегрузка операций и функций, использование виртуальных функций.

*Декомпозицией* называют разбиение целого на составные элементы. Декомпозиция широко используется в программировании. В объектном подходе рассматривают два вида декомпозиции: *алгоритмическую* и *объектную*.

В соответствии с *алгоритмической декомпозицией* предметной области при анализе задачи разработчик пытается понять:

какие алгоритмы необходимо разработать для ее решения,

каковы спецификации этих алгоритмов (вход, выход),

как эти алгоритмы связаны друг с другом.

В языках программирования данный подход в полной мере поддерживается средствами модульного программирования.

*Объектная декомпозиция* предполагает:

выделение основных содержательных элементов задачи, разбиение их на типы (классы),

определение свойств (данные) и поведения (методы, операции) для каждого класса,

взаимодействия классов друг с другом.

Объектная декомпозиция поддерживаются всеми современными объектно-ориентированными языками программирования

Абстрагирование также применяется при решении многих задач, поскольку любая модель позволяет абстрагироваться от реального объекта, подменяя его изучение исследованием формальной модели.

Абстрагирование в ООП позволяет выделить основные элементы предметной области, обладающие одинаковой структурой и поведением. Такое разбиение предметной области на абстрактные классы позволяет существенно облегчить анализ и проектирование системы.

Согласно этому принципу в модель включаются только те аспекты проектируемой системы, которые имеют непосредственное отношение к выполнению системой своих функций.

## Понятие «класс»

Класс является типом данных, определяемым пользователем. Он представляет собой модель реального объекта в виде данных и функций для работы с ними. Класс есть расширение понятие структуры языка C++. Каждый представитель класса называется объектом.

Объединение данных и функций, которые обрабатывают объект определенного типа, в одном описании называется инкапсуляцией.

Данные класса называются полями или данными-членами класса. Функции класса – методами или функциями-членами класса. Поля и методы называются элементами класса. Описание класса выглядит примерно так:

```
class имя {
 private:
 описание скрытых элементов
 public:
описание доступных элементов
 protected:
описание защищенных элементов
```

```
};
```

Описанию элементов предшествуют ключевые слова, являющиеся спецификаторами доступа: private (закрытый), public(открытый) и protected (защищенный).

Открытые элементы (public) доступны снаружи класса. Они, как правило, отвечают за внешний интерфейс класса и являются методами класса.

Закрытые элементы (private) предназначены только для внутреннего использования в классе. Как правило, они являются полями. Закрытые элементы доступны только методам класса, в состав которого они входят.

Защищенные элементы (protected**)** доступны для методов данного класса и классов, производных от него.

При создании класса программист самостоятельно решает, какие из членов класса будут общедоступными, а какие закрытыми. Однако, большинство классов имеет типичную схему: закрытая часть содержит данные (поля), а открытая – функции для работы с этими данными (методы).

В классе могут присутствовать многочисленные открытые и закрытые секции сколько угодно раз и в производном порядке.

Спецификатор доступа не является обязательным, и, если он не присутствует, по умолчанию элементы класса становятся закрытыми.

Для полей класса справедливы следующие правила:

- они могут быть любого типа, кроме типа того же класса (но могут быть ссылками или указателями на класс, в котором объявлены);
- поля могут быть объявлены с модификаторами const, но при этом они принимают значение только один раз (с помощью конструктора) и не могут изменяться;
- они могут быть описаны с модификаторами static.

Рассмотрим класс, созданный для хранения даты и времени.

```
class TTime {
        private:
        unsigned year,  month, day, hour, minute;
```

```
    public:
    void Dispay( );
    void SetTime(unsigned d, unsigned m, unsigned y, unsigned hr, unsigned min);
};
```

В рассмотренном примере класс TTime имеет семь элементов класса: пять полей – year, month, day, hour, minute и два метода класса - Dispay( ) и SetTime( ).

Методы класса объявлены прототипами функций. Предположительно, они выполняют какие-то действия над полями. Их тела будут описаны позднее. Тогда их описанию должны обязательно предшествовать имя класса и операция разрешения видимости ( :: ), сообщающая о принадлежности метода классу.

```
// Определение методов
void TTime::SetTime(unsigned d, unsigned m, unsigned y, unsigned hr, unsigned min)
{
    day = d;
    month = m;
    year = y;
    minute = min;
    hour = hr;
};
void TTime::Display( ) {
 char s[40];
cout<< "Дата:"<< day<< "."<< month << "." << year<<endl;
cout<< "Время:" << hour << ":" << minute) << endl;
};
```

Тело метода может также содержаться внутри определения класса. В этом случае метод является встроенной (inline) функцией.

Рассмотренный пример показывает применение невстроенных методов.

Можно определить встроенную функцию-элемент и вне тела класса, указав в заголовке определения ключевое слово inline.

Существуют отличия между элементами класса и обычными функциями.

1. При описании имени метода предшествует имя класса и оператор разрешения области видимости **::.** Имя класса однозначно определяет имя метода, поэтому в программе могут быть другие функции с такими же именами, но принадлежащими другим классам.

2. Внутри метода операторы имеют прямой доступ к элементам собственного класса.Передавать их как параметры не следует.

3. Функция main( ) использует класс как и любой другой тип. Для использования метода, требуется создать объект его типа, чтобы использовать метод класса.

Классы, определяемые программистом, похожи на встроенные типы (объект – это переменная типа «класс»). Можно объявлять сколько угодно объектов, причем синтаксис их объявления аналогичен объявлению переменной любого другого типа.

class идентификатор класса идентификатор переменной;

Объекты можно передавать в качестве параметров функций или возвращать их как значение; объявлять массивы, состоящие из объектов и указатели на объект.

Как и любая переменная, объект класса может быть динамическим (т.е. создаваться каждый раз, когда управление достигает его объявления, и уничтожаться, когда управление выходит из данного блока) или статическим (т.е. создаваться один раз и уничтожаться по завершению программы).

С другой стороны, язык C++ обрабатывает классы иначе, чем встроенные типы. Большинство встроенных операций не могут применяться к объектам. Нельзя использовать математические операции (сложение, вычитание, умножение, деление, остаток от деления) для двух объектов, операторы сравнения, инкремента и декремента. Но все эти операции применимы для полей классов.

Для самих объектов справедливы следующие встроенные операции: выбор элемента (.), присваивание (=), получение адреса (&), косвенная адресация (*), и операция ->.

Доступ к элементам объекта аналогичен доступу к полям структуры. Для этого используются операции точка (выбора) при обращении к элементу через имя объекта и операция -> при обращении через указатель. Обращение таким способом возможно только к элементам со спецификатором доступа public. Получить или изменить значения закрытых элементов (private) можно только через обращение к соответствующим методам класса.

Использование методов класса TTime возможно в функции main( ) следующим образом.

```
  int main ( )
{
    TTime day;                  //объявление объекта day класса TTime
    day.SetTime( 24, 2, 1996, 4, 20);       // вызов метода SetTime для объекта day
    day.Display( );                         // вызов метода  Display для объекта day
};
```

Методы класса могут вызывать другие функции-элементы того же класса, используя их имена.

Возможно определение указателя на функцию-элемент класса.

