ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ

ВВЕДЕНИЕ В ЯЗЫК С

Часть 1

Учебно-методическое пособие

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНСТВО ПО ОБРАЗОВАНИЮ
Томский государственный университет
Факультет прикладиой математики и кибернетнки

"УТВЕРЖДАЮ"
Декан ФПМК
Профессор _____ А.М. Горцев
11 декабря 2009 г.

ВВЕДЕНИЕ В ЯЗЫК С

Часть 1

Учебно-методическое пособие

Томск
2010

РАССМОТРЕНО и УТВЕРЖДЕНО методической комиссией факультета прикладной математики и кибернетики

Председатель комиссии

профессор _____ С.Э. Воробейчиков

Протокол 34 от 11 декабря 2009 г

В данном учебно-методическом пособии кратко излагаются основы языка программирования С (с элементами языка С++), изучаемые в первом семестре курса "Информатика", читаемого на ФПМК ТГУ для специальности "Прикладная математика и информатика" В пособие вошли следующие темы: типы данных, функции ваода-вывода, операции и выражения, основные операторы языка С, функции, массивы (числовые, символьные, многомерные), указатели, работа с динамической памятью.

Составители:

старший преподаватель каф. программирования В.А. Сибирякова,

доцент каф. программирования О.И. Голубева

1. Структура программы на языке С

Программа нв языке С состонт нз:

- директив препроцессорв
- описаний
- функций

Среди функций выделяется глввная функция с именем *main*, с которой прогрвмма нвчинвет работу.

Пример простой программы:

```
// включение заголовочного файла стандартной библиотеки,
// содержащей функции ввода-вывода
#include <stdio.h>
// вычисление площади круга – это строчный комментарий
void main()                          // заголовок основной функции
{
  // описание переменных и инициализация числа пт
  float r, pi = 3 14159, s;
  // запрос на ввод радиуса круга
  printf("Введите радиус круга: ");
  // ввод радиуса, %f – спецификатор вещественного числа
  scanf("%f",&r),
  s = pi * r * r;                    // вычисление площади круга
  // вывод площади в формате. 7 позиций для
  // всего числа и 2 позиции для дробной части
  printf("s = %7.2f ", s);
}
```

В программе используется *строчный комментарий*: после знвков // вся последующая часть строки считается комментарием и компилятором игнорируется. В ограничителях /* */ задвется *фрагментарный комментарий*, который может стоять в любом месте строки или состоять из нескольких строк.

Например,

```
// определение перемвнной b может быть временно удалено
int a = 5, /* b = 7, */ z = 25;
// переменная n инициализирована числом 10 и
// пока авод не выполняется
int n = 10,
/* printf("\n Введите n: "); scanf("%d", &n); */
```

Вниманне! Язык С <u>различвет</u> большие и маленькие буквы!

2. Базовые типы данных

К основным данным в программе на языке С относятся:

1) *константы* – величины, заданные в программе своими значениями. В языке С определены следующие основные типы констант:

- *целые десятичные* 234, -37, 0;
- *целые восьмеричные* 0177, 0231;
- *целые шестнадцатеричные* 0x25, 0x1a;
- *вещественные* 99.89, -0.00125, 273, 2.5e-7;
- *символьные* '+', '!', '5', '\n' (перевод курсора на новую строку);
- *строковые* "a+b", "Введите n", "Hello World!", "!", ":)";
- *логические* имеют два возможных значения *true* и *false*.

Вещественная константа 2.5e-7 показывает пример записи вещественного числа в так называемой экспоненциальной форме, где "e" означает основание системы счисления 10. Значение числа $2.5e\text{-}7 = 2.5*10^{-7} = 0.00000025$. Эта форма записи используется для более краткой записи очень маленьких или очень больших чисел.

2) *Переменные* – величины, которые могут менять значения в процессе работы программы. Переменные задаются с помощью идентификатора.

Идентификатор – это последовательность букв, цифр и знака _, начинающаяся с буквы или знака подчеркивания

Примеры идентификаторов.

x, y1, arg1, z_1, z_2, value, radius

Каждая переменная до использования должна быть описана. Описание имеет формат:

тип идентификатор

Если несколько переменных имеют одинаковый тип, то их можно описать одним списком через запятую. При описании переменной ее можно инициализировать, т.е. присвоить начальное значение. Инициализация имеет вид:

тип идентификатор = значение

Значение задается константой соответствующего типа.

Основные типы переменных:

- *int* (*long int*) – переменная целого типа, занимающая 4 байта, положительная или отрицательная;
- *unsigned int* – переменная целого типа, занимающая 4 байта, только положительная (беззнаковое число);

- *short int* – короткая переменная целого типа, занимающая 2 байта,
- *unsigned short int* – беззнаковое целое число размером 2 байта,
- *float* – вещественная переменная длиной 4 байта;
- *double float* или просто *double* – вещественная переменная длиной 8 байтов;
- *char* – символьная переменная, звнимающая 1 байт и содержащая *ascii*-код символа;
- *unsigned char* – 1 байт, рассмвтриваемый как целое беззнаковое число;
- *bool* – логическая переменная, занимающая один байт, и принимающая только два значения *true* и *false*.

