

ГЛАВА 11. ФУНКЦИИ В ЯЗЫКЕ C++

Функция – это логически завершенный и определенным образом оформленный фрагмент программы, который может быть вызван из других частей программы. Функции подразделяют большие вычислительные задачи на более мелкие и дают возможность пользоваться уже написанными подпрограммами, в том числе и другими разработчиками.

Если функция написана грамотно и корректно, внутри ее тела оказываются скрытыми несущественные для других частей программы детали ее реализации, что делает программу в целом более ясной, а также облегчает ее отладку и внесение изменений. Функции считаются глобальными для всей программы.

11.1. Объявление и определение функций

Для того чтобы иметь возможность пользоваться функцией, ее необходимо описать до первого обращения к ней.

Для *объявления* функции используют ее описание, называемое *прототипом* и содержащее информацию о ее параметрах:

тип имя_функции (инф. пар1, инф. пар2,...),

где информация о параметре **инф. пар.** имеет один из форматов:

тип

или

тип имя_параметра,

т. е. параметры функций могут быть объявлены с использованием как именных, так и безымянных параметров:

```
void My_Func (int Par1, double Par2, double Par3);
void My_Func (int, double, double);
```

Приведенные формы записи эквивалентны, так как компилятор игнорирует имена параметров, обращая внимание только на их тип. Это связано с тем, что описание функции с помощью прототипа дает возможность компилятору выполнять проверку на соответствие количества и типов параметров при каждом обращении к функции, а также по возможности проводить необходимые преобразования. Прототип функции всегда заканчивается символом «точка с запятой».

Информация о функции, содержащаяся в ее прототипе, т. е. число и типы аргументов, называется *сигнатурой* функции.

Каждая функция должна быть определена где-нибудь в программе после ее объявления, но не внутри другой функции, в том числе и не внутри main().

Для *определения* функции необходимо сначала повторить ее заголовок, который должен полностью совпадать с заголовком, объявленным в прототипе, но без завершающей точки с запятой и с указанием имен параметров, а затем расположить тело функции, заключенное в фигурные скобки.

Формат определения функции имеет вид:

тип имя_функции (тип имя_параметра, ..., тип имя_параметра)

```
{
// тело_функции
}
```

Пример 11.1

```
//include "stdafx.h"
#include <iostream>
#include <iomanip>
using namespace std;
void CntUp(void); // объявление, или заголовок, или
void CntDown(void); // прототип функции.
int _tmain(int argc, _TCHAR* argv[])
{
    CntUp(); // использование функций
    CntDown();
    return 0; // в программе (т.е. их вызов)
}

void CntUp(void) // определение функции CntUp()
{
    int i;
    cout << "From 1 to 10: " << endl;
    for (i=1; i<=10; i++) cout << setw(4) << i;
    cout << endl;
}

void CntDown(void) //определение функции CntDown()
{
    int i;
    cout << "From 10 to 1: " << endl;
    for (i=10; i>=1; i--) cout << setw(4) << i;
    cout << endl;
```

Ключевое слово void, стоящее перед именем функции в ее прототипе в заголовке, сообщает программе о том, что функция выполняет некоторые действия, но не имеет возвращаемых значений, т. е. не передает в вызывающую программу никакого конкретного результата. Слово void внутри круглых скобок, где обычно находится список параметров, сообщает компилятору, что функция не требует передачи аргументов. Длянова такой функции достаточно просто написать ее имя в нужном месте программы и оставить скобки, в которых должны стоять параметры, пустыми.

Такие функции встречаются не так уж редко, но все же не они являются определяющими в языках программирования. Чаще функции организованы таким образом, что для их работы вызывающей программе

приходится сообщать им некоторую информацию путем передачи параметров. В свою очередь, функции могут сообщать программе результаты своей работы посредством передачи возвращаемого значения.

11.2. Понятие о параметрах функций

Формальными называются параметры функции, находящиеся в скобках, при объявлении прототипа функции и при ее определении, а *фактическими* – параметры, подставляемые на место формальных при вызове функции. Если для работы функции требуется более одного параметра, то в скобках задается список параметров, разделенных запятыми.

Пример 11.2. Передача параметров в функцию и получение возвращаемого значения.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int max(int a, int b); // прототип функции поиска
// максимального из двух параметров
int _tmain(int argc, _TCHAR* argv[])
{
    int m, k, l;
    cout << "Input m and k -> ";
    cin >> m >> k;
    l = max(m, k);
    cout << "max = " << l << endl;
    return 0;
}
int max(int a, int b)
{
    if (a > b) return a;
    else return b;
}
```

Здесь формальным параметрам *a* и *b* соответствуют фактические параметры *m* и *k*.

При выходе из функции используют оператор *return*. При этом если функция не имеет возвращаемого значения, то сразу после слова *return* стоит точка с запятой. Если функция имеет возвращаемое значение, то после слова *return* стоит это возвращаемое значение.

Все функции, которые возвращают значения, должны содержать, по крайней мере, один оператор *return*, за которым следует возвращаемое значение объявленного типа.

В общем случае при выходе из функции типа *void* оператор *return* может отсутствовать, но иногда возникает ситуация, когда использование этого оператора необходимо.

Пример 11.3

```
void Func_with_if(параметры)
{
    if (условие) // при истинном значении условия происходит
    {
        return; // немедленное завершение функции,
    }
    блок операторов; // при ложном – после выполнения блока операторов
    return;
}
```

Применение оператора *return* без указания значения вызывает завершение функции и передачу управления в вызывающую программу. Но так как у данного оператора отсутствует возвращаемое значение, в вызывающую программу никакого значения не передается.

Если алгоритмически функция не должна иметь возвращаемого значения, но содержит возможность выхода по условию, то, введя возвращаемое значение, можно сделать функцию более информативной.

Пример 11.4

```
int Func_with_if(параметры)
{
    if (условие) // При истинном значении условия происходит
    {
        return 0; // возвращение значения 0,
    }
    блок операторов; // при ложном – возвращение значения 1.
    return 1;
}
```

Если условие оператора *if* истинно, первый оператор *return* передает 0 в качестве значения функции, а последующий блок операторов и второй оператор *return* не выполняются. Если условие ложно, выполняется блок операторов и второй оператор *return* в качестве значения функции возвращает 1. Вызывающая программа может затем анализировать возвращенное значение и в зависимости от полученного результата выполнять различные действия.

Пример 11.5. Написать функцию, которая по заданному признаку обеспечивает поиск либо большего, либо меньшего из двух заданных значений.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int MaxMin(int p, int x, int y); // прототип функции
int _tmain(int argc, _TCHAR* argv[])
{
    int sign, s1, s2, res;
    cout << "Input sign (0 - found min, 1(not 0) - max) ";
    cin >> sign; // ввод признака
    cout << "Input values ";
    cin >> s1 >> s2; // ввод значений для сравнения
    res = MaxMin(sign,s1,s2); // вызов функции
    cout << "res = " << res;
    return 0;
}

int MaxMin(int p, int x, int y) // определение функции
```

```

int m_m;
if (p==0)           //ищем минимальный элемент
{ if (x < y) m_m = x;
  else m_m = y; }
else { if (x > y) m_m = x;
       else m_m = y; }
return m_m;
}

```

Очевидно, что функции могут возвращать не только целые, но и вещественные значения, а также значения любого базового типа C++.

Пример 11.6

```

double Cube (double r); //объявление прототипа функции Cube
int main ()
{ double x,y=3.14159;
  x=Cube(y);           //возвращаемое значение функции
                        //присваивается переменной x.
  return 0;
}
double Cube (double r) //аргумент у передается по значению параметру r
{   return r*r*r;      //возвращаемое значение функции параметр r
}                      //равен значению переданного аргумента

```

Лучше всего, если типы аргументов (фактических параметров) соответствуют типам объявленных в функции формальных параметров. Значения совместимых типов (например, при передаче аргумента типа int параметру типа float) расширяются автоматически, но может произойти искажение информации из-за ошибок округления. При передаче аргумента типа float параметру типа int выдается предупреждение (Warning), но преобразование допустимо, хотя и с явной потерей информации. В случае несовместимых типов компилятор фиксирует ошибку «Type mismatch...».

11.3. Локальные и глобальные переменные

Любые переменные, объявленные внутри функции (в том числе и внутри функции main), называются *локальными* и существуют в памяти ЭВМ, только пока функция активна. Переменные, объявленные вне функций (в том числе и вне функции main), называются *глобальными*. Локальные переменные также называются *автоматическими переменными*, так как они создаются и разрушаются программами автоматически.

Работа программы при наличии в ней функций происходит следующим образом. Операторы программы выполняются последовательно до тех пор, пока не встретится оператор вызова функции. При каждом вызове функции текущий адрес программы проталкивается в стек и служит адресом возврата из функции после завершения ее работы.

Стеком называется область памяти ЭВМ, организованная таким образом, что позволяет добавлять и удалять элементы данных, но при этом доступен только последний добавленный элемент.

Локальные переменные функции запоминаются в стеке. Параметры функций также хранятся в стеке и, по сути дела, считаются локальными переменными.

Стек динамически изменяется по мере того, как происходит вызов функций и возврат из них.

В начале работы функция выделяет память в стеке для запоминания своих локальных переменных. Эта память существует только до тех пор, пока функция активна. После возврата из функции стековая память удаляется, уничтожая все хранящиеся в ней переменные. Этим обеспечиваются:

- большее по сравнению с доступной памятью суммарное пространство, занимаемое всеми локальными переменными;
- бесконфликтное объявление одинаковых идентификаторов для локальных переменных, используемых в различных функциях одной программы.

Глобальные переменные существуют в течение всего времени работы программы. Они запоминаются в сегменте данных программы и занимают память независимо от того, используются они или нет.

Локальным переменным нужно присваивать начальные значения до их использования, так как в момент объявления значения этих переменных не определены. Глобальные переменные автоматически инициализируются нулевыми значениями при их объявлении.

В Borland C++ существуют *регистровые переменные*, которые запоминаются непосредственно в регистрах процессора. Для объявления регистровых переменных используют спецификатор памяти register. Регистровые переменные бывают только локальными. Они ускоряют работу программы, поэтому чаще всего их применяют для объявления управляющих переменных циклов. Объявление регистровой переменной имеет вид:

```
register int Var_Reg;
```

11.4. Отсутствие прототипов функций

Программа, содержащая функции, может не содержать их прототипов. При этом необходимо, чтобы ее определение находилось до первого обращения к ней.

Пример 11.7. Даны отрезки a, b, c, d. Для каждой тройки этих отрезков, из которых можно построить треугольник, определить площадь этого треугольника.