Пример использования указателей при работе с объектами класса.

```
class Coord {
                int x, y;
public:
                void SetCoord(int _x, int _y) {x = _x; y = _y};
                void GetCoord(int & _x, int & _y) {_x = x; _y = y};
};
int main( )
{
        Coord org;                      // локальный объект
        // указатель на класс инициализирован адресом объекта org
        Coord *orgPtr = &org;
        org.x = 0;
```

```
    orgPtr->y = 0;
    org.SetCoord(10, 10);
    int col, row;
    orgPtr->GetCoord(col, row);
    return 0;
};
```

В C++ структура, класс и объединение являются сложными типами данных. Структура и класс схожи друг с другом, но в структуре и объединении элементы имеют по умолчанию доступ public, а в классе private.

## Конструктор

Язык C++ имеет две встроенных особенности при работе с классами – конструкторы и деструкторы – помогающие не забывать про инициализацию объектов и про очистку памяти после завершения работы с ними. Использование конструкторов и деструкторов необходимо во избежание массы досадных ошибок.

Конструктор – функция, автоматически вызываемая при создании объекта (инициализации класса).

Внутрь конструкторов могут быть помещены процедуры инициализации для установки необходимых значений полей до использования объекта. В одном классе можно объявлять сразу несколько конструкторов. Это требуется для инициализации объектов различными способами.

Конструктор является методом класса и носит тоже имя. Он вызывается автоматически компилятором при создании каждого представителя класса.

Если конструктор в программе не определен, то компилятор автоматически генерирует конструктор без параметров, называемый конструктор умолчанию.

Для конструкторов выполняется следующие правила:

конструктор не возвращает значение, поэтому в его описании возвращаемый тип не указывается, причем даже void;

возможно объявление несколько конструкторов с разными наборами параметров для разных видов инициализации;

конструктор, вызываемый без параметров, называется конструктором по умолчанию;

конструктор не наследуется;

конструктор не может быть объявлен как const, volatiler, virtual или static.

Синтаксис объявления конструктора аналогичен синтаксису объявления любого другого метода класса. Например,

```
class XYValue {
        int x, y;
    public:
        XYValue (int X = 100, int Y = 10) { x=X;  y=Y; }   //конструктор
};
```

Примерами вызова конструкторов для объектов данного класса могут служить следующие конструкции:

```
XYValue S( 200, 300), M(50), Z;
```

Здесь создаются три объекта с именами S, M, Z. Круглые скобки после имен объектов указывают на вызов конструктора с параметрами. Значения не указанных параметров устанавливаются по умолчанию. В результате для объекта S значения поля x равно 200, а y равно 100. Для объекта M поле x=50, а y устанавливается по умолчанию равным 10. Для объекта Z компилятор формирует конструктор по умолчанию, и значения полей будут равны нулю.

Таким образом, конструктор вызывается, если в программе встретилась какая-либо из следующих конструкций:

имя класса имя объекта(список параметров);

Список параметров не должен быть пустым.

имя класса (список параметров);

Создается объект без имени (список может быть пустым)

имя класса  имя объекта = выражение;

Создается объект без имени и копируется

Рассмотрим пример класса с несколькими конструкторами.

```
enum color {white, black, auburn, spotted};
```

```
class kitten {
        int age;
        color skin;
        char *name;
        public:
        kitten ( int a=1);        //объявлен конструктор по умолчанию
        kitten ( color sk);
        kitten (char *nam);
        …
        };
        kitten :: kitten ( int  a)                  // описания конструкторов
        { age=a; skin=black; name=0;}

        kitten :: kitten  ( color sk)
{ switch (sk) {
                case black :   age=10; skin=black;  strcpy (name, “черныш” );   break;
                case white :   age=2; skin=white; strcpy (name, “снежок” ); break;
                case auburn : age=3;  skin=auburn; strcpy (name, “рыжик” );  break;
                case spotted : age=5; skin=spotted; strcpy (name, “ ” );  break;
               }
 }
kitten:: kitten ( char *nam)
{ name = new char [strlen(nam) +1];
// к длине строки добавляется 1 для хранения нуль-символа
strcpy (name, nam);          // копирует содержимое одной строки в другую
cout<< “введите возраст:”;
cin>> a;
cout<< “окрас”;
cin>>skin; }
```

При использовании нескольких конструкторов в одном классе необходимо

помнить, что они должны отличаться набором параметров. Каждый из них должен иметь уникальный набор параметров, иными словами, уникальную сигнатуру. Кроме того, если определение конструктора содержит список инициализации элементов, то список может определяться от заголовка определения функции двоеточием. В этом случае поля перечисляются через запятую. Для каждого поля в скобках указывается инициализирующее значение, которое может быть выражением. Без этого способа не обойтись при инициализации полей-констант, полей-ссылок, полей-объектов. Пример использования инициализации полей в конструкторе.

```
kitten :: kitten ( color c) :  age(1), skin(c),  name(0) { };
```

## Деструктор

Деструктор является дополнением конструктора. По завершению работы с объектом автоматически вызывается функция, называемая деструктором.

Он также носит имя класса, но ему предшествует префикс-тильда (~). Деструктор вызывается всякий раз, когда уничтожается представитель класса. Для деструктора выполняются те же правила, что и для конструктора.

Пример простейшего деструктора:

```
#include <iostream>
using namespace std;
class Pair {
private:          int first, second;                  // поля
public:
        Pair ( int one, int two): first(one), second (two)              //конструктор
        { cout<< " объект создан "<< endl; }
         ~Pair ( ) { cout <<" объект удален "<< endl;}                  //деструктор
        void out ( ) { cout<< first<< " << second; }                    //метод
  }
int main( )  {
 Pair num1(2,3); num1. out( );
```

```
Pair num(4,5); num2. out( )
return 0; }
```

Выделение динамической памяти одна из важных функций конструктора. Память, выделенная для динамического объекта в конструкторе, освобождается при удалении объекта – в деструкторе.

Примером деструктора для рассмотренного класса kitten является

```
kitten :: ~ kitten( ) {delete [ ] name;}
```

Вызов деструктора никогда не выполняется явным образом. Он всегда осуществляется компилятором автоматически.

## Указатель this

Иногда внутри метода требуется обратиться к указателю на объект. Это можно сделать через ключевое слово this. У каждого объекта указатель this свой. Он содержит в себе адрес объекта, полем которого он является.

Указатель this называют неявным указателем. Он автоматически создается компилятором. Тогда каждый объект класса имеет свою копию полей класса. Методы же класса существуют только в одном экземпляре.