Примеры описания и инициализации переменных:

```
int a, b, s = 0, n = 10;
unsigned short int x = 65534,
float x, y, step = 0.01, pi = 3.14159, t = -25;
char s1= '+', s2 = '-', s3 = ' ', s4 = '\n';
bool f = true, c;
```

3. Функции ввода-вывода

Функция ввода

Форматированный ввод данных выполняется функцией *scanf* из стандартной библиотеки функций ввода-вывода (заголовочный файл библиотеки – *stdio.h*), имеющей формат:

***scanf*("*спецификатор_ввода...*",*список_адресов_вводимых_данных*)**

Спецификатор ввода – знак % с последующей буквой (*спецификатор типа*), определяющей *тип* вводимых данных.

Основные спецификаторы типов:

d – целое десятичное число,

f – вещественное число,

s – строкв символов,

c – один символ,

u – беззиаковое десятичное число,

x – число в 16-ой системе счисления,

o – число в восьмеричной системе счисления.

Адреса переменных задаются специальной операцией **&**.

Примеры ввода:

```
int a, b;
float x;
char s, r;
scanf("%d%d", &a, &b);  // вводятся целые числа в переменные a и b
scanf("%f", &x);        // вводится вещественное число x
scanf("%c%c", &s, &r);  // вводятся 2 символа в перемеиные s и r
```

Функция вывода

Форматированный вывод данных выполняется функцией ***printf*** (из той же библиотеки функций с заголовочным файлом stdio.h) формата:

***printf*("*комментарии_и_спецификаторы_вывода*",**
***список_вывода*)**

Комментарии и спецификаторы вывода определяют вид выводимой строки и формат выводимых значений Формат выводимых зиачений задается спецификатором вида. *%спецификатор_типа*. В функции *printf* используются те же спецификаторы типов, что и в функции *scanf*.

Список вывода может содержать: константы, переменные, выражения, вызовы функций.

Спецификаторов вывода должно быть столько же, сколько элементов в списке вывода Список вывода и спецификаторы вывода могут отсутствоввть. В этом случае оператор используется для вывода запросов и комментариев.

Примеры вывода.

```
int k = 10;
char s = '*';
float pi = 3.14159;
printf("\n n = %d", k);          // '\n' – вывод с новой строки
printf("\n пи = %f", pi);
printf("\n %c  %u", s, s);       // вывод символа '*' и его кода
```

Между символом **%** и спецификатором типа может стоять *спецификатор ширины* - целое число, определяющее количество позиций для вывода зиачения на экране. Если перед спецификатором ширины стоит знак '-', то значение данного при выводе будет располагаться слева в поле вывода.

Пример.

```
int x =100; printf("x = !%7d!", x);
```

Вид на экране.

```
x = !    100!

pnntf("x=!%-5d!", x);
```

Вид на экране:

```
x = !100  !
```

При выводе вещественных чисел, для указания числа выводимых символов после запятой, используется *спецификатор точности*, который ставится через точку после спецификатора ширины.

Пример.

```
float pi = 3.14159; printf("pi = %4 2f", pi),
```

Вид на экране

```
pi = 3.14
```

Для вывода переменных типа *short int* и *unsigned short int* перед спецификатором типа (d, u, o или x) добавляется буква h.

Пример.

```
short int x = 32767, printf("%hd", x);
```

4. Операции и выражения

Выражение строится из переменных, констант, знаков операций, обращений к функциям и круглых скобок ( ).

Порядок выполнения операций в выражении определяется приоритетами операций и круглыми скобками.

Знаки арифметических операций: +, -, *, /, %, ++, --.

Пример выражения:

((a * x + b) * x + c) * x – 5.

Особо следует пояснить операцию целочисленного деления '/'. Если оба операнда операции целочисленные, то результатом деления будет целая часть - частное. Для получения остатка от деления применяется операция '%'.

Например,

```
int a = 5, b = 8, c, d;
c = b/a; d = b%a;          // результаты: c = 1, d = 3
c = a/b; d = a%b,          // результаты: c = 0, d = 5
```

Обратите внимание, что даже если присваивание результата деления a/b будет выполнено вещественной переменной:

```
float r;
r = a/b;
```

то ответ будет не 0.625, а 0, так как сначала вычисляется значение выражения в правой части, а потом полученное значения присваивается переменной левой части.

В языке C++ введены 2 новые арифметические операции ++ (инкремент) и -- (декремент). Эти операции выполняют увеличение (++) или уменьшение (--) на 1 значения операнда.