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

```
#include <math.h>
#include <conio.h>
using namespace std;
double square(double x, double y, double z)
{ double p;
  if ((x+y>z) && (y+z>x) && (x+z>y))
    { p = (x+y+z)/2;
      return sqrt(p*(p-x)*(p-y)*(p-z));
    }
  else
    {cout << "Нельзя построить треугольник с такими сторонами" << endl;
     return 0;
    }
}
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian");
double a, b, c, d;
cout << "Введите значения сторон ";
cin >> a >> b >> c >> d;
cout << "a=" << a << "b=" << b << "c=" << c
     << "S=" << square(a,b,c) << endl;
cout << "a=" << a << "b=" << b << "d=" << d
     << "S=" << square(a,b,d) << endl;
cout << "a=" << a << "c=" << c << "d=" << d
     << "S=" << square(a,c,d) << endl;
cout << "b=" << b << "c=" << c << "d=" << d
     << "S=" << square(b,c,d) << endl;
_getch();
return 0;
}
```

11.5. Строки, массивы и структуры в качестве параметров функций

Рассмотрим передачу строк и массивов в качестве параметров функций. Можно передавать строки в качестве параметров функций, а функции могут возвращать строки в программу:

```
void Print_String(char string[])
{ cout << string;
}
void main()
{
  char MyStr[] = "C++ – язык для профессионалов";
  Print_String(MyStr);
}
```

В функцию передается не сама строка, а только адрес ее первого элемента, связанный с именем строки. Другими словами, символы строки, являющейся аргументом функции (в данном случае MyStr для функции Print_String), не передаются через стек.

Пример 11.8

```
const int MAXLEN = 128;
void A_Func(char s[],int len); // объявление прототипа функции,
                                // обрабатывающей строку длины len
...
char author[MAXLEN] = "Tom Swan";
A_Func(author, MAXLEN);
...
```

Здесь A_Func – любая функция обработки строк, при этом строки рассматриваются как элементы символьного массива. При передаче символьных строк функциям следует помнить о том, что в случае объявления

`const int MAXLEN = 128;`
могут запоминаться только (MAXLEN-1) символов и завершающий нульевой байт.

Borland C++ передает функциям адреса строк и массивов, что существенно экономит стековое пространство. Вследствие такой передачи (по ссылке) вызываемая функция может изменять значения элементов массива непосредственно в тех ячейках памяти, где он располагается. Если нужно запретить изменять значения элементов массива, следует объявить его как массив констант.

В языке C++ допустима передача любых массивов функциям.

Пример 11.9.

Пусть требуется найти сумму элементов массива. Объявим:

```
const int M = 100;
int data[M];
```

Теперь можно объявить функцию, принимающую в качестве аргумента массив:

```
int Summa(int mas[M]);
```

Но лучше поступить следующим образом:

```
int Summa(int mas[], int n);
```

Тогда вызов функции будет

```
cout << "Summa= " << Summa(data, M) << endl;
```

Сама функция суммирования может при этом выглядеть следующим образом:

```
int Summa(int mas[], int n)
```

```
{
    int sum = 0;
    for (int i=0; i<n; i++)
        sum += mas[i];
    return sum;
}
```

Другой пример реализации функции суммирования:

```
int Summa(int mas[], int n)
{
    int sum = 0;
    while (n>0)
        sum += mas[- - n];
    return sum;
}
```

Кроме одномерных массивов, как числовых, так и символьных, функциям можно передавать многомерные массивы. При передаче многомерных массивов в качестве параметров функции следует задавать минимум информации, необходимой компилятору для получения передаваемых адресов памяти.

Можно объявить функцию

```
void SomeFunc ( int arr[rows][cols]);
```

или

```
void SomeFunc (int arr[][cols]);
```

причем второй вариант более предпочтителен.

Но объявление

```
void SomeFunc (int arr [rows][]);
```

скомпилировано не будет, как и объявление

```
void SomeFunc(int arr[][]);
```

Пример 11.10. В данной целочисленной матрице 4×5 вычислить сумму элементов, меньших чем введенное с клавиатуры число.

```
#include "stdafx.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <iomanip>
#include <iomanip>
using namespace std;
const int row = 4;
const int col = 5;
//Объявляем прототипы функций
void form (int arr[][col], int r, int c);
void print_arr (int arr[][col], int r, int c);
int sum_less (int arr[][col], int r, int c, int dig);
int _tmain(int argc, _TCHAR* argv[])
```

```
{
    int matr[row][col];
    int a, summa;
    form (matr, row, col);
    print_arr (matr, row, col);
    cout << "Input value ";
    cin >> a;
    summa = sum_less (matr, row, col, a);
    cout << "Summa = " << summa;
    return 0;
}
// формирование матрицы
void form (int arr[][col], int r, int c)
{
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            arr[i][j] = rand()%100 - 50;
}
// вывод матрицы на экран
void print_arr (int arr[][col], int r, int c)
{
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            cout << setw(4) << arr[i][j];
    cout << endl;
}
//вычисление суммы
int sum_less (int arr[][col], int r, int c, int dig)
{
    int s_=0;
    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            if (arr[i][j] < dig) s_ += arr[i][j];
    return s_;
}
```

Структуры могут передаваться как параметры функций, а функции могут возвращать структуры в качестве своих результатов.

Пример 11.11. Объявим структуру, определяющую окружность путем указания координат ее центра и радиуса:

```
typedef struct circle { int coord_x;
                        int coord_y;
                        int radius;
} Circle;
```

Можно написать функцию, которая возвращает инициализированную структуру типа Circle:

```
Circle C;
= Def_Circle(25,43,15);
//этот Def_Circle() может иметь вид:
Circle Def_Circle(int x, int y, int r)
```

```
Circle rtemp;
rtemp.coord_x=x;
rtemp.coord_y=y;
rtemp.radius=r;
return rtemp;
}
```

Несмотря на то что переменная `rtemp` является локальной для функции и не существует вне области ее действия, функция может возвращать значение переменной `rtemp` в качестве своего результата. При выполнении оператора `return` байты переменной `rtemp` копируются временно в стековую память, а оттуда – в переменную, которой присваивается результат функции. После этого временно записанные в стек байты удаляются.

Необходимо помнить о том, что при передаче структур функциям и из них структуры могут занимать большие объемы стекового пространства, а так как размеры стека ограничены одним сегментом, передача слишком больших структур через стек может вызвать его переполнение.

11.6. Рекурсия

Рекурсия означает возврат. *Рекурсивной* называется функция, вызывающая сама себя. Рекурсивные функции, которые прямо вызывают сами себя, называются *включительно рекурсивными*. Если две функции рекурсивно вызывают друг друга, они называются *взаимно рекурсивными*.

Решение рекурсивной задачи обычно разбивается на несколько этапов. Одним из этапов является решение базовой задачи, т. е. простейшей задачи, для которой написана вызываемая функция. Если задача оказывается более сложной, чем базовая, она делится на две подзадачи: выделение базовой задачи и выделение всего остального. При этом часть, не являющаяся базовой задачей, должна быть проще, чем исходная задача, иначе рекурсия не сможет завершиться. Процесс продолжается до тех пор, пока не сведется к решению базовой задачи.

Каждый вызов рекурсивной функции называется *рекурсивным вызовом* или *шагом рекурсии*.

Существует много задач, рекурсивных по своей природе: $n!$, a^n , числа Фибоначчи и др.

Пример 11.12. Рекурсивное вычисление факториала.

```
n!=n*(n-1)*(n-2)*...*2*1,
```

причем $0!=1$, $1!=1$ – по определению факториала.

Любая рекурсивная функция может быть вычислена и без применения рекурсии:

```
fact=1;
for (cnt=n; cnt>=1; cnt--)
```

```
fact=fact*cnt; // или fact*=cnt;
```

Рекурсивное решение заключается в многократном вызове функции вычисления факториала из самой этой функции. Рассмотрим для примера вычисление $5!$

$5!=5*4!$

$4!=4*3!$

$3!=3*2!$

$2!=2*1!$

$1!=1$

Вычисление $1!$ – базовая задача, а также признак завершения рекурсии. Таким образом:

$1!=1$ – возвращает 1;

$2!=2*1!=2*1=2$ – возвращает 2;

$3!=3*2!=3*2=6$ – возвращает 6;

$4!=4*3!=4*6=24$ – возвращает 24;

$5!=5*4!=5*24=120$ – возвращает 120.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
// Рекурсивное описание функции факториала:
unsigned long fact(unsigned long i_dat)
{ if (i_dat <=1) return 1;
  else return i_dat*fact(i_dat-1);
}

int _tmain(int argc, _TCHAR* argv[])
{
  cout << "5! = " << fact(5)<< endl;
  return 0;
}
```

Пример 11.13. Рекурсивное вычисление степенной функции.

$$x^n = x \cdot x \cdot x \cdots x \quad x^n = x \cdot x \cdot x \cdots x$$

$$x^n = x^{n-1} \cdot x \quad x^n = x^{n-1} \cdot x$$

...

$x^0 = 1$ – признак завершения рекурсии.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <math.h>
#include <cconio.h>
using namespace std;
// Рекурсивное описание степенной функции:
long power(int number,int x)
```

```

{ if (number == 0) return 1;
  else return x*power(number-1,x);
}
int _tmain(int argc, _TCHAR* argv[])
{ setlocale(LC_ALL, "Russian");
cout << "4 в 5-й степени = " << power(5,4);
_getch();
return 0;
}

```

Если рекурсивная функция объявляет какие-либо локальные переменные, то эти переменные повторно создаются на каждом уровне рекурсии.

Значения параметров рекурсивных функций запоминаются в стеке, что может привести к ошибке его переполнения. Кроме того, вызовы функций отнимают много времени, а нерекурсивные функции могут работать быстрее, чем их рекурсивные эквиваленты.

Обычно рекурсивные решения используются лишь в тех случаях, когда они поддаются отладке легче, чем нерекурсивные.

11.7. Встраиваемые функции

Вызов функции всегда сопровождается дополнительными действиями по обращению к функциям, передачей параметров через стек, передачей возвращаемого значения. Все это ухудшает характеристики программы, в частности ее быстродействие.

Но C++ позволяет задавать функции, которые не вызываются, а встраиваются в программу непосредственно в месте ее вызова. В результате этого в программе создается столько копий встраиваемой функции, сколько раз к ней обращалась вызывающая программа.

Использование встраиваемых функций не связано с механизмами вызова и возврата, следовательно, работают они гораздо быстрее обычных. Однако они способны значительно увеличить объем программного кода.

Для объявления встраиваемой функции указывают спецификатор `inline` перед определением функции.

Пример 11.14. Встраиваемая функция, определяющая четность передаваемого ей аргумента.

```

#include "stdafx.h"
#include <iostream>
using namespace std;
inline int even(int x)
{ return !(x%2); }
int _tmain(int argc, _TCHAR* argv[])
{
int value;
cout << "Input value ";
cin >> value;
if (even(value)) cout << value << " - even" << endl;
else cout << value << " - odd" << endl;
}

```

```

return 0;
}

```

Спецификатор `inline` является не командой для компилятора, а только запросом. Поэтому, если компилятор по каким-либо причинам не может встроить функцию, она компилируется как обычная и никаких сообщений об этом на экран не выдается.

Некоторые компиляторы не могут сделать функцию встраиваемой, если:

- функция содержит операторы цикла,
- функция содержит операторы `switch` или `goto`,
- функция рекурсивная.

Реально компилятор может проигнорировать объявление `inline`, если сочет функцию слишком большой, вследствие чего характеристики программы могут оказаться хуже, чем в случае вызываемой функции.

11.8. Перегрузка функций

В программах, написанных на языке C++, возможно существование нескольких различных функций, имеющих одно имя. При этом однотипные функции обязательно должны отличаться числом и/или типом своих аргументов. Такие функции называются *перегруженными*.

Для того чтобы перегрузить функцию, необходимо задать все требуемые варианты ее реализаций. Компилятор автоматически выбирает правильный вариант вызова согласно числу и типу аргументов. Тип возвращаемого значения при перегрузке большой роли не играет.