При вызове метода ему передается неявный аргумент, который обозначает конкретный объект класса. Неявный указатель this можно использовать и явно, как и любой другой указатель, для работы с полями объекта.

```
class Simple {
private:        int a;
public:
                Simple( );
                void Greet( ) { cout<<“Hello!”; }
};
 Simple::Simple( ) {
 this->a = 15;
 Greet( );
 (*this).Greet;                  //оба оператора вызывают функцию Greet
```

```
};
```

Другой классический пример использования указателя this в конструкторах для случая одинаковых имен, используемых в качестве полей класса и для аргументов конструктора. В этом случае при обращении к полям необходимо использовать внутренний указатель.

```
class Example
{
private:   int b;                    // b - поле
public Example   ( int b)            // b - параметр функции-метода класса
{ this->b = b; }                     // имена поля и параметра совпадают
};
```

С помощью this можно также возвращать текущий объект класса как результат метода.

```
#include <iostream>
using namespace std;
class Point
{
private:    int x,  y;
public:
            Point(int x, int y)
            {
              this->x = x;
               this->y = y;
              }
              void ShowCoords()
             {
               cout << "Coords x: " << x << "\t y: " << y << endl;
```

```
        }
        Point Modify (int x, int y)
        // метод возвращает объект собственного класса
        {
          this->x += x;
           this->y += y;
           return *this;
         }
};

int main()
{
    Point p1(20, 50);
    p1.modify(10, 5).Modify(10, 10);
    p1.ShowCoords();          // x: 40  y: 65
    return 0;
}
```

## Конструктор копии

По числу параметров и их назначению конструкторы делятся на четыре группы:

1 конструкторы по умолчанию,

2 конструкторы с параметрами,

3 конструкторы копирования,

4 конструкторы преобразования типов.

Первые две группы рассмотрены ранее. Конструкторы по умолчанию не имеют параметров. В них можно выполнять только самые простые действия, например, задание начальных значений переменных и выделение динамической памяти. Конструкторы с параметрами применяются в случаях, когда необходимо вручную установить начальные параметры полей объекта.

Конструктор копии воспринимает в качестве аргумента только один параметр, тип которого константная ссылка на объект класса (const тип_класса &) или простая ссылка на объект (тип_имени &). Например,

```
class Coord {
                int x,y;
public:
                Coord(const Coord & scr);          // Конструктор копии
 };
Coord::Coord(const Coord &scr) {
        x = scr.x;       y = scr.y;
};
```

Ссылка передается каждый раз, когда новый объект инициализируется значениями уже существующего объекта.

Существует операция присваивания, которая присваивает объекту значение другого объекта. Операция присваивания является функцией-элементом с именем operator =, который воспринимает единственный аргумент типа const тип_класса & или тип_класа &.

```
class Coord {
                int x,y;
      public:
              Coord& operator = (const Coord &scr);
               };
Coord& Coord::operator = (const Coord &scr) {
        x = scr.x;
        y = scr.y;
        return *this;
        };
```

Конструктор преобразования типов – конструктор с параметрами, которому передается только один параметр, не совпадающий с типом класса, к которому принадлежит конструктор.

Важно, что при передаче по ссылке копирования аргументов не происходит, поэтому этот способ гораздо эффективнее и быстрее передачи по значению, особенно при работе с большими структурами или классами.

## Статические элементы класса

*Статическое поле* класса существует в единственном экземпляре для всех объектов. При этом любой объект класса может получать и изменять его значение. Статическое поле класса широко применяется для подсчета числа используемых объектов. Например:

```
class race_cars {
private:  static int count;                    //статическое поле
           int number;
           char name[30];
public:    race_cars( ) { count++; }      // при создании объекта count увеличивается
           ~race_cars( ) {count--; }     //при удалении объекта count уменьшается
          int GetCount( ) { return count; }
};
int rase_cars :: count = 0;
```

Объявление статического поля производится вне класса. Оно аналогично объявлению глобальной переменной. В классе статическая переменная лишь скрывается от воздействия извне. К статическому полю можно обратиться только через экземпляр класса, хотя оно имеет значение даже тогда, когда ни одного объекта не существует.

Существуют также статические методы. Они не являются полноценными методам, так как могут обращаться только к статическим полям класса. Однако их можно вызывать даже тогда, когда не существует ни одного объекта класса.

```
class A {
private: static int count;
public:  A( )  { count ++};
                  static void display_count( ) { cout<<count;}
```

```
};
 int A::count=0;        // объявление статической переменной
int main( )
{
A::display_count( );  //число объектов до их создания
A a1, a2, a3;
A::display_count( );  //число объектов после их создания
return 0;
}
```

## Дружественные функции

Дружественные функции – это функции, которые не являются членами класса, однако имеют доступ к его закрытым членам – переменным и функциям, которые имеют спецификатор private.

Для определения дружественных функций используется ключевое слово friend.

Дружественная функция имеет доступ к закрытым членам класса, как если бы она сама была членом этого класса. Во всех других отношениях дружественная функция является обычной функцией. Ею может быть, как обычная функция, так и метод другого класса. Для объявления дружественной функции используется ключевое слово friend перед прототипом функции, которую необходимо сделать дружественной классу. Дружественная функция может быть объявлена как в секции открытых элементов, так и закрытых. Например,

```
class Anything
{
private:     int value;
public:       Anything() { m_value = 0; }
                 void add(int v) { value += v; }
   // Делаем функцию reset() дружественной классу Anything
 friend void reset(Anything &anything);
```

```
// reset() теперь является другом класса Anything
};

void reset(Anything &anything)       // описание reset( )
{
  anything.value = 0;
// Имея доступ к закрытым элементам  класса Anything, вносим изменения
}

int main()
{
    Anything one;
    one.add(4); // добавляем 4 к value
    reset(one); // сбрасываем value к 0
     return 0;
}
```

Функция может быть дружественной сразу нескольким классам. Дружественной может стать функция, объявленная внутри другого класса. Можно сделать дружественным весь класс.

```
#include <iostream>
#include <string>
using namespace std;
 class Auto;
 class Person
{
private:   string name;
public:    Person( string n)
                {
                   name = n;
```

```
            }
            void drive(Auto &a);
            void setPrice(Auto &a, int price);
};
class Auto
{
friend class Person;
private:
          string name;   // название автомобиля
           int price;  // цена автомобиля
public:
           Auto(string autoName, int autoPrice)
          {
            name = autoName;
             price = autoPrice;
            }
           string getName( ) { return name; }
           int getPrice( ) { return price; }
};

void Person :: drive(Auto &a)
{
  cout << name << " drives " << a.name << endl;
}
void Person :: setPrice(Auto &a, int price)
{
   if (price > 0)         a.price = price;
}

int main()
```

```
{
    Auto tesla("Tesla", 5000);
    Person tom("Tom");
    tom.drive(tesla);
    tom.setPrice(tesla, 8000);
    cout << tesla.getName() << " : " << tesla.getPrice() << endl;
     return 0;
}
```

Поля классов, как правило, являются закрытыми, и доступ к ним можно получить только средствами методов того же класса. Для преодоления этого ограничения создаются, так называемые дружественные функции. Эти функции объявляются в описании класса с помощью ключевого слова friend и получают доступ к полям класса, при этом, не являясь его элементами.

Дружественная функция объявляется внутри класса, к элементам которого ей нужен доступ. Одна и та же функция может быть дружественной сразу нескольким классам. В качестве параметра ей должен передаваться объект или ссылка на объект класса.

```
class A;                                  // предварительное объявление класса
class B {
                char name;
                . . .
public:         friend void display (A x, B y);         //дружественная функция
}
class A {
                . . .
public:         friend void display (A x, B y) {  cout << y. name; . . . }
}
```

Использование дружественных функций лучше избегать, так как это затрудняет отладку и модификацию программы.