Перегруженные функции позволяют упростить программу, допуская обращение к одному имени для выполнения близких по смыслу (но, возможно, алгоритмически различных) действий.

Пример 11.15. Перегруженная функция суммирования: двух целых чисел, трех целых чисел, двух вещественных чисел.

```

#include "stdafx.h"
#include <iostream>
using namespace std;
int sum(int a, int b) {return a + b;}
int sum(int a, int b, int c) {return a + b + c;}
double sum(double a, double b) {return a + b;}
int _tmain(int argc, _TCHAR* argv[])
{
int m, k, l;
double p, q;
cout << " Input m, k, l -> ";
cin >> m >> k >> l;
cout << " Input p, q -> ";
cin >> p >> q;
}

```

```

cout << "m + k = " << sum(m, k) << endl;
cout << "k + l = " << sum(k, l) << endl;
cout << "m + k + l = " << sum(m, k, l) << endl;
cout << "p + q = " << sum(p, q) << endl;
return 0;
}

```

Возможность существования в одной программе нескольких однотипных функций является одним из проявлений *полиморфизма*. Поскольку выбор нужного варианта происходит на этапе компиляции программы, то это – *статический полиморфизм*.

Существует три жестких ограничения на возможность перегрузки функций.

- Функции не могут быть перегружены, если они отличаются только типом возвращаемого значения:

```

int OverloadFn (char *str);
char OverloadFn (char *str);

```

Попытки перегрузить функции таким образом приводят к появлению ошибки компиляции «Type mismatch in redeclaration». Это происходит потому, что компилятор принимает во внимание только аргументы функций.

- Функции не могут быть перегружены, если их аргументы отличаются только использованием ссылок:

```

int OverloadFn (int param);
int OverloadFn (int &param);

```

Такие объявления вызывают сообщения компилятора «'OverloadFn (int)' cannot be distinguished from 'OverloadFn (int &)'».

- Функции не могут быть перегружены, если их аргументы отличаются только применением модификаторов *const* или *volatile*:

```

int OverloadFn (int param);
int OverloadFn (const int param);
int OverloadFn (volatile int param);

```

Модификатор *volatile* сообщает компилятору, что значение переменной может изменяться периферийным устройством или некоторой фоновой процедурой: *volatile int vInt;*.

В языке C++ появилось понятие *ссылка*, которая является альтернативным именем переменной. Использование ссылок будет рассмотрено в гл. 12.

Наличность использования одинаковых идентификаторов для выполнения блоков по смыслу, но, возможно, алгоритмически различных действий называется *полиморфизмом*.

11.9. Использование аргументов по умолчанию

Применение аргументов по умолчанию представляет собой самую простую форму перегрузки функций. При этом один или несколько аргументов функции при ее вызове могут отсутствовать, тогда им передается значение по умолчанию. Для того чтобы задать аргументу значение по умолчанию, нужно в заголовке функции поставить после этого аргумента знак равенства и значение, которое будет передаваться по умолчанию:

```

void Fn(int arg1 = 0, int arg2 = 1000);
void Fn(int = 0, int = 1000);

```

Оба прототипа эквивалентны.

Если при вызове функции часть аргументов отсутствует, то компилятор считает, что эти аргументы находятся в конце списка аргументов:

```

Fn();           // arg1 = 0, arg2 = 1000
Fn(10);        // arg1 = 10, arg2 = 1000
Fn(10,99);     // arg1 = 10, arg2 = 99

```

т. е. нельзя задать явно второй аргумент, задав первый по умолчанию.

В общем случае все параметры по умолчанию должны находиться правее параметров, задаваемых явно.

Аргументы по умолчанию должны быть константами или глобальными переменными.

При создании функции с аргументами по умолчанию их следует задавать единожды: либо в прототипе функции, либо в заголовке ее определения, но не вместе.

Если значение по умолчанию не задано, а соответствующий аргумент при вызове функции отсутствует, то это вызывает ошибки компиляции.

Пример 11.16. Функция возвращает произведение четырех целых чисел, из которых по умолчанию может быть задано от нуля до четырех значений.

```

#include "stdafx.h"
#include <iostream>
#include <stdlib.h>
#include <conio.h>

using namespace std;
int multiple(int m1= 1, int m2= 1, int m3= 1, int m4= 1)
{ return m1*m2*m3*m4; }
int _tmain(int argc, _TCHAR* argv[])
{ cout << multiple(5,6,7,8) << endl;           // 1680
  cout << multiple(3,4,5) << endl;             // 60
  cout << multiple(8,10) << endl;              // 80
  cout << multiple(3) << endl;                 // 3
  getch();
  return 0;
}

```

Контрольные вопросы

1. Дать определение функции. Для чего функции используются в программах?
2. Что такое описание и определение функции?
3. Для чего нужен прототип? Всегда ли его присутствие необходимо в программе, содержащей функции?
4. Признаком чего служит слово void, стоящее перед именем функции?
5. Признаком чего служит слово void, стоящее на месте параметров? Всегда ли оно необходимо?
6. Как осуществляется выход из функции?
7. В чем разница между функциями, не возвращающими никаких значений и возвращающими значения объявленного типа?
8. Что называют формальными параметрами функции и что фактическими? Как эти два понятия соотносятся между собой?
9. Какие переменные называются локальными и какие глобальными?
10. Почему локальные переменные называют автоматическими?
11. Где хранятся локальные переменные? Может ли их суммарный объем превышать доступное стековое пространство?
12. Чем характеризуются регистровые переменные?
13. В чем заключается особенность передачи массивов в качестве параметров функций?
14. Как запретить изменять значения элементов массива, переданного в функцию в качестве параметра?
15. Есть ли принципиальные отличия при передаче массивов и символьных строк как параметров функций?
16. Можно ли возвращать массив из функции, пользуясь оператором return?
17. Что происходит при передаче переменной структурного типа в функцию в качестве ее параметра?
18. Описать механизм возвращения из функции переменной структурного типа.
19. В чем заключаются преимущества и недостатки использования рекурсивных функций?
20. Что представляют собой встраиваемые функции? Перечислить преимущества и недостатки их использования.

Практические задания 11

1. Вычислить $z = (\text{sign } x + \text{sign } y) * \text{sign } (x + y)$, где

$$\text{sign } a = \begin{cases} 1, & \text{при } a < 0 \\ 0, & \text{при } a = 0 \\ -1, & \text{при } a > 0. \end{cases}$$

2. Написать функцию поиска максимального элемента MAX(int a, int b, int c).
 3. Написать две функции: вычисления объема и площади поверхности усеченного конуса, если параметрами обеих функций являются высота и оба радиуса.
 4. Написать функцию вычисления суммы бесконечного ряда $S = \sum_{i=1}^{\infty} \frac{i^2}{x^i}$, задав значение x и точность в качестве параметров функции.
 5. Написать функцию, выполняющую поиск корня уравнения $x^3 + x^2 - 3 = 0$ методом деления отрезка пополам, если параметрами служат значения концов отрезка и требуемая точность.
- При решении следующих задач все логически завершенные фрагменты программ оформлять в виде функций.
6. Задать значения целочисленным элементам массивов
 $M = \{m_i \mid i = 0, 1, \dots, 7\}$, $L = \{l_j \mid j = 0, 1, \dots, 6\}$, $C = \{c_k \mid k = 0, 1, \dots, 8\}$ и вычислить
$$Z = \frac{\prod_{i=0}^7 (m_i - 1) + \prod_{k=0}^8 (c_k - 5)}{\prod_{j=0}^6 l_j - \prod_{i=0}^7 m_i}.$$
 7. Задать значения вещественным элементам массивов
 $A = \{a_i \mid i = 0, 1, \dots, 6\}$, $B = \{b_j \mid j = 0, 1, 2, 3\}$, $C = \{c_k \mid k = 0, 1, 2, \dots, 10\}$ и вычислить
$$Z = \left(\min_{0 \leq i \leq 6} \max \{a_i\}, \max_{0 \leq j \leq 4} \{b_j\}, \max_{0 \leq k \leq 6} \{c_k\} \right).$$
 8. Задать значения целочисленным элементам матриц
 $A = \{a_{ij}\}$, $B = \{b_{ij}\}$,
где $i = 0, 1, 2, 3; j = 0, 1, 2, \dots, 6$,
и сформировать массивы C и D, состоящие из максимальных элементов столбцов матриц A и B соответственно.
 9. Задать значения целочисленным элементам матриц
 $W = \{w_{ij}\}$ и $Z = \{z_{ij}\}$,
где $i = 0, 1, 2; j = 0, 1, 2, \dots, 7$,
и сформировать массивы T и S соответственно из элементов матриц W и Z, больших заданного числа P. Число P также задать как один из параметров функции.
 10. Дан список участников научного семинара:
 - фамилия, имя, отчество,
 - номер секции,
 - ученоое звание,

- город,
- признак докладчик/участник.

Составить программу, позволяющую по желанию получить:

- список участников секции, номер которой введен с клавиатуры;
 - данные по конкретному участнику;
 - список докладчиков из города, название которого введено с клавиатуры.
11. В составе программы описать функцию, определяющую, является ли четным числом количество положительных элементов матрицы float arr[5][7]. Если да, заменить максимальный элемент матрицы ее средним арифметическим значением, если нет, найти минимальное значение и его координаты.

ГЛАВА 12. ПОНЯТИЕ ОБ УКАЗАТЕЛЯХ И ССЫЛКАХ. РЕЗЕРВИРОВАНИЕ ПАМЯТИ

12.1. Объявление указателей

Компьютерная память организована таким образом, что любой фрагмент информации всегда располагается по вполне определенному адресу и занимает известный объем памяти, измеряемой в байтах. При программировании на языках высокого уровня нет необходимости заботиться о размещении своих переменных в памяти, так как эти проблемы успешно решаются компилятором. Тем не менее на определенных этапах программирования бывает удобнее обращаться к переменной, используя ее адрес. Для этой цели служат указатели.

Указатель – это переменная, содержащая адрес другой переменной. Под *адресом* будем понимать местоположение ячейки памяти. При использовании имени переменной мы получаем доступ к ее значению непосредственно, а при употреблении указателей – косвенно.

В общем случае адрес занимает 4 байта и хранится как два слова: одно определяет сегмент, другое – смещение.

Для объявления указателя следует добавить звездочку после идентификатора типа данных, непосредственно или через пробел.

Пример 12.1

```
int i; // i – переменная целого типа;
char c; // с – переменная символьного типа;
int *iPtr; // указатель на переменную целого типа,
            // сохраненную в каком-либо месте памяти;
            // интерпретируется как "*iPtr есть нечто типа int";
            // то же самое, что и объявление int* iPtr;
char *cPtr; // "выражение *cPtr есть нечто типа char"
```

При объявлении более чем одного указателя звездочка ставится перед каждым из них:

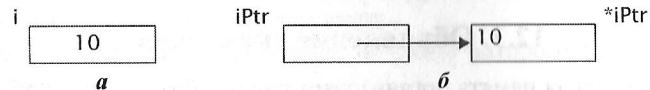
```
double *fPtr1, *fPtr2, *tPtr3; // или double * fPtr1, * fPtr2, * tPtr3;
```

Необходимо запомнить следующие три положения:

- указатель – это переменная, подобная переменной типа int или float;
- переменная типа указателя содержит адрес, который указывает на некоторое место в памяти;
- значения, сохраненные в определенном месте памяти, являются данными, которые адресует указатель (рис. 12.1 и 12.2).



Рис. 12.1. Графическая интерпретация указателя

Рис. 12.2. Ссылки на переменную:
а – без помощи указателя; б – с помощью указателя

Как и другие переменные, помимо объявления, указатели требуют инициализации, так как в противном случае они содержат непредсказуемые значения. При инициализации указателям присваивается значение 0 (NULL) или конкретный адрес.

Никогда не следует использовать неинициализированные указатели, потому что это приводит к ошибкам, которые весьма трудно поддаются отладке. Инициализация указателей более актуальна, чем переменных, так как поведение программы, имеющей неопределенные указатели, непредсказуемо и может привести к потере управления и «зависанию» системы.

12.2. Разыменование указателей

Использование указателей для получения доступа к данным, которые эти указатели адресуют, называется *разыменованием* указателя или операцией *косвенной адресации*.

Пример 12.2

```
int i; // переменная i целого типа; указатель iPtr
int *iPtr; // на целое значение. В данный момент этот указатель
            // не инициализирован и не связан ни с какой переменной.

Можно выполнить присваивание переменной iPtr адреса целой переменной i:
```

```
iPtr = &i; // & – унарная операция адресации, возвращающая адрес
            // своего операнда. Теперь iPtr указывает на переменную i.
```

После такого присваивания переменная i и указатель iPtr оказываются связанными между собой и к значению переменной можно обращаться непосредственно, через имя переменной, или косвенно, через указатель.

Разыменованные указатели на переменные объявленных типов позволяют производить над ними те же действия, что и над переменными соответствующих типов.

Унарный оператор взятия адреса & возвращает адрес объекта в памяти. Оператор & можно использовать для вычисления адресов переменных и функций, но нельзя – для вычисления адресов выражений и символов.

ских констант, объявленных в виде #define name «department of informatics», так как эти объекты не имеют адреса.

Пример 12.3

```
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    int x = 1;
    int *xPtr;
    xPtr = &x;
    x *= 2;           // удвоение значения переменной x непосредственно
    *xPtr *= 2;       // удвоение x путем использования указателя
    *xPtr = *xPtr+10; // увеличение *xPtr (а следовательно, и x) на 10
    // Отобразить значения переменной i на экране можно двумя способами:
    cout << *xPtr << endl; // через указатель
    cout << x << endl;     // непосредственно
    return 0;
}
```

Выражение *xPtr разыменовывает указатель, разрешая доступ к содержащемуся ячейки по адресу xPtr.

Рассмотрим пример, демонстрирующий понятия объявления, инициализации и разыменования переменной типа указатель:

Пример 12.4

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    char c;
    char *cPtr; // объявление указателя на символьную переменную
    cPtr = &c; // инициализация указателя
    for (c = 'A'; c <= 'Z'; c++)
        cout << *cPtr << ' ';
    // на экран будет выведен алфавит
    cout << endl;
    getch();
    return 0;
}
```

Указателю разрешено ссылаться только на объекты заданного типа. Это требование обусловлено тем, что переменные разных типов занимают разное количество ячеек памяти ЭВМ. Исключение составляют указатели на тип void.

Поскольку указатели ограничены заданным типом данных, компилятор осуществляет проверку соответствия типов:

```
float *fPtr;
char c;
fPtr=&c; // ошибка компиляции
```

12.3. Нулевые указатели и указатели на тип void

В языке C++ существуют нулевые указатели. *Нулевой указатель* – это указатель, который в данный момент не адресует никакого достоверного значения в памяти. Значение нулевого указателя равно нулю – единственный адрес, к которому нет доступа. Обычно для удобства где-нибудь в заголовочном файле stddef.h, stdio.h или stdlib.h делается определение `#define NULL 0`.

Часто при программировании используются проверки вида
`if (fPtr != NULL) оператор; // оператор;` выполняется при fPtr,
`// адресующем достоверные данные`

Если указатель равен NULL, он не адресует достоверных данных, поэтому присваивание вида `fPtr = NULL;` или `fPtr = 0;` является инициализацией и может защитить от ошибок, вызванных использованием неинициализированных указателей.

Как и другие глобальные переменные, глобальные указатели инициализируются равными нулю, во всех остальных случаях требуется явная инициализация:

```
float *fPtr = NULL;
```

В языке C++ можно встретить также указатели типа void, которые указывают на неопределенный тип данных:

```
void *SomePlace;
```

Такие указатели не ограничиваются определенными типами данных и могут адресовать любое место в памяти. Указатель на void представляет собой обобщенный указатель для адресации данных без необходимости заранее определять их тип.

Не следует путать нулевой указатель и указатель на void. Нулевой указатель не адресует достоверных данных, а указатель на void адресует данные неопределенного типа и также может быть нулевым.

В основном указатели на void применяются для некоторых специальных целей, например:

- для адресации буферов;
- заполнения блоков памяти;
- чтения аппаратных регистров;
- передачи указателей в функции, которые работают независимо от типа данных, передаваемых по указателю в качестве параметров.

Указатель можно присваивать другому указателю, если оба они имеют одинаковый тип. Исключение – указатель на void, которому можно присваивать указатели любых типов, но сначала указатель на void должен быть приведен к типу соответствующего указателя.

Указатель на void нельзя разыменовать, пока он не приведен к какому-либо конкретному типу, так как неизвестно, на какой объем памяти он указывает.

Пример 12.5

```
char buffer[1024];           // 1024-байтовый буфер
void *bPtr;
bPtr = buffer;               // указателю bPtr присвоен адрес массива buffer.
```

Если при этом задать еще переменную с символьного типа
`char c;`

то можно записать
`c = *(char *)bPtr;`

Запись `(char *)bPtr` приводит к тому, что компилятор временно считает `bPtr` указателем на тип `char`. Разыменование выражения `*(char *)bPtr` относится к содержимому, которое адресуется указателем `bPtr` и имеет значение типа `char`.

Для типа `int`

```
i = *(int *)bPtr;
```

указатель `bPtr` будет адресовать данное целого типа.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    int i;
    float f;
    int *iPtr;
    float *fPtr;
    void *vPtr;
    iPtr = &i;
    fPtr = &f;
    vPtr = &i;
    vPtr = &f;
    fPtr = &i;                                // ошибка трансляции
    iPtr = &f;                                // ошибка трансляции
    getch();
    return 0;
}
```

Существует понятие *ближнего* (near) и *дальнего* (far) указателей, которые используются для согласования с сегментированной архитектурой памяти процессора 80 × 86; 16-разрядный ближний указатель может адресоваться к памяти в пределах 64 Кбайт (65 536 байт) любого сегмента, который может начинаться с любой 16-битовой адресной границы, сохраняемой в регистре сегмента, принадлежащего к процессору. Такие указатели хранят только смещение в текущем сегменте, поэтому им дос-

таточно 16 байт; 32-разрядные дальние указатели включают еще значение адреса сегмента и поэтому могут адресоваться к любой области памяти в пределах 1 Мбайт:

```
float near *fPtr1;
float far *fPtr2;
```

Лучше, однако, объявлять указатели без квалификаторов *near* и *far*. При их отсутствии компилятор имеет возможность корректировать форматы хранения указателей и их размеры в соответствии с различными моделями памяти.

12.4. Ссылки

Ссылка на некоторую переменную может рассматриваться как указатель, который при работе с ним всегда разыменовывается. Для ссылки не требуется дополнительного пространства в памяти: она является просто другим именем или псевдонимом переменной. Для определения ссылки применяется унарный оператор *&*:

```
тип &идентификатор_1 = идентификатор_2;
```

Пример 12.7

```
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    int a = 5, b = 10;
    int &aRef = a;           // aRef является ссылкой на a
    aRef = b;               // a равно b
    aRef++;
    cout << a << endl;
    return 0;
}
```

Ссылка не создает копию объекта, а лишь является другим именем объекта. Чаще всего ссылки используются для передачи параметров в функции.

12.5. Резервирование памяти в куче

Переменные, находящиеся в фиксированных областях памяти, называются *статическими* и объявляются прямо в тексте программы. Память для них резервируется на этапе трансляции.

Существует способ выделения памяти для переменных во время выполнения программы. Такие переменные называются *динамическими*, они создаются по требованию программы и запоминаются в блоках памяти переменного размера, принадлежащих к специальному образом организованной области памяти — куче. Память в куче резервируется и освобождается, не подчиняясь какой-либо структуре. Динамические переменные также называются *переменными адресуемыми* указателями.

Существует несколько способов резервирования памяти в куче для динамических переменных. В C++ наиболее распространенным способом является использование операции *new*, которая резервирует указанный блок памяти в куче и присваивает указателю адрес начала выделенного блока или 0, если по каким-либо причинам резервирование невозможно:

```
int *iPtr;
iPtr = new int;
```

Оператор резервирования может отказать, если куча уже заполнена или если память фрагментирована таким образом, что любое ее свободное пространство меньше, чем требуемое количество байтов. В этом случае указателю присваивается нулевое значение. Следовательно, необходимо производить проверку:

```
if (iPtr == NULL)      // если нет нужного числа свободных байтов,
    exit(0);           // происходит вызов функции exit()
```

При резервировании памяти посредством оператора *new* допускается сразу присвоить значение созданному объекту:

```
float *fPtr = new float(25.759);
```

В результате этой операции переменной, адресуемой указателем *fPtr*, будет присвоено значение 25.759.

После выделения блока памяти и присваивания его адреса указателю можно разыменовать этот указатель и использовать память, как если бы она принадлежала к статической переменной.

После того как необходимость в использовании динамической переменной отпала, занимаемую ею память следует освободить. Это можно сделать с помощью операции *delete*:

```
delete iPtr;
delete fPtr;
```

Операция *delete* не требует указывать размер освобождаемой памяти, так как в C++ существует специальный механизм, позволяющий автоматически связывать с указателем размер области, на которую он указывает.

При работе с операторами *new* и *delete* не нужно подключать заголовочный файл *alloc.h*, который необходим при резервировании памяти с помощью функций резервирования, таких, как *malloc()* или *calloc()*, которые чаще используется в языке C.

Вызов функции *malloc()* — memory allocation выделяет для программы заданное количество байтов в куче. Функция возвращает адрес выделенного блока, который обычно присваивается указателю.

Пример 12.8

```
#include <alloc.h>           // подключить файл alloc.h
```

```
char *sPtr; // объявить указатель на тип char
sPtr=(char *)malloc(129); // резервирование 129 байт
```

Количество выделяемых в куче байтов может определяться с помощью функции sizeof():

```
vPtr=malloc(sizeof(double));
```

После этого, если указатель vPtr не равен нулю, он будет адресовать выделенную в куче память, точно соответствующую размерам объявленного типа. Теперь можно разыменовать vPtr и присвоить ему значение `*vPtr=3.5926;`

Для резервирования памяти вместо функции malloc() можно использовать функцию calloc(), прототип которой также объявлен в файле alloc.h

Эта функция, как и malloc(), резервирует память в куче, но требует два аргумента – количество объектов, которое необходимо разместить, и размер одного объекта.

```
long *lPtr;
lPtr=calloc(10,sizeof(long));
```

Этот оператор резервирует память для 10 длинных значений и присваивает указателю lPtr адрес первого значения.