Можно разрешить элементам класса полный доступ к элементам другого

класса, объявленным как private или protected, включив в определение класса описание friend. Тогда все методы дружественного класса будут иметь доступ к скрытым полям.

```
class myclass
{ friend class anotherclass;
};
```

В этом случае все методы anotherclass будут иметь доступ ко всем полям класса myclass.

Класс anotherclass будет дружественным по отношению к myclass.

## Наследование

Простое наследование описывает родство между двумя класса. Класс, являющийся прародителем называется базовым классом (родителем). Класс, созданный на основе базового класса, называется производным классом (потомком). Производный класс наследует все элементы базового класса и может обладать новыми элементами, свойственными только ему.

На основе одного базового класса можно создавать многие классы. Производный класс сам может быть базовым. Таким способом создается иерархия классов.

Синтаксис механизма наследования следующий:

class имя производного класса: ключ доступа имя базового класса

{ тело класса};

Разница между использованием различных ключей доступа при объявлении наследования представлена в таблице:

| Ключ доступа | Спецификатор в базовом классе | Ключ доступа в производном классе |
|---|---|---|
| private | private | нет |
| | protected | private |
| | public | private |
| protected | private | нет |

| | | |
|---|---|---|
| | protected<br>public | protected<br>protected |
| public | private<br>protected<br>public | нет<br>protected<br>public |

Таким образом, закрытые элементы базового класса в производном классе недоступны вне зависимости от ключа. Обращение к ним осуществляется только через методы базового класса.

Элементы protected при наследовании с ключом private становятся в производном классе private, а в остальных случаях доступ к ним не изменяется.

Доступ к открытым элементам при наследовании становится соответственным ключу доступа.

Указание ключа доступа необязательно, по умолчанию для классов используется ключ доступа private.

Производный класс наследует из базового класса поля и методы, а также деструктор, но не конструкторы и операции присваивания.

В общем случае производный класс наследует все данные и функции своих предков, например:

```
class Base {
private:
        int count;                          // поле класса
public:
Base( ) { count = 0; }                  // определение конструктора
void SetCount(int n) { count = n; }  // определение метода
int GetCount( ) { return count; }      // определение метода
};
```

Если требуется класс, имеющий все свойства Base, но, дополнительно, обладающий способностью изменения значения поля объекта на заданную величину, можно объявить производный класс вида

```
class TDerived: public Base {
public:
Derived( ):Base( ){  };
void ChangeCount(int n) {SetCount( GetCount( ) + n ) };
};
```

Порядок вызова конструктора в производном классе определяется следующими правилами. Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса по умолчанию (то есть без параметров). Если конструктор базового класса требует указания параметров, он должен быть вызван явным образом в конструкторе производного класса в списке инициализации.

Для иерархии классов, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После выполняются конструкторы элементов классов, которые являются объектами, в порядке их объявления в классе.

В случаях нескольких базовых классов их конструкторы вызываются в порядке объявления.

Вызов функций базового класса предпочтительнее копирования фрагментов кода из базового класса в производный. Кроме сокращения объема программы, этим достигается упрощение ее модификации.

Правила вызова деструкторов в производном классе:

Если в производном классе деструктор не объявлен, он формируется по умолчанию и вызывает деструкторы всех базовых классов.

В отличие от конструкторов, при создании деструктора в производном классе не требуется явного вызова деструкторов базовых классов.

Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор класса, затем – деструкторы элементов класса, а потом деструктор базового класса. Например,

```
#include<iostream>
```

```
#include<conio.h>
#include<math.h>
using namespace std;
/* Класс прямоугольник, содержащий конструктор, деструктор, функцию
нахождения площади */
 class pryamoug      {
  protected:     float x, y;
  public:
                 pryamoug( ) {                                  // конструктор по умолчанию
                 cout<< “Введите ширину и длину прямоугольника”;
                 cin<<x<<y;
                 }
                 pryamoug ( float xx, float yy) { x=xx; y=yy; }    //конструктор с параметрами
                 ~pryamoug( ) { cout<< “треугольник уничтожен”<<endl; }       //деструктор
                 float S( ){ return (x*y); }                                  // метод
};
```

/* производный класс *пирамида*, определяемый основанием и высотой и содержащий функции нахождения площади поверхности и объема пирамиды*/

```
 class piramida : public pryamoug {
  private:           float h, s;
  public:
  //конструктор производного класса с вызовом конструктора базового класса
  piramida( float xx, float yy, float hh ) : pryamoug ( xx, yy )
  { h = hh; s = pryamoug :: S( ); }       // вызов экземпляра метода S базового класса
  float pover( )  {                          // нахождение площади поверхности
   float p, a;
  p=2*( x + y );
   a=sqrt ( h*h + sqrt( ( x / 2 )*( x / 2 )+( y / 2 )*( y / 2 ) ) );
   return ( p*a / 2 );
  }
```

```
 float V( )
  { return ( s*h / 3); }                          //нахождение объема
~piramida( )
{ cout<< “пирамида уничтожена”<<endl; }  //деструктор
};
int main( ) {
float  x1, y1, h1;
cout<<"Введите ширину основания";
cin>>x1;
cout<<"Введите длину основания";
cin>>y1;
cout<<"Введите высоту пирамиды";
cin>>h1;
pryamoug A(x1, y1);
piramida B(x1, y1, h1);
cout<<"Площадь основания пирамиды”<<A.S( )<<endl;
cout<<"Площадь поверхности пирамиды"<<B.pover( ) <<endl;
cout<<"Объем пирамиды",<<B.V()<<endl;
return 0;
}
```

На экране в конце работы программы появится

Пирамида уничтожена

Треугольник уничтожен

При желании можно изменить поведение унаследованного объекта. То есть переопределить метод базового класса в производном классе. Этот механизм называется *замещением.*

Если элемент производного класса имеет то же имя, что и элемент базового класса, для объекта производного класса используется представитель производного класса, а для объекта базового класса – метод базового. Например,

```
#include<iostream>
```

```
#include<conio.h>
using namespace std;
class Base {                              // базовый класс
 protected:     int Price;
public:          void Print Me( );
};
void Base:: Print Me( )   { cout<<" Base"<<Price<<"\n";}

class Derived: public Base            // Derived унаследован от Base
{
public:     void Print Me( );             //Print Me( ) другая ( замещенная )
};
 void Derived:: Print Me( )
{
cout<<"Derived"<<Price<<"\n";
Base::Print Me( );                // вызываем метод базового класса
}
int main( ) {
Base Bclass;
Derived CopyClass;
Bclass. Price=1;
CopyClass. Price=7;
Bclass. Print Me( );
CopyClass. Print Me( );
 cout<<" \Press any key to end ";
while (!kbhit( ));
return 0;
}
```

## Множественное наследование

Множественное наследование позволяет создавать новый класс из нескольких базовых. При этом в объявлении производного класса после его имени следует перечислить имена всех базовых классов с соответствующими ключами доступа через запятую. Например,

```
class D: public A, public B, public C {… };
```

Разрешается использовать спецификаторы доступа в произвольном порядке. Множественное наследование отличается от простого лишь тем, что производный класс наследует все свойства всех перечисленных классов.