Параметры функции calloc() могут быть заданы в любом порядке, т. е. calloc(num,size) резервирует num*size байт так же, как и calloc (size, num).

Кроме выделения памяти, функция calloc() устанавливает каждый зарезервированный ею байт равным нулю, что очень удобно для инициализации памяти.

По окончании работы с выделенным блоком памяти следует освободить его, чтобы функции резервирования памяти могли снова использовать эту область памяти. В противном случае доступ к неосвобожденной памяти невозможен до конца работы программы. Это одна из наиболее распространенных ошибок, являющаяся причиной «утечки памяти».

Освобождение памяти, которая была зарезервирована функциями malloc(), calloc() и им подобным, осуществляется функцией free():

```
char *sPtr;
sPtr=malloc(256);
```

```
free (sPtr);
```

Задавать размер освобождаемой памяти также не нужно, так как он запоминается в нескольких байтах, примыкающих к каждому зарезервированному блоку, и совпадает с объемом первоначально зарезервированной памяти, с которой связан указатель.

После вызова функции free(Ptr) или операции delete Ptr для освобождения памяти, адресуемой указателем Ptr, ни в коем случае нельзя использовать этот указатель для выполнения чтения или записи в эту память. Можно зарезервировать другой блок памяти и присвоить адрес указателю Ptr, но значение освободившегося адреса, содержащегося в указателе, недействительно и не должно использоваться. Во избежание случайного использования незащищенных областей памяти можно присвоить значение NULL указателям на освободившуюся память.

Контрольные вопросы

- Что такое указатель?
- Привести примеры объявления указателей.
- Почему указатели необходимо инициализировать?
- Для чего используется оператор взятия адреса?
- Что такое нулевой указатель и указатель на тип void? В чем заключается их принципиальное отличие?
- Куда указывает нулевой указатель и куда неинициализированный?
- Каким образом указатель на тип void может быть использован для обращения к типизированным данным?
- Определить статические и динамические переменные.
- Что такое куча? Какие элементы она содержит?
- Что общего у функций malloc() и calloc() и в чем состоят их различия?
- Как освободить память?
- Всегда ли возможна работа с кучей?
- Можно ли использовать содержимое динамически выделяемой памяти после того, как она была освобождена?
- Почему функции free() нет необходимости сообщать, сколько байтов нужно освобождать?
- Чем может быть вызвана «утечка памяти»?
- Можно ли захватить память функциями malloc() или calloc() и освободить ее операцией delete?
- Как избежать случайного использования незащищенных областей памяти?
- Как при резервировании памяти сразу присвоить значение созданному объекту?

Практические задания

- Написать комментарии к каждой строке фрагмента программы

```
int x = 1, y = 2, z[10];
int *ip;
ip = &x;
```

```
y = *ip;
*ip = 0;
ip = &z[0];
```

2. Найти ошибки, если они есть, в следующей программе. Что будет напечатано в результате ее работы?

```
#include <iostream.h>
{ int *xp;
  int *wp;
  int y;
  int z;
  *xp = 24;
  y = -35;
  wp = &y;
  cout << "Размер указателя xp = " << sizeof(xp) << endl;
  cout << "Значение указателя xp = " << xp << endl;
  cout << "Значение, адресуемое xp" << *xp << endl;
  cout << "Адрес переменной y = " << &y << endl;
  cout << "Адрес переменной z = " << &z << endl;
  cout << "Разыменованное значение wp = " << *wp << endl;
}
```

ГЛАВА 13. УКАЗАТЕЛИ И СТРУКТУРЫ ДАННЫХ

Указатели могут адресовать переменные всех видов – от простых, таких, как int, char, double и т. д., до массивов и структур.

13.1. Указатели и массивы

В языке C++ связь между указателями и массивами настолько тесная, что программисты обычно предпочитают использовать указатели при работе с массивами.

Любой доступ к элементу массива, осуществляемый индексированием, может быть выполнен с помощью указателя (рис. 13.1). При этом обычно программа работает быстрее, а во многих случаях имеет место экономия памяти.

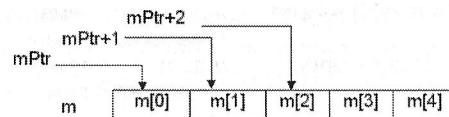


Рис. 13.1. Связь между элементами массива и указателями

Объявление

```
int m[5];
```

определяет массив a из пяти элементов.

Запись m[i] позволяет обратиться к i-му элементу массива. Если mPtr – указатель на int:

```
int *mPtr;
```

то в результате присваивания

```
mPtr=&m[0];
```

mPtr будет указывать на нулевой элемент массива m, т. е. содержать адрес элемента m[0]. Таким образом, в определенном смысле выражение m[0] является аналогом разыменованного указателя mPtr.

Присваивание

```
int x=*mPtr;
```

будет копировать содержимое m[0] в переменную x.

Если mPtr указывает на некоторый элемент массива, то mPtr+1 указывает на следующий элемент, mPtr+i – на i-й элемент после mPtr, а mPtr-i – на i-й элемент перед mPtr. Другими словами, если mPtr указывает на m[0], то *(mPtr+i) есть содержимое m[i].

Тип и размер элементов массива m при этом не имеют значения.

Смысл слов «добавить единицу к указателю» заключается в том, что ука-

затель будет указывать на следующий объект объявленного типа, а не на следующую ячейку памяти.

Пример 13.1

```
#include "stdafx.h"
#include <iostream>
using namespace std;
const int M = 10;           // размер массива
int mas[M];                // глобальный массив из M целых значений
int _tmain(int argc, _TCHAR* argv[])
{mas[0]=123;               // присваивание значения с помощью индекса
// на экран дважды будет выведено значение 123
cout << "mas[0]=" << mas[0] << endl; // доступ к элементу
                                         // массива с помощью индекса
cout << "*mas=" << *mas << endl; // доступ к элементу
                                         // массива с помощью указателя
*mas = 456;                 // присваивание значения с помощью указателя
                                         // на экран дважды будет выведено значение 456
cout << "mas[0]=" << mas[0] << endl; // доступ к элементу
                                         // массива с помощью индекса
cout << "*mas=" << *mas << endl; // доступ к элементу
                                         // массива с помощью указателя
return 0;
}
```

Несмотря на то что объявления `int m[5];` и `int *mPtr;` во многом схожи, они не являются полными аналогами. Переменная `mPtr` может указывать на начало массива `m`, и тогда результаты выражений `m[i]`, `*(m+i)`, `*(mPtr+i)` будут одинаковы. Однако можно написать выражение `*(mPtr++)`, но нельзя `*(m++)`, потому что `mPtr` является указателем переменной, содержащей некоторый адрес, а `m` – указатель-константа, который сам является адресом нулевого элемента массива `m`.

Поскольку массивы являются, хотя и неполными, аналогами указателей, язык C++ позволяет программам разыменовывать имена массивов с помощью такого выражения, как ***имя_массива**.

Пример 13.2. Адресация элементов массивов с помощью указателей.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
const int M = 10;           // размер массива
int mas[M];                // глобальный массив из M целых значений
int _tmain(int argc, _TCHAR* argv[])
{int *mPtr;                // mPtr – указатель на массив значений типа int
mPtr = mas;
// присваивание значений элементам массива,
// осуществляющееся с помощью индексов
for (int i=0; i< M; i++) mas[i]=i;
// вывод значений элементов массива с помощью указателей
```

```
for (int i=0; i<M; i++) cout << *mPtr++ << ' ';
cout << endl;
for (int i=0; i< M; i++)
{ mPtr=&mas[i];
  cout << *mPtr << ' ';
}
cout << endl;
for (int i=0; i< M; i++) cout << *(mas+i) << ' ';
cout << endl;
return 0;
}
```

Использование оператора взятия адреса от имени массива (в данном случае `&mas`) вызывает ошибку компиляции, так как имя массива (здесь `mas`) – это уже указатель. Но взятие адреса от элемента массива (например, `&mas[5]`) вполне допустимо.

Выражение `*mPtr++` разыменовывает `*mPtr` и передвигает указатель `mPtr` к следующему элементу массива. Это возможно, поскольку известен тип адресуемых данных, а значит, и их размер.

В арифметических выражениях, связанных с указателями, допустимы только операции сложения и вычитания. При этом арифметическое выражение вида `Ptr+=5` перемещает указатель вперед на 5 элементов от текущей позиции, а `Ptr-=3` возвращает его на 3 позиции назад.

Кроме того, указатели можно сравнивать с помощью выражений вида (`Ptr1<Ptr2`) или (`Ptr1>=Ptr2`). Однако эти операторы сравнения работают корректно лишь при сравнении близких указателей. В случае дальних указателей `far` сообщений об ошибке компилятор не выдает, но гарантировать правильность работы этих выражений нельзя.

Причиной этого является сегментированность памяти компьютеров с процессорами 80x86, которая позволяет адресовать один и тот же физический адрес несколькими значениями сегментов и смещений. Например, адрес `Ptr1=0000:0010` физически эквивалентен адресу `Ptr2=0001:0000`, но при сравнении их указателей мы получим ложное значение выражения (`Ptr1==Ptr2`).

Одним из способов решения этой проблемы является использование указателей `huge`, которые физически не отличаются от дальних (`far`), но имеют нормализованные значения смещений в диапазоне от 0(10) до 15(10). При этом каждая ячейка памяти имеет уникальный нормализованный дальний адрес, соответствующий ее физическому адресу.

Объявление указателей `huge` аналогично объявлению указателей `far`:

```
char huge *fPtr;
```

Однако при этом следует помнить о том, что нормализация требует времени, поэтому программы, имеющие `huge`-указатели, работают медленнее программ с дальными указателями (`far`).

Использование указателей при работе с массивами является мощным средством экономии памяти. Существует ряд программ, требующих объявления массивов, но не использующих их полностью. Например, программа, объявляющая массив

```
double BigArr[100];
```

но использующая лишь половину элементов в силу некоторых условий, понапрасну теряет 50 значений по 8 байт, т. е. 400 байт памяти.

Если размер массива точно не определен, лучше использовать динамические массивы, объявленные с применением указателей.

Сначала следует объявить указатель требуемого типа данных

```
double *BigArrPtr;
```

затем где-нибудь в программе выделить память в куче для массива

```
BigArrPtr = new double [50];
```

После этого с массивом можно работать обычным образом, например

```
for (int i = 0; i < 50; i++)
    BigArrPtr[i] = rand()%100;
```

После использования массива следует освободить его память операцией

```
delete [] BigArrPtr;
```

Теперь можно снова использовать освобожденный объем памяти посредством повторного резервирования памяти.

Инициализировать массив, для которого память выделяется динамически, невозможно.

Если при удалении массива оператором `delete` не указать квадратные скобки, то результат окажется непредсказуемым.

Пример 13.3. Дан массив из 100 целочисленных элементов. Создать другой массив, содержащий только четные элементы исходного массива.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
#include <stdlib.h>
using namespace std;
const int M = 100;
int _tmain(int argc, _TCHAR* argv[])
{
    int mas[M];           // исходный статический массив из M целых чисел
    int *mPtr;             /* mPtr - указатель на массив значений типа int
    int ent = 0;           // счетчик четных элементов
    int j = 0;
    for (int i=0; i< M; i++)
    {
        mas[i] = rand()%100-50;
        cout << setw(4) << mas[i];
    }
}
```

```
    if (mas[i]%2 == 0) cnt++;
}
cout << endl;
mPtr = new int[cnt];
for (int i=0; i< M; i++)
{
    if (mas[i]%2 == 0)
    {
        mPtr[j]=mas[i];
        cout << setw(4) << mPtr[j];
        j++;
    }
}
cout << endl;
return 0;
}
```