Также как и при простом наследовании, во множественном – необходимо инициализировать переменные базовых классов. Для этого в конструкторе производного класса необходимо инициировать конструкторы базовых классов их перечислением после двоеточия через запятую.

```
D( ): A( ), B( ), C( ){. . .};
```

В случае конфликта имен элементов базовых классов следует использовать оператор разрешения видимости. Таким образом, если в базовых классах есть одноименные элементы, при этом может произойти конфликт идентификаторов, который устраняется:

```
class A { public : int print( );
. . .
};
class B { public : int print( );
. . .
};
class C: public A, public B {
. . .
};
int main ( )
{
C object;
object. A::print( );
```

```
object. B::print( );
return 0;
}
```

Использование в этом случае обычного вызова метода класса object. print( ); приведет к ошибке.

```
#include<iostream>
#include<iomanip.h>
#include<conio.h>
using namespace std;
class Person {                          // базовый класс
protected:      char name[20];
                int age;
                char gender;
public: void Get_info( );
                void Print( );
};
class Academics {
protected:      char course_name[20];
                int semester;
public:          int Get_info( );
                void Print( );
};
class Student: public Person, public Academics {
private:         float amount;
public:          int Get_info( );
                 void Print( );
};
void Person :: Get_info( ) {
cout<< "Имя?  ";              cin>>name;
cout<< "Возраст?";            cin>>age;
```

```
cout<< “Пол ( М / Ж) ?;       cin>>gender;
}
void Person :: Print ( ) {
cout<< name<< “/ t ” age<< “/ t ”<<gender << “/ t “;
}
void  Academics :: Get_info( ) {
cout<< “Шифр специальности ( АСОИУ/ ИС / ПМ …)?  ”;
cin>>course_name;
cout<< “Семестр(1 / 2/ 3 … )?”;
cin>>semester;
}
void Academics :: Print ( ) {
cout<< course_name<< “/ t ” << semester << “/ t “;
}
void Student :: Get_info( ) {
Person :: Get_info( );
Academics :: Get_info( );
cout<< “Оплата ( тыс. руб.) ?;      cin>>amount;
}
void Person :: Print ( ) {
Person :: Print ( );
Academics :: Print ( );
 cout<< setw(4)<< amount<<endl;
}
void main( )
{
clrscr( );
const n=10;
Student object[n];
cout << “Введите следующую информацию: ”;
```

```
for ( int k=0; k<n; k++)
{
        object[ k ]. Get_info( );
        object[ k ]. Print( );
}
```

## Перегрузка функций и операций

В языке C++ допускается использование перегруженных функций. Под перегрузкой функций (не обязательно методов класса) понимается создание нескольких прототипов функции, имеющих одинаковое имя. Компилятор различает их по набору параметров.

Перегруженные функции оказываются весьма полезными, когда одну и ту же операцию необходимо выполнить над аргументами различных типов. Компилятор по типу фактических параметров определяет, какой экземпляр функции требуется вызвать. Этот процесс называется разрешением перегрузки. Тип возвращаемого значения в разрешении перегрузки не участвует.

Например, необходимо создать функцию для подсчета суммы элементов массивов различного типа int и float

```
#include <iostream>
using namespace std;
int summa (int iarray[ ] );              // прототип функции для массива типа int
float summa (float farray[ ] );          // прототип функции для массива типа float
int main ( ) {
int iarray[5] = {1, 6, 0, 4, 3 };
float farray[5] = { 6.7, 8.0, 3.9, 7.6, 0.4 };
int isum;
float fsum;
isum=summa(iarray);
cout<<”сумма массива целых чисел равна”<<isum<<endl;
```

```
fsum=summa(farray);
cout<<”сумма массива дробных чисел равна”<<fsum<<endl;
return 0;
}

int summa (int array[ ])
{ int i, s=0;
for (i=0; i<5;i++)
s+=iarray[i];
return s;
}
float summa (float array[ ])
{ int i;
float s=0;
for (i=0; i<5;i++) s+=iarray[i];
return s;
}
```

При использовании перегруженных функций необходимо помнить: если функции отличаются только типом возвращаемого значения, а не типами параметров, такие функции не могут носить одинаковое имя. Не могут перегружаться функции, если их параметром является ссылка.

Разрешается осуществлять не только перегрузку функций, но и операторов. В большинстве языков программирования реализована концепция перегрузки операторов, пусть и в неявном виде. Так, например, оператор суммирования позволяет складывать значения разных типов.

В создаваемом классе можно изменить свойства большинства стандартных операторов, таких как +, -, *, /, заставив их работать не только с данными базовых типов, но и также с объектами. Перегрузка операций реализует принцип полиморфизма.

Перегружать явным образом можно большинство операций, за исключени-

ем: .  ?:  #  ##  sizeof.

Перегрузка осуществляется с помощью методов специального типа (функций-операций). Синтаксис функции-операции:

тип operator операция (список параметров) {тело функции}

Например,

```
class angle {
                int degree, minites, seconds;
public:         angle_value (char *);
                int operator > ( angle &a)          // перегруженный оператор
  { if (3600*degree + 60*minites + seconds>3600*a.degree + 60*a.minites + a.seconds)
                return 1;
   else return 0;
  }
};
```

На перегрузку операторов накладываются следующие ограничения:

- сохраняются количество аргументов, приоритет оператора и порядок группировки его операндов, используемые в стандартных типах данных;
- невозможно изменить синтаксис оператора;
- смысл стандартного оператора применительно к базовым классам переопределять нельзя;
- невозможно создавать новые операторы,
- функции-операции не наследуются;
- функции-операции не могут быть объявлены как static.

```
#include <iostream>
#include <string.h>
#include <conio.h>
using namespace std;
const size = 80;
class Phrase
```

```
{
private:
        char str[size];
public:
        Phrase ( ) { strcpy (str, " "); }
        Phrase (char *s) { strcpy (str, s); }
        void display ( ) { cout<< str<<endl;}
        Phrase operator + = (Phrase a)              // перегрузка оператора + =
        {       if ( strlen(str) + strlen(a. str) < size)       strcat (str, a. str);
                else cout<< "строка слишком длинная";
                return (*this);
        }
};
int main ( )
{
Phrase b;
Phrase c (" тест");
b += c;
b.display( );
Phrase d ( "еще ");
d + = c +=d;
d. display( );
return 0;
}
```

## Виртуальные функции

Полиморфизм – это третий принцип, лежащий в основе создания классов. При полиморфизме родственные объекты (то есть происходящие от одного базового класса) могут вести себя по-разному в зависимости от ситуации, возникающей в момент выполнения программы. Для этого один и тот же метод базового

класса переопределяют в каждом классе потомке. Принцип полиморфизма в языке C++ реализуется перегруженными операциями и виртуальными функциями.

Такие функции должны объявляться в обоих классах с атрибутом virtual, записываемым перед типом функции.

Например,

virtual void Set (int x, int y);

Правила объявления виртуальных функций:

Если в базовом классе метод определен как виртуальный, то метод, определенный в производном классе с тем же именем и набором параметров, автоматически становится виртуальным, а отличающимися параметрами - обычным.

Виртуальные методы наследуются, т.е. переопределять их в производном классе требуется только при необходимости задания различающихся действий.

Если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ к методу базового класса с помощью операции разрешения видимости.

Виртуальный метод не может быть объявлен как static, но может быть дружественным.

По отношению ко всем виртуальным методам компилятор применяет стратегию динамического связывания. Это означает, что на этапе компиляции он не определяет, какой из методов должен быть вызван, а передает ответственность программе, которая принимает решение на этапе выполнения, когда уже известно, каков тип объекта.

Виртуальная функция вызывается только через указатель или ссылку на базовый класс. Определение того, какой экземпляр виртуальной функции вызывается, зависит от класса объекта, адресуемого указателем или ссылкой, и осуществляется во время выполнения.

Например,

```
#include <iostream>
using namespace std;
class Base {
```

```
protected: int x;
public: Base (int y) {x=y;}
                void PrintX( ) { cout<< “x=”<<x;}
                virtual ModifyX( ) {x*=2;}
                };
class Derived : public Base {
        public: Derived ( int f) :  Base( f ) { }
                        virtual ModifyX( ) {x/=2;}
};
int main ( )
{
Base b(10); Derived d(10);
b.PrintX( ); d.PrintX( );
Base *pB;
pB=&b;   pB-> ModifyX( ); pB-> PrintX( );
pB=&d;   pB-> ModifyX( ); pB-> PrintX( );
return 0;
}
```

Программа выведет

x=10 x=10 x=20 x=5

Если в классе вводится описание виртуального метода, он должен быть определен хотя бы как чисто виртуальный.

Виртуальная функция, объявленная в базовом классе иерархии порождения, часто никогда не используется для объектов этого класса, т.е. не имеет определения. Чтобы подчеркнуть это, используется следующая запись:

```
virtual int init(void) = 0;
```

Такие функции называются *чисто виртуальными функциями*.

Класс с одной или с большим количеством чисто виртуальных функций называется *абстрактным классом*. Абстрактный класс может быть использован только как базовый класс для последующих порождений новых классов. Следова-

тельно, нельзя создавать объекты абстрактного класса и нельзя использовать его в качестве типа значения, возвращаемого функцией, и типа параметров функции.

Пример

```
class Shapes {
protected:    int x, y;
public:       virtual void draw ( ) = 0;
              virtual void rotate ( int ) = 0;
};
class Circle : public Shapes {
private:      radius;
public:       circle ( int r, int xx, int yy ) { r=radius; x=xx; y=yy;}
              void draw ( ) { …}
              void rotate ( int u ) { …}
};
```

Теперь невозможно создать объекта класса Shapes, поскольку он содержит чисто виртуальные функции. Объявлять объекты класса Circle возможно. Так как виртуальные функции draw ( ) и rotate ( ) в нем переопределены.

Если базовый класс содержит хотя бы один базовый метод, то рекомендуется снабжать этот класс виртуальным деструктором, даже если он ничего не делает. Это предотвратит некорректное удаление объектов производного класса, адресуемых через указатель на базовый класс.

### *Контрольные вопросы*

1. Перечислите основные принципы объектно-ориентированного программирования.
2. Что такое объект?
3. Как обратиться к элементам класса?
4. Какие операции можно применять к объектам?
5. В чем отличия открытых и закрытых элементов класса?
6. Для чего используют конструкторы и деструкторы?

7. В чем отличия конструктора от обычных методов?

8. Каковы правила объявления конструктора?

9. В чем отличие простого и множественного наследования?

10. Каков порядок вызова конструктора в производном классе?

11. Что такое чисто виртуальная функция? В каких случаях она используется?

12. Как называется класс, содержащий объявление чисто виртуальной функции?

13. Можно ли создать объект абстрактного класса?

14. Какие операции нельзя перегружать?

# Глава 4. СТАНДАРТНАЯ БИБЛИОТЕКА ЯЗЫКА С++

## Потоковые классы языка программирования С ++

Библиотека С++ содержит три класса с помощью которых можно управлять файловым вводом-выводом:

ifstream подключает к программе файл, предназначенный для ввода данных (входной файловый поток)

ofstream подключает к программе файл, предназначенный для вывода данных (выходной файловый поток)

fstream подключает к программе файл, предназначенный как для ввода, так и для вывода

Для подключения данных классов необходимо подключить файл ifstream.h. Для создания объекта класса ifstream и связи с ним файла, находящегося в текущем каталоге, требуется записать следующее:

```
ifstream f (“text.txt”, ios::in);
```

Итак, создан объект с именем ifsin класса ifstream. С ним связан файл text.txt. Если файл, с которым связан объект, находится не в текущем каталоге необходимо полностью указать путь его расположения. Вторым аргументом указывается один из следующих флагов:

| ***Флаг*** | ***Назначение*** |
|---|---|
| ios :: in | Файл открывается для чтения, его содержимое не открывается |
| ios :: out | Файл открывается для записи |
| ios :: ate | После создания объекта маркер текущей позиции устанавливается в конец файла |
| ios :: app | Все выводимые данные добавляются в конец файла |
| ios :: trunc | Если файл существует, его содержимое очищается автоматически |

Для объектов рассмотренных классов определено множество методов. Самыми часто используемыми из них являются:

get (c) – получение символа из потока и запись его в с;

put(c) – выводит символ в поток;

seekg( pos) – устанавливает текущую позицию чтения в значение pos;

tellg( ) – возвращает текущую позицию чтения (записи) в поток;

close( ) – закрытие потока.

Пример

```
#include <fstream. h>
#include <iostream>
using namespace std;
int main ( )
{
 char ch;
ifstream fin (“text.txt”, ios :: in);
if   (!fin) cout<<” Невозможно открыть файл”<<endl;
ofstream fout (“text1.txt”, ios :: out);
if   (!fout ) cout<<” Невозможно открыть файл”<<endl;
while (fout && fin.get(ch) );
fout.put(ch);
fin. close( );
fout. close( );
return 0;}
```

В программе демонстрируется создание потоков ifstream и ofstream для обмена данными между файлами.

## Шаблоны классов

Шаблоны классов поддерживают парадигму обобщенного программирования, т.е. программирования с использованием типов в качестве параметров. Механизм шаблонов в C++ допускает применение абстрактного типа в качестве параметра при определении класса или функции.

В дальнейшем шаблон класса может быть использован для создания классов. Процесс генерации компилятором определения конкретного класса по шаблону класса и аргументам шаблона называется инстанцированием (актуализаци-

ей) шаблона.

Определение шаблона класса имеет следующий синтаксис:

template <параметры шаблона> class имя класса {тело} ;

Параметры шаблона перечисляются через запятую. В их качестве могут использоваться не только типы и переменные, но и шаблоны.

Типы, используемые в шаблонах, могут быть как встроенными, так и определенные пользователем. Внутри шаблона параметр типа может применяться в любом месте, где допустимо в дальнейшем использовать спецификацию типа.