Пример 13.4. Дан массив из 20 целых чисел. Создать другой массив, содержащий элементы исходного массива, меньшие числа, введенного с клавиатуры.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <time.h>
#include <conio.h>
using namespace std;
const int N = 20;
int _tmain(int argc, _TCHAR* argv[])
{
    time_t t;
    srand((int)time(&t));
    setlocale(LC_ALL, "Russian");
    int mas[N];           // исходный статический массив
    int *mPtr;             /* *mPtr - указатель типа int
    int cnt = 0;           // счетчик
    int j = 0;
    int r;                 // число, вводимое с клавиатуры
    cout << "Введите число ";
    cin >> r;
    cout << "Исходный массив:\n ";
    for (int i=0; i< N; i++)
    {
        mas[i] = rand()%100-50;
        cout << setw(5) << mas[i];
        if (mas[i] < r) cnt++;
    }
    cout << endl;
    mPtr = new int[cnt];
    cout << "Новый массив:\n ";
    for (int i=0; i< N; i++)
    {
        if (mas[i] < r)
        {
            mPtr[j]=mas[i];
            cout << setw(5) << mPtr[j];
            j++;
        }
    }
}
```

```

    }
    _getch();
    return 0;
}

```

13.2. Указатели и многомерные массивы

Создадим указатели для многомерных массивов.

Пусть описаны

```

int arr[4][2];           // массив [4x2]
int *arr_Ptr;            // указатель на целый тип.

```

В этом случае arr_Ptr=arr указывает на первый элемент первой строки: arr_Ptr==&arr[0][0], arr_Ptr+1 – на arr[0][1] и т. д. согласно расположению элементов массива в памяти, которое всегда однозначно: arr[0][0], arr[0][1], arr[1][0], arr[1][1], arr[2][0], arr[2][1], arr[3][0], arr[3][1]. Другими словами, сначала запоминаются элементы первой строки, за ней – элементы второй строки и т. д. Тогда

```

arr_Ptr ==&arr[0][0]
arr_Ptr +1 ==&arr[0][1]
arr_Ptr +2 ==&arr[1][0]
...
arr_Ptr +5 ==&arr[2][1]

```

Мы описали двумерный массив как массив массивов:

```

arr[0]==&arr[0][0]
arr[1]==&arr[1][0]
arr[2]==&arr[2][0]
arr[3]==&arr[3][0]

```

Выделить память под двумерный массив с помощью функции malloc() можно, перемножив количество строк и столбцов:

```
ArrPtr=malloc(Row*Col*sizeof(double)); ... free(ArrPtr);
```

А, воспользовавшись операцией new, можно сделать это следующим образом:

```
int *ArrPtr = new int[Row*Col]; ... delete []ArrPtr;
```

При необходимости создать именно двумерный массив поступают, как показано в примере 13.5.

Пример 13.5

```

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <time.h>
#include <conio.h>
using namespace std;

```

```

const int c = 5;
int _tmain(int argc, _TCHAR* argv[])
{
    time_t t;
    srand((int)time(&t));
    int *arr[c], i, j, r;
    cout << "Input r ";
    cin >> r;
    for (i = 0; i < r; i++)
    {
        arr[i] = new int [c];      // количество столбцов должно быть константой
        for (j = 0; j < c; j++)
        {
            arr[i][j] = rand()%50;
            cout << arr[i][j] << endl;
        }
    }
    cout << endl;
    for (i = 0; i < r; i++) delete []arr[i];
    _getch();
    return 0;
}

```

При работе с двумерными массивами можно представлять их как одномерные, пользуясь формулой

$i_{current} = Nx \cdot iy + ix$,

где $i_{current}$ – текущее значение координаты в одномерном массиве; Nx – количество столбцов в матрице; iy, ix – координаты точки $i_{current}$ в матрице (рис. 13.2).

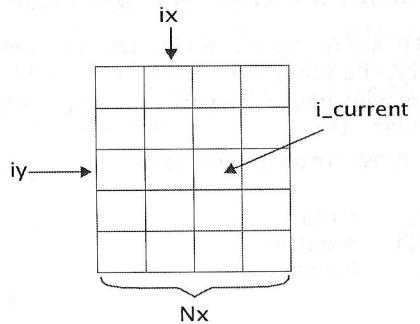


Рис. 13.2. Представление матрицы в виде одномерного массива

13.3. Строковые указатели

Строковые указатели являются адресами, которые определяют местонахождение первого символа строки, сохраненной в памяти. Строковые указатели объявляются как char*:

```
char *stringPtr;
```

При выделении памяти в куче для строкового указателя пользуются оператором new или функцией malloc() или ей подобной:

```
stringPtr=new char[81]; // 80 элементов строки и 1 дополнительный
                        // байт для завершающего нуля
cout << stringPtr[2]; // отображает третий символ в строке
delete [] stringPtr; // освобождает место в куче
```

Пример 13.6

```
#include "stdafx.h"
#include <iostream>
#include <stdlib.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ char sArr[] = "String as array";
char *sPtr = "String as pointer";
cout << sArr << endl; // sArr - указатель-константа
cout << sPtr << endl; // sPtr - указатель-переменная
return 0;
}
```

Строки, объявленные с помощью указателей, оказываются более удобными в применении, особенно при передаче их в качестве аргументов функций.

Строки часто определяют как символьные массивы, поэтому над ними допустимы те же действия, что и над массивами.

13.4. Указатели и структуры

Язык C++ позволяет не только объявлять динамические массивы, но и адресовать структуры с помощью указателей. Если указатель адресует структуру, то для обращения к членам структуры используется специальный синтаксис.

Объявим структуру, которую можно применять для запоминания даты year, month, day:

```
struct ymd { int year; // год
             int month; // месяц
             int day; // день
};
```

Теперь можно объявить переменную Data типа struct ymd, записав:

```
ymd Data; // Data – структурная переменная типа ymd.
```

Чтобы присвоить значения полям структурной переменной Data, воспользуемся операторами

```
Data.year = 2014;
Data.month = 7;
Data.day = 2;
```

Однако для более рационального использования компьютерной памяти большие структуры лучше размещать в куче, а не в глобальных или в локальных переменных.

Объявим указатель на структурный тип

```
ymd *dat_Ptr; // *dat_Ptr – указатель на структуру типа ymd
```

Затем вызовем оператор new для выделения памяти в куче и присвоим адрес указателю dat_Ptr:

```
dat_Ptr = new ymd;
```

Теперь указатель dat_Ptr адресует блок памяти, размер которого как раз такой, чтобы запомнить одну структуру типа ymd, но уже нельзя использовать привычный способ доступа к полям структуры. Описанные нами операторы доступа к полям структуры

```
dat_Ptr.year = 2001; // это не работает,
dat_Ptr.month = 7; // ошибка
dat_Ptr.day = 2; // трансляции
```

не будут работать.

Указатель dat_Ptr является указателем на структуру, поэтому после того, как он получит адрес структурной переменной, используют символ -> :

```
dat_Ptr->year = 2001;
dat_Ptr->month = 7;
dat_Ptr->day = 2;
```

Оператор доступа к полю структуры ->, стоящий между указателем структуры и именем поля, указывает на его расположение в памяти относительно начала структуры. Указатель структуры можно разыменовать обычным способом. Выражение *dat_Ptr относится к структуре, адресуемой указателем dat_Ptr. Выражение (*dat_Ptr).day эквивалентно dat_Ptr->day.

Пример 13.7

```
#include "stdafx.h"
#include <iostream>
#include <stdlib.h>
using namespace std;
struct ymd { int year;
             int month;
             int day; };
int _tmain(int argc, _TCHAR* argv[])
{ ymd *dat_Ptr; // объявление указателя на структуру ymd
dat_Ptr = new ymd;
if (dat_Ptr == NULL)
{ cout << "Memory ERROR" << endl;
exit(1);
}
dat_Ptr->year = 2014; // присваивание 2014 полю year
```

```

dat_Ptr->month = 4; // присваивание 4 полю month
dat_Ptr->day = 20; // присваивание 20 полю day
cout << "Year = " << dat_Ptr->year << endl;
cout << "Month = " << dat_Ptr->month << endl;
cout << "Day =" << dat_Ptr->day << endl;
return 0;
}

```

Пример 13.8. Дан массив переменных структурного типа, определяющего следующие метеорологические данные в апреле 2014 г.: число, дневная и ночная температура, скорость ветра, видимость на дорогах. Создать массив, в котором разместить данные о днях, когда среднесуточная температура была положительной.

```

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <time.h>
#include <conio.h>
using namespace std;
struct meteo { int day;
    double t_d;
    double t_n;
    double s_w;
    int road;
};
const int M = 30; // размер массива
int _tmain(int argc, _TCHAR* argv[])
{ setlocale(LC_ALL, "Russian"); // подключение русификатора
meteo *i_Ptr; // исходный динамический массив
meteo *o_Ptr; // полученный массив
int j = 0, cnt = 0; // cnt - счетчик дней
i_Ptr = new meteo[M];
for (int i = 0; i < M; i++)
{ i_Ptr[i].day = i + 1;
cout << "Дата:" << i_Ptr[i].day << "апреля " << endl;
cout << "Дневная температура: ";
cin >> i_Ptr[i].t_d;
cout << "Ночная температура: ";
cin >> i_Ptr[i].t_n;
cout << "Скорость ветра: ";
cin >> i_Ptr[i].s_w;
cout << "Видимость на дорогах: ";
cin >> i_Ptr[i].road;
if ((i_Ptr[i].t_d+i_Ptr[i].t_n)/2 > 0) cnt++;
}
cout << endl;
o_Ptr = new meteo[cnt];
for (int i=0; i < M; i++)
{ if ((i_Ptr[i].t_d+i_Ptr[i].t_n)/2 > 0)

```

```

{ o_Ptr[j]=i_Ptr[i];
cout << setw(5) << o_Ptr[j].day << " апреля: " << ' ' << o_Ptr[j].t_d << ''
<< o_Ptr[j].t_n << ' ' << o_Ptr[j].s_w << ' ' << o_Ptr[j].road << endl;
j++;
}
delete []i_Ptr;
delete []o_Ptr;
_getch();
return 0;
}

```

Следует помнить о необходимости удалять динамические переменные, а тем более массивы, после того, как работа с ними завершена. Если обнаруживается, что память в системе «исчезает», значит, с большой степенью вероятности в программе резервируется память, которая не освобождается.

13.5. Указатели и модификатор const

Модификатор `const` указывает компилятору на то, что значение переменной, с которой он используется, не должно изменяться.

Поскольку указатель – это переменная (содержащая адрес другой переменной), то модификатор `const` может использоваться и с указателями. Но есть особенности, связанные с тем, что можно объявить **константный указатель**, а можно – **указатель на константу**, и эти понятия не являются синонимами.

Объявления константного указателя и указателя на константу различаются синтаксически:

```

const тип *имя_указателя; // указатель на константу
типа* const имя_указателя; // константный указатель

```

Например