Пример. Для представления точки на плоскости разработан класс Point, в котором координаты задаются двумя числами типа double. А в другом приложении требуется задать точки для целочисленной системы координат (тип int). Поэтому сначала объявлен шаблон класса.

```
template <class T> class Point {
private:        T x, y;
public: Point (T a, T b) : x(a), y(b) {}
void show( ) {cout<< "("<<x<<","<<y<<")"<<endl; }
};
```

Префикс template <class T> означает, что объявлен шаблон класса, в котором T – некоторый абстрактный тип. T – параметр шаблона. Вместо T может использоваться любое имя типа. После своего объявления T используется внутри шаблона, аналогично именам обычных типов.

Вместо template <class T> class Point можно писать конструкции вида template <typename T> class Point, но первый вариант считается более распространенным.

При создании шаблона класса в программе генерация классов не происходит немедленно. Для этого нужно создать экземпляр шаблонного класса, который создается либо объявлением объекта, либо объявлением указателя на актуализированный шаблонный тип с присваиванием ему адреса, возвращаемого операцией new.

```
Point <int> anyPoint(-13, 5);
```

```
Point <double> otherPoint=new Point<double>(-13.56789, 5.65478);
```

Возможно вынесение определений шаблона в отдельный файл, а затем его подключение к программе директивой препроцессора.

Для создания шаблона для массивов из n элементов возможна следующая конструкция.

```
template <class T, int n> class Array {…}
```

Актуализация данного шаблона:

```
Array<Point, 20> array; // массив из 20 элементов
```

В этом случае параметры class T, int n могут рассматриваться как формальные параметры шаблона, на место которых при компиляции встанут конкретные значения.

Методы шаблона автоматически становятся шаблонами функций. Если метод описывается вне шаблона, его заголовок должен иметь следующие элементы:

template <описание параметров шаблона>

тип имя класса <параметры шаблона>:: имя функции (параметры)

Правила описания шаблонов:

Локальные классы не могут содержать шаблоны в качестве своих элементов;

шаблоны методов не могут быть виртуальными;

шаблоны классов не могут содержать статические элементы, дружественные функции и классы;

шаблоны могут быть производными как от шаблонов, так и от обычных классов, а также, в свою очередь являться базовыми для шаблонов и обычных классов;

внутри шаблонов нельзя объявлять дружественные шаблоны.

Шаблоны являются мощным и эффективным средством обращения с различными типами данных. К недостатком использования шаблонов можно отнести то, что программа с шаблонами должна содержать полный код для каждого порождаемого типа, что значительно увеличивает размер исполняемого файла.

Стандартная библиотека C++ содержит достаточно большой набор шабло-

нов.

## Контейнерные классы

Контейнерные классы – это классы, предназначенные для хранения данных, организованных определенным образом. К ним относятся массивы, линейные списки, стеки и другие. Для каждого типа контейнера определены методы работы с его элементами, независимо типа элементов данных, хранимых в контейнере. Поэтому один и тот же вид контейнера можно использовать для хранения данных различных типов. Эта возможность реализована стандартной библиотекой шаблонов (STL – Standard Template Library) языка C++.

Использование контейнеров позволяет повысить надежность, универсальность, переносимость программ, уменьшить время их разработки, но за это приходится расплачиваться снижением их быстродействия.

Все контейнеры делятся на два класса: последовательные и ассоциативные. Последовательные обеспечивают хранение конечного количества однотипных величин в виде непрерывной последовательности. К ним относятся векторы (vector), двусторонние очереди (deque), списки (list), стеки (stack) и очереди с приоритетами (priority_queue). Шаблоны указанных контейнеров хранятся в одноименных библиотеках языка C++.

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Они построены на основе сбалансированных деревьев. Это словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset), битовые множества (bitset).

На основе контейнеров стандартной библиотеки можно создавать собственные контейнерные классы.

Контейнерные классы обеспечивают стандартизированный интерфейс их применения. Смысл одноименных операций для различных контейнеров одинаков. Стандарт определяет только интерфейс контейнеров, поэтому их реализации могут отличаться.

Практически в любом контейнерном классе определены следующие поля

указанных типов:

| | |
|---|---|
| value_type | тип элемента контейнера |
| size_type | тип индексов, счетчиков элементов и т.д. |
| iterator | итератор |
| const_iterator | константный итератор |
| reverse_iterator | обратный итератор |
| const_reverse_iterator | константный обратный итератор |
| reference | ссылка на элемент |
| const_reference | константная ссылка на элемент |
| key_type | тип ключа (для ассоциативных контейнеров) |
| key_compare | тип критерия сравнения (для ассоциативных контейнеров) |

Термин «итератор» является аналогом указателя на элемент. Он может применяться для прохода по элементам контейнера в прямом и обратном направлениях. Когда значения элементов контейнера не предполагается изменять применяют константные итераторы.

В помощь для работы с итераторами определено несколько методов, представленных в таблице 2.

В каждом контейнере эти типы и методы определяются способом, зависящим от их реализации.

Таблица 2

| | |
|---|---|
| iterator begin( )<br>const_iterator begin( ) const | указывают на первый элемент |
| iterator end( )<br>const_iterator end( ) const | указывают на элемент за последним |
| reverse_iterator rbegin( )<br>const_reverse_iterator rbegin() const | указывают на первый элемент в обратной последовательности |
| reverse_iterator rend( )<br>const_reverse_iterator rend( ) const | указывают на элемент, следующий за последним в обратной последовательности |

Во всех контейнерах определены методы, позволяющие получить сведения о размере контейнеров:

size( ) – число элементов,

max_size( ) – максимальный размер контейнера,

empty( ) – булевская функция, отвечающая на вопрос: пуст ли контейнер.

## Последовательные контейнеры

Вектором называют структуру, которая позволяет обеспечить эффективный произвольный доступ к своим элементам, добавление и удаление из конца структуры.

Двусторонняя очередь в дополнение к вектору разрешает удаление и добавление с обеих ее сторон.

Список эффективно реализует вставку и удаление элементов в произвольное место своей структуры, но не обеспечивает доступа к этим элементам. Операции последовательных контейнеров представлены в таблице 3.

Обращение к встроенным методам выполняется аналогично обращению к обычным методам класса – операцией доступа (точкой).

Таблица 3

| Операция | Метод | vector | deque | list |
|---|---|---|---|---|
| Вставка в начало | push_front | - | + | + |
| Удаление из начала | pop_front | - | + | + |
| Вставка в конец | push_back | + | + | + |
| Удаление из конца | pop_back | + | + | + |
| Вставка в произвольное место | insert | + | + | + |
| Удаление из произвольного места | erase | + | + | + |
| Произвольный доступ к элементу | [ ], at | + | + | - |

Пример. Программа считывает и выводит на экран ряд целых чисел, хранящийся в файле.

```
#include <fstream>
```

```
#include <vector>
using namespace std;
int main( ) {
ifstream in (“primer.txt”);
vector <int> v;
int x;
while (in.eof( ) )
in>>x;
v.push_back(x);
for (vector<int>::iterator i=v.begin(); i!=v.end(); ++i)  cout<<*i<< “ “;
}
```

В примере для создания вектора используется конструктор по умолчанию. Можно пользоваться и другими конструкторами:

```
explicit vector(); // конструктор по умолчанию
explicit vector(size_type n, const T& value=T()); //создается вектор длины n и заполняется одинаковыми элементами – копиями значения value
vector (const vector<T>& x); //конструктор копирования
```

Ключевое слово explicit используется, чтобы при создании объекта исключить выполнение неявного преобразования при присваивании значения другого типа. Например,

```
vector <int> v2(10, 1); // вектор из 10 элементов равных единице
vector <int> v4 (v2); // создается вектор v4 равный v2
```

Другой пример обработки векторов.

```
#include <fstream>
#include <vector>
using namespase std;
int main() {
double arr[]= {1.1, 2.2, 3.3, 4.4 };
int n=sizeof(arr)/sizeof(doule);
vector<double> v1(arr, arr+n);
```

```
vector<double> v2;    //пустой вектор
v1.swap(v2);  //обменять содержимым v1 и v2
while (!v2.empty() )
{
cout<<v2.back()<< ' '; //вывод последнего элемента
v2.pop_back();         //удаление элемента
}
return 0;
}
```

## Двусторонние очереди и списки

Двусторонняя очередь (deque) является последовательным контейнером, который наряду с вектором, поддерживает произвольный доступ к элементам и обеспечивает вставку и удаление из обоих концов очереди. Указанные операции выполняются за время, пропорциональное количеству перемещаемых элементов, что обеспечивается эффективным способом построения очереди.

Для обеспечения произвольного доступа к элементам очередь разбивается на блоки, доступ к каждому из которых осуществляется через указатель.

Для инициализации двусторонней очереди используются конструкторы, аналогичные конструкторам вектора.

```
deque <int>  D1 (20, 1);
//создается очередь из 20элементов, каждый из которых равеных единице
deque <int>  D2 (D1);                    //вызывается конструктор копии
```

В шаблоне deque определены: операция присваивания, функция копирования, итераторы, операция сравнения, операции и функции доступа к элементам и изменения объектов.

Дополнительно определены функции добавления и выбораэлемента из начала очереди:

```
void push_front( )(const T& value);
void pop_front( );
```

При выборке элемент удаляется из очереди.

Для очереди не доступны функции capacity и reserve, но можно применять функции resize и size.

В списке не поддерживается произвольного доступа к своим элементам, но выполняются их вставка и удаление. Каждый узел списка содержит ссылки на последующий и предыдущий его элементы. Контейнер список поддерживает конструкторы, операцию присваивания, методы копирования, операции сравнения и итераторы

Доступ к элементам списков ограничивается методами: referencefront(); const_reference front() const; reference back(); const_reference back() const;

Для списков определено несколько собственных методов. Первый из них, splice – соединение списков, служит для перемещения элементов из одного списка в другой без перераспределения памяти, только за счет указателей:

```
void splice(iterator position, list<T>& x);
void splice(iterator position, list<T>& x, iterator i);
void splice(iterator position, list<T>& x, iterator first, iterator last);
```

Оба списка должны иметь элементы одного типа. В первой форме метода вставка в вызывающий список осуществляется перед элементом, позиция которого указана первым параметром, всех элементов списка, указанного в качестве вторым параметра. Элементы второго списка при этом обнуляются.

Не разрешается дублировать список, вставкой в себя.

Вторая форма переносит элемент, заданный третьим параметром, из списка x в вызывающий список.

Третья форма функции переносит из списка в список несколько элементов, диапазон которых задается третьим и четвертым параметрами.

```
#include <list>
using namespase std;
int main() {
list<int> SPISOK;
list<int>::iterator i, j, k;
```

```
for(int i=0; i<5; i++) SPISOK.push_back(i+1);
for(int i=12; i<14; i++) SPISOK.push_back(i);
cout<< «Исходный список»;
for(i=SPISOK.begin( ); i!=SPISOK.end( ); ++i) cout<<*i<< “  “;
cout<<endl;
i=SPISOK.begin( ); i++;
k=SPISOK.end( );
j=--k; k++; j--;
SPISOK.splice(i, L1, j, k);
cout<< “Список после сцепки”;
for(i=L1.begin(); i!=L1.end(); ++i) cout<<*i<< “  “;
}
```

К остальным специфическим методам контейнера список относятся:

remove ( ) – удаление элемента по его значению:

void remove(const T& value);

sort( ) – упорядочивание элементов списка по возрастанию:

void sort();

unique( ) – сохранение в списке только первого элемента из каждой серии идущих подряд одинаковых элементов.

merge( ) – слияние списка в упорядоченном виде:

void merge(list<T>& x);

## Контрольные вопросы

1. Соотнесите названия библиотек с их содержимым:

а) ifstream 1) класс входных файловых потоков

б) ofstream 2) класс двунаправленных файловых потоков

в) fstream 3) класс выходных файловых потоков

2. Какие контейнеры обеспечивают быстрый доступ к данным по ключу, поскольку они построены на основе сбалансированных деревьев.

3. Соотнесите название метода и действие, им реализуемое

| | |
|---|---|
| а) вставка в начало | 1) pop_back |
| б) удаление из начала | 2) push_back |
| в) вставка в конец | 3) pop_front |
| г) удаление из конца | 4) push_front |

4. Какое ключевое слово используется при объявлении шаблонов?
5. Перечислите методы, доступные в списках.
6. Определите результат выполнения программы:

```
#include <fstream>
#include <vector>
using namespase std;
int main() {
float  arr[]= {1.8, 2.38, -37, 0.88, -19.4 };
int n;
n=sizeof(arr)/sizeof(float);
vector<float > v1(arr, arr+n);
vector< float > v2;
v1.swap(v2);
while (!v2.empty() ) {
cout<<v2.back()/10<< ‘ ‘;
v2.pop_back();
}
return 0;}
```

*Библиографический список*

1. Бабэ Б. Просто и ясно о C++. М.: БИНОМ, 1995. – 400 с.

2. Галаган Т.А., Соловцова Л.А. Язык программирования С++ в примерах и задачах. Практикум. Благовещенск: АмГУ. 2005. - 102 с.

3. Галаган, Т.А. Алгоритмические языки и программирование. Язык С++. Курс лекций (Рек. ДВРУМЦ) / Т.А. Галаган – Благовещенск: изд-во АмГУ, 2007. – 147 с.

4. Дейл Н., Уимз Ч., Хедингтон М. Программирование на С++. М.: ДМК. 2000. – 672 с.

5. Павловская Т.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2004. – 461 с.

6. Павловская Т.А., Щупак Ю.А. С/С++. Структурное программирование. Практикум. СПб.: Питер, 2004. – 239 с.

7. Павловская Т.А., Щупак Ю.А. С++. Объектно-ориентированное программирование. Практикум. СПб.: Питер, 2011. – 265 с.

8. Паппас К., Мюррей У. Программирование на С и С++. Киев: Издательская группа BHV. 2000. – 320 с.

9. Подбельский В.В. Язык Си++: Учебное пособие. – М.: Финансы и статистика. 2003. – 560 с.

10. Сайт о программирование https://metanit.com/cpp/

**Содержание**

Татьяна Алексеевна Галаган,
*доцент кафедры ИиУС АмГУ*

**Программирование на языке С++. Часть 2**

---

Изд-во АмГУ. Подписано к печати Формат 60х84/16. Усл. печ. л.
Тираж Заказ