```

int i = 5, j;
const int *icPtr = &i; // указатель на константу
int* const iPtr_c = j; // константный указатель

```

При использовании указателя на константу нельзя изменять значение той переменной, на которую указывает указатель, но можно изменять значение самого указателя, т. е. оператор `icPtr++`; или `icPtr = &j;` транслируется и выполняется, а оператор `*icPtr = 100;` – вызывает ошибку трансляции.

При работе с константным указателем можно изменять значение той переменной, на которую он указывает, но нельзя изменять значение самого указателя. То есть при попытке написать оператор `iPtr_c++`; или

iPtr_c = &j; возникает ошибка трансляции, а оператор **iPtr_c = 100;* – транслируется и выполняется.

Эти возможности модификатора *const* особенно широко используются при передаче указателей в качестве параметров функций.

Контрольные вопросы

- Что могут адресовать указатели?
- К чему приводит увеличение указателя на некоторую величину?
- Почему можно разыменовать имя массива, но нельзя определить его адрес?
- Как осуществляется работа с динамически выделенным массивом?
- Как выделить память под матрицу?
- Как происходит обращение к полям структуры, если она адресуется указателем?
- На что указывает оператор доступа к полю структуры *->*, стоящий между указателем структуры и именем поля?
- Что происходит, если не освобождать память, занятую под массивы и структуры после завершения работы с ними?

Практические задания 13

- Дан вектор, содержащий 25 вещественных элементов. Сформировать два других вектора, один из которых содержит элементы исходного вектора, большие среднего арифметического значения его элементов, другой – меньшие.
- Задать значения элементам вещественных динамических массивов $A = \{a_i\}$ и $B = \{b_i\}$, где $i = 0, 1, 2, \dots, 9$ и вычислить элементы массивов $X = \{x_i\}$ и $Y = \{y_i\}$ по формулам:

$$x_i = \frac{\sin(a_i)}{\sin(a_i)^2 - \sin^2(a_i)}; \quad y_i = \frac{\operatorname{tg}(b_i)}{\operatorname{tg}(b_i)^2 - \operatorname{tg}^2(b_i)}.$$

- Дана матрица $A[7][5]$ целого типа, содержащая как положительные, так и отрицательные элементы. Сформировать динамические одномерные массивы, содержащие:
 - отрицательные элементы исходной матрицы;
 - нечетные элементы исходной матрицы;
 - положительные четные элементы исходной матрицы;
 - элементы исходной матрицы, меньшие числа, введенного с клавиатуры.
- Дана вещественная квадратная матрица 5-го порядка. Прибавить к элементам каждой строки матрицы наименьшее значение элементов этой строки. Сформировать динамический одномерный массив, со-

держащий количество четных элементов в четных столбцах полученной матрицы и количество нечетных элементов в нечетных столбцах.

- Дана действительная матрица 7×7 . Найти минимальное значение среди элементов, стоящих над главной диагональю, и максимальное среди элементов, находящихся ниже главной диагонали, а также их местоположение. Сформировать два динамических одномерных массива, содержащих сумму элементов четных строк и произведение элементов нечетных строк.
- Дана база данных библиотеки, реализованная в виде динамического массива структур:
 - фамилия автора;
 - его имя и отчество;
 - название книги;
 - год издания;
 - форма доступа: читальный зал или абонемент;
 - признак, указывающий, есть ли книга в наличии или находится у читателя.

Составить программу, позволяющую:

- по введенной с клавиатуры фамилии автора получить список его книг;
 - найти книги указанного автора, изданные не позднее заданного года, имеющиеся в наличии на абонементе.
 - Дана база данных автомобилистов, реализованная в виде динамического массива структур: фамилия, имя, отчество, адрес, реализованный как вложенная структура, марка автомобиля, цвет, номерной знак, дата получения водительских прав.
- Составить программу, позволяющую:
- по введенной с клавиатуры фамилии получить полную информацию;
 - по номеру автомобиля найти его владельца;
 - выделить всех владельцев автомобилей указанной марки и цвета.

ГЛАВА 14. УКАЗАТЕЛИ И ФУНКЦИИ

Функции могут принимать указатели в качестве своих аргументов, а также возвращать их в вызывающую функцию. В C++ имеется несколько видоизмененная форма указателя, называемая *ссылкой*. Ссылки также широко используются в качестве параметров функций. Кроме того, указатель может адресовать функцию, т. е. указывать код, отличный от данных.

14.1. Способы передачи параметров

Передача аргументов в функцию может происходить:

- по значению,
- по указателю,
- по ссылке.

Если функция в процессе своей работы должна изменить значение аргумента, то его можно передать по ссылке или по указателю, так как передача аргумента по значению приводит к созданию его копии в стеке. В момент завершения работы функции стековое пространство, в котором находятся аргументы функции, очищается, уничтожая находящиеся в нем данные. При этом значение-оригинал оказывается недоступным для функции.

При использовании ссылки для передачи в функцию ей передается фактически разыменованный адрес переменной, являющейся аргументом. Таким образом, все изменения, которые производит функция, выполняются непосредственно над переменной, ссылка на которую передается в функцию.

При передаче в функцию аргумента по указателю функция получает адрес этого аргумента и может применить операцию разыменования, получая таким образом доступ непосредственно к переменной, находящейся по этому адресу.

Пример 14.1. Передача параметров функции по значению, по ссылке и с использованием указателя.

```
#include "stdafx.h"
#include <iostream>
#include <stdlib.h>
#include <conio.h>
using namespace std;
int Square_Val ( int i ) // параметр-значение
{
    i++;
    return i*i;
}
```

```
void Square_Ref ( int &i ) // параметр-ссылка
{
    i = i * i;
}
void Square_Ptr ( int *i ) // параметр-указатель
{
    *i = *i * *i;
}
int _tmain(int argc, _TCHAR* argv[])
{
int v = 5, s, p = 5;
s = Square_Val(v);
cout << "s = " << s << " v = " << v << endl; //s=36 v=5
Square_Ref(p);
cout << "p = " << p << endl; // p = 25
Square_Ptr(&p);
cout << "p = " << p << endl; // p = 625
_getch();
return 0;
}
```

14.2. Передача массивов посредством указателей

При передаче массива в функцию без использования указателей, функция имеет возможность изменять значения массива (если, конечно, он передан не как массив констант). Это происходит вследствие того, что в языке C++ массивы и указатели имеют много общего и имя массива фактически является константным указателем на свой массив.

Однако передача массивов в функции по указателю используется чаще, хотя с точки зрения работы программы оба способа эквивалентны.

Синтаксис прототипа функции с передачей массива по указателю:

тип_возвращаемого_значения имя_функции (тип_элементов массива * имя_указателя);

Пример 14.2

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;
const int N = 10;
// функция работает с именем массива:
void Fn_Name(int arr[], int n);
// функция работает с указателем на массив:
void Fn_Ptr(int *aPtr, int n);
int _tmain(int argc, _TCHAR* argv[])
{
int *mPtr, mas[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
mPtr = mas;
for (int i = 0; i < N; i++) cout << setw(4) << mas[i];
cout << endl;
Fn_Name(mas, N);
for (int i = 0; i < N; i++) cout << setw(4) << mas[i];
cout << endl;
Fn_Ptr(mPtr, N);
```

```

for (int i = 0; i < N; i++) cout << setw(4) << mas[i];
cout << endl;
_getch();
return 0;
}

void Fn_Name(int arr[], int n)
{ for(int cnt = 0; cnt < n; cnt++) arr[cnt]*=2; }

void Fn_Ptr(int *aPtr, int n)
{ for(int cnt = 0; cnt < n; cnt++)
    *aPtr++*=3; // эквивалентно *(aPtr++)*=3, поскольку разыменование и
// инкремент имеют одинаковый приоритет и ассоциативность справа
}

```

Будет напечатано:

```

1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
6 12 18 24 30 36 42 48 54 60

```

Поскольку имя массива является адресом его первого элемента, то при передаче его в функцию не требуется использование операции взятия адреса.

14.3. Строки как аргументы функций

Поскольку строки являются фактически массивами элементов символьного типа, то и передача их в функции происходит аналогично передаче массивов элементов других типов.

Пример 14.3

```

#include <iostream>
#include <stdlib.h>
#include <conio.h>
#include <iomanip>
using namespace std;
void Fn_Str(char *sPtr)
{ while (*sPtr) { if (*sPtr != ' ') *sPtr -=('a' - 'A');
    sPtr++;
}
}

int _tmain(int argc, _TCHAR* argv[])
{ char str[] = "this string get to the function";
cout << str << endl;
Fn_Str(str);
cout << str << endl;
_getch();
return 0;
}

```

Для перевода строки в верхний регистр нужно ASCII-код каждого символа уменьшить на фиксированное значение, которое является разницей в значениях данного символа в нижнем и верхнем регистрах.

Указателю sPtr передается копия адреса массива str. Значение *sPtr представляет собой изменяемый символ, если он не \0 и не пробел.

14.4. Передача структур по указателю и по ссылке

Функции могут принимать в качестве своих параметров не только указатели на стандартные типы, но и на структуры.

Пример 14.4. Передача структуры по указателю и по ссылке.

```

#include "stdafx.h"
#include <iostream>
#include <stdlib.h>
#include <conio.h>
#include <iomanip>
using namespace std;
typedef struct date
{ int y;           // год
  int m;           // месяц
  int d;           // день
} Date;           // псевдоним созданного структурного типа
// по указателю
void PrintDate( Date *dPtr)
{ cout << setw(4) << dPtr->y << '/';
  if (dPtr->m < 10) cout << '0';
  cout << setw(1) << dPtr->m << '/';
  if (dPtr->d < 10) cout << '0';
  cout << setw(1) << dPtr->d << endl;
}
int _tmain(int argc, _TCHAR* argv[])
{ Date birthday= {1980,1,28};
PrintDate(&birthday);
_getch();
return 0;
}

// по ссылке
void PrintDate( Date &dPtr)
{ cout << setw(4) << dPtr.y << '/';
  if (dPtr.m < 10) cout << '0';
  cout << setw(1) << dPtr.m << '/';
  if (dPtr.d < 10) cout << '0';
  cout << setw(1) << dPtr.d << endl;
}
int _tmain(int argc, _TCHAR* argv[])
{ Date birthday= {1980,1,28};
PrintDate(birthday);
_getch();
return 0;
}

```

Здесь параметр dPtr адресует структурную переменную объявленного типа. Поскольку структура адресуется указателем (ссылкой), доступ к полям структуры осуществляется оператором -> (оператором «точка»). Чтобы отобразить дату, программа объявляет и инициализирует структурную переменную birthday и передает ее адрес параметру функции PrintDate(). Компилятор C++ воспринимает выражение &birthday как адрес birthday, который передается параметру dPtr функции PrintDate().

Передача указателей в функцию экономит время и память ЭВМ, так как при этом не происходит побайтового копирования структуры в стек.

К переменным структурного типа можно также применить механизм передачи по ссылке.

Пример 14.5. Перевод размеров комнаты из метров в футы (передача параметров происходит по ссылке).

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;
struct room { double w; // ширина
    double l; // длина
    double h; // высота
};
double m_to_Ft (room &);
void print_room (room &);
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian"); //подключение русификатора
room big_room = {3.6, 4.8, 2.75};
room small_room = {2.7, 3.6, 2.75};
double big_S, small_S;
cout << "Размеры комнаты в метрах - " << endl;
print_room(big_room);
print_room(small_room);
big_S = m_to_Ft(big_room);
cout << "Площадь большой комнаты- " << big_S << " кв.футов\n";
small_S = m_to_Ft(small_room);
cout << "Площадь маленькой комнаты - " << small_S
    << " кв.футов\n";
cout << "Размеры комнаты в футах - " << endl;
print_room(big_room);
print_room(small_room);
_getch();
return 0;
}
double m_to_Ft (room &Room)
{ Room.w = Room.w * 3.28;
Room.l = Room.l * 3.28;
Room.h = Room.h * 3.28;
return Room.w * Room.l;
}
void print_room(room &Room)
{cout << Room.w << " x " << Room.l << " x " << Room.h << endl;
}
```

Результаты работы программы:

```
Размеры комнаты в метрах
3.6 x 4.8 x 2.75
2.7 x 3.6 x 2.75
Площадь большой комнаты - 185.905 кв.футов
Площадь маленькой комнаты - 104.572 кв.футов
Размеры комнаты в футах -
```

11.808 x 15.744 x 9.02
8.856 x 11.808 x 9.02

В процессе работы функции m_to_Ft() размеры комнаты, заданные в метрах, переводятся в футы и вычисляется площадь комнаты в квадратных футах. Значения, передаваемые по ссылке, изменяются в процессе работы функции, и эти изменения сохраняются после завершения работы функции.

14.5. Ссылка в качестве возвращаемого значения функции

При использовании ссылки в качестве возвращаемого значения функции появляется возможность использовать функции в качестве левого операнда в операции присваивания.

Пример 14.6. [12]. Возвращение значения по ссылке.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
int x; // глобальная переменная
int &setx() {return x;}
int _tmain(int argc, _TCHAR* argv[])
{ setx() = 92; // присваивание значения x
// при помощи функции, возвращающей ссылку
cout << "x = " << x << endl; // результат: x = 92
_getch();
return 0;
}
```

Здесь функция setx() возвращает ссылку на целое значение. Механизм этого процесса следующий. Возвращение функцией ссылки – это то же самое, что возвращение псевдонима переменной, стоящей в операторе return этой функции. В примере 14.6 это ссылка на глобальную переменную x.

Поскольку возвращаемое значение является псевдонимом некоторой переменной, то возвращать константу такая функция не может. Попытка написать

```
return 2;
```

вызывает ошибку трансляции.

То же самое происходит при возвращении локальной переменной, поскольку при выходе из функции локальные переменные уничтожаются, а ссылаться на несуществующие переменные нельзя.

Конечно, присвоить значение переменной x можно и менее экзотичным способом, но функции, возвращающие ссылки, нашли свое приме-

нение в объектно-ориентированном программировании при перегрузке операций.

14.6. Функции, возвращающие указатели

Функции могут не только принимать указатели в качестве аргументов, но и возвращать их. При этом принимающая переменная должна иметь тип указателя.

Пример 14.7. Возврат из функции указателя [19].

```
#include <stdafx.h>
#include <iostream>
#include <conio.h>
#include <ctype.h>
#include <string.h>
using namespace std;
//функцияUpperCase() возвращает указатель на тип char
char* UpperCase(char *s)
{ unsigned int i;
  for (i = 0; i < strlen(s); i++) s[i] = toupper(s[i]);
  return s; // возвращает адрес аргумента
}
int _tmain(int argc, _TCHAR* argv[])
{ char title[] = "Upper the Alphabet";
// Переведем буквы латинского алфавита в верхний регистр
  char *cPtr;
  cPtr = UpperCase(title);
  cout << cPtr << endl;
  getch();
  return 0;
}
```

При передаче указателей в качестве аргументов функции предполагается, что эти аргументы должны изменяться внутри функции и эти изменения следует передать в вызывающую программу. Если это не так, лучше передавать аргументы по значению или использовать указатели на константы. Это защитит программу от несанкционированного изменения данных функцией.

Пример 14.8. [19]. Описать функцию, которая захватывает память в куче и инициализирует ее некоторым значением.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;
const int SIZE = 25;
int *ResMem(unsigned long int size, int init);
int _tmain(int argc, _TCHAR* argv[])
{
```

```
{ int *iPtr;
  iPtr = ResMem(SIZE, 125);
  for (int i = 0; i < SIZE; i++) cout<<setw(5)<<iPtr[i];
  cout << endl;
  delete iPtr;
  _getch();
  return 0;
}
int *ResMem(unsigned long int size, int init)
{ int *tmpPtr;
  unsigned int i;
  tmpPtr = new int[size]; // захват памяти в куче
                           // под size целочисленных значений
  if (tmpPtr)
    { for (i = 0; i < size; i++) tmpPtr[i] = init; }
  return tmpPtr;
}
```

14.7. Константные ссылки и константные указатели в качестве параметров функций

При передаче аргумента по ссылке или по указателю обычно предполагается, что этот аргумент будет изменен функцией. Но ссылки и указатели могут использоваться и с целью оптимизации характеристик программы. Если аргумент передается по значению, в стеке создается копия этого аргумента. Например, при передаче аргумента, представляющего собой большую структуру, его размещение в стеке сопровождается неэкономным расходованием стекового пространства и ухудшением временных характеристик программы. В случае передачи аргумента по ссылке или по указателю в стек передается лишь адрес переменной, являющейся аргументом функции, что может существенно улучшить характеристики программы.

Но при этом весьма велика вероятность несанкционированного изменения аргумента в процессе работы функции. Во избежание этого используют константные ссылки, константные указатели и указатели на константы.

Рассмотрим два примера, демонстрирующих механизмы передачи в функции константных аргументов.

Пример 14.9. Использование константной ссылки.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
struct Date { int y; // год
              int m; // месяц
              int d; // день
}
```

```

};

void MyFunc(int &a, const Date &day);
int _tmain(int argc, _TCHAR* argv[])
{
    Date birthday = {1980, 3, 25}; // день рождения
    int old; // возраст
    MyFunc(old, birthday);
    _getch();
    return 0;
}

void MyFunc(int &a, const Date &day)
{
    a = 2014 - day.y; // корректные действия
    day.y = 1970; // ошибка трансляции:
    day.m = 12; // попытка изменить
    day.d = 17; // константный аргумент, т.е.
    // error C2166: левостороннее значение
    // указывает на объект-константу
}

```

Пример 14.10. Использование неконстантного указателя на константные данные.

```

#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
struct Date { int y; // год
               int m; // месяц
               int d; // день
};
void MyFunc(int *a, const Date *day);
int _tmain(int argc, _TCHAR* argv[])
{
    Date birthday = {1980, 3, 25}; // день рождения
    int old; // возраст
    MyFunc(&old, &birthday);
    _getch();
    return 0;
}

void MyFunc(int *a, const Date *day)
{
    *a = 2014 - day->y; // корректные действия
    day->y = 1970; // ошибка трансляции:
    day->m = 12; // попытка изменить
    day->d = 17; // константный аргумент, т.е.
    // error C2166: левостороннее значение
    // указывает на объект-константу
}

```

14.8. Указатели на функции

В C++ функции не являются переменными, но можно определить указатель на функцию и работать с ним как с обычной переменной: вы-

полнять операции присваивания, размещать в массиве, передавать в качестве параметра функции. Описание указателя на функцию должно соответствовать описанию самой функции: число и типы аргументов указателя должны совпадать с числом и типами аргументов функции.

Пример 14.11. Написать функцию поиска корня уравнения $x = F(x)$ с точностью ϵ методом простых итераций при заданном грубом значении корня. Используя написанную функцию, найти корни уравнений $x = x^2 - 7 + 1/x$ и $x = \sin(x) + \sqrt{x}$. Точность и грубое значение корня ввести с клавиатуры.

```

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <math.h>
#include <conio.h>
using namespace std;
// Описание заданных функций:
double fx1(double x)
{
    return x*x-7+1/x;
}
double fx2(double x)
{
    return x=sin(x)+sqrt(x);
}
double root(double (*fx)(double), double x_rough, double accuracy )
{
    double x, x0 = x_rough;
    x=(*fx)(x0);
    while (fabs(x - x0) > accuracy)
    {
        x0 = x;
        x = (*fx)(x0);
    }
    return x;
}
int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "Russian"); // подключение русификатора
    double xg, eps;
    double r1,r2;
    cout << "Для первого уравнения " << endl;
    cout << "введите грубое значение и точность ";
    cin >> xg >> eps;
    r1 = root(*fx1, xg, eps);
    cout << "Для второго уравнения " << endl;
    cout << "введите грубое значение и точность ";
    cin >> xg >> eps;
    r2=root(*fx2, xg, eps);
    cout << "Корень функции fx1 r1 равен " << r1 << endl;
    cout << "Корень функции fx2 r2 равен " << r2 << endl;
    _getch();
    return 0;
}

```

Указателем на функцию является адрес, по которому расположен код функции. Указатели на функции можно передавать другим функциям как параметры и получать в качестве возвращаемых функциями значений, а также присваивать другим указателям.

Контрольные вопросы

1. Дать определение ссылки.
2. В чем заключаются особенности передачи параметров функций по значению, по указателю и по ссылке?
3. Как защитить данные от несанкционированного изменения их функцией?
4. Почему следует особенно внимательно относиться к параметрам, передаваемым по ссылке?
5. Как реализовать возврат из функции более чем одного значения?

Практические задания 14

1. Написать функцию для вычисления длины окружности и площади круга заданного радиуса.
2. Написать функцию для вычисления объема и площади поверхности шара заданного радиуса.
3. Написать функцию, которая меняет значения переменных a , b , c местами так, чтобы оказалось $a \geq b \geq c$.
4. Определить функцию $\max_{-}\min$, принимающую 3 целых значения и возвращающую их максимальное и минимальное значения.
5. В данном вещественном векторе из 10 элементов поменять местами его минимальное значение и стоящее последним. Процесс обмена оформить в виде функции.
6. В данном целочисленном векторе из 20 элементов определить среднее арифметическое значение его элементов, сумму элементов, не превышающих полученного значения, и количество элементов, меньших среднего значения. Все полученные значения реализовать как возвращаемые значения одной функции.
7. Данна целочисленная матрица размера 6×9 . Определить функцию поиска значения и местоположения максимального и минимального элементов матрицы и передачи их в вызывающую программу.
8. Данна вещественная матрица размера 5×7 . Определить функцию поиска суммы положительных и произведения отрицательных элементов матрицы и передачи их в вызывающую программу.
9. Написать функцию вычисления суммы бесконечного ряда

$$S = \sum_{i=1}^{\infty} F(x),$$

задав $F(x)$ и точность в качестве параметров функции.

Используя написанную функцию, найти суммы рядов

10. Написать функцию, выполняющую поиск корня уравнения $F(x)=0$ методом деления отрезка пополам. Используя написанную функцию, найти корни уравнений $x = 3x^2 - 7 + 4/x$ и $x = \cos(x) + \text{sqr}(x)$.
11. Написать функцию вычисления по формуле Симпсона приближенного значения интеграла $\int_{p1}^{p2} F(x)dx$. Точность и пределы интегрирования задать как параметры функции.
12. Написать функцию вычисления методом прямоугольников приближенного значения интеграла $\int_{p1}^{p2} F(x)dx$. Точность и пределы интегрирования задать как параметры функции.