Например,

```
int i = 0;
i++;        // i = 1, т.е. операция эквивалентна оператору i = i + 1
--i;        // i = 0, т.е. операция эквивалентна оператору i = i - 1
```

При использовании операций *инкремент* и *декремент* в выражениях различают *префиксную форму* операций и *постфиксную*. При использовании префиксной формы (++i, --i) сначала операция выполняется, и уже измененное значение переменной участвует в вычислении выражения. А при использовании постфиксной формы (i++, i--) исходное значение переменной участвует в вычислении выражении, а затем изменяется на 1.

Например,

```
int a = 2, b = 3, c, d;
c = ++a * b;                    // a = 3; c = 3 * 3 = 9
d = c++ - b;                    // d = 9 – 3 = 6; c = 10
```

Заметим, что при использовании операции ++ или -- как отдельного оператора, результаты i++ (i--) и ++i (i--) одинаковые.

В языке С определена *условная операция*. Она имеет вид:

условие ? выражение1 : выражение2

Если условие истинно, значением условной операции будет *выражение1*, если ложно – *выражение2*. Заметим, что условная операция должна определять значение при любом значении условия, другими словами *выражение2* не может отсутствовать. В частности, *выражение1* и *выражение2* также могут быть условными операциями. Значение условной операции должно либо присваиваться, либо быть частью другого оператора. Рассмотрим примеры

Пример 1. Условная операция находит max(a,b).

```
max = a > b ? a : b;
```

Пример 2. Вычисление y = sign(x).

```
y = x > 0 ? 1  x < 0 ? -1 : 0;
```

Замечание. В выражении могут использоваться вызовы функций из библиотеки математических функций. Для их использования необходимо подключить заголовочный файл библиотеки - *math.h*.

Например,

```
#include <math.h>
...
float x = 2, y;
y = sin(2 * x) + cos(x / 4);
```

В языке С также определены и другие операции: логические, поразрядные, операции сравнения и т.д., часть из которых мы рассмотрим ниже. Формат данного методического пособия не позволяет рассмотреть все операции языка С.

5. Основные типы операторов

Каждый оператор в языке С заканчивается знаком **точка с запятой.**

1. ***Оператор присваивания*** имеет вид:

переменная = выражение

Типы величин в левой и правой частях оператора могут быть произвольными. После вычисления выражения тип результата приводится к типу переменной в левой части.

Например,

```
int i, x, a = 5, h = 1, y, b = 3, c = 2;
i = 0;                          // правая часть - константа
x = a,                          // правая часть - переменная
// правая часть - выражение, значение которого вычисляется при
// старом значении x; полученный результат заиосится в x,
// заменяя старое значение на новое
x = x + h,
// правая часть – выражение
y = ((a * x + b) * x + c) * x – 5;
bool g; int z = 3;
g = (z > 4);                    // g = false
```

Как сказано аыше, в процессе присваиваиия выполняется *неявное приведение* полученного результата к типу переменной левой части. При этом возможны потери значений выражений.

Например, при вычислении площади круга

```
int t; float r = 2;
t = 2*3.14*r;                   // t = 12, а не 12.56
```

после вычисления аыражения в правой части результат приводится к целому типу путем **отбрасывания дробной части** (без округления).

В следующем примере преобразование выполняется при присваивании длинного числа короткому путем **отбрасывания старшнх двончных разрядов!**

```
long int x = 20000000;          // x16 = 1 31 2d 0016 – 4 байта
short int y;
// y = 1 31 2d 0016 = 2d 0016 = 11520₁₀ вместо 20 миллионов!
y = x;
```

При этом компилятор ие аыдает никаких сообщений.

Замечание 1. Для операторов присваивания вида:

x = x o y,

где o – арифметическая или поразрядная операция, в языке С существует сокращенная запись:

x o= y;

Например, оператор x += h равносилен оператору x = x + h.

Замечание 2.

Операция '=' может использоваться в правой части оператора присваивания для сохранения промежуточного значения при вычислении выражения.

Например,

```
// z и y получат соответствующие значения
y = (z = x + a) * (x - b);
```

Замечание 3. В языке С можно выполнять цепочки присваиваний. Присванвания в цепочке выполняются справа налево.

Например,

```
i = j = k = 0;
```

2. ***Условный оператор***

Условный оператор имеет 2 формы:

- полный условный оператор

  if **(*условие*)** ***оператор1*****;** ***else оператор2***

  если условие истинно, то выполняется *оператор1*, если – ложно, то *оператор2*.

- сокращенный условный оператор

  if **(*условие*)** ***оператор***

  если условие истинно, то выполняется *оператор*, если ложно, то выполняется следующий после оператора *if* оператор.

Условие обязательно записывается в круглых скобках и может быть *простым* или *сложным*.

Простое условие – это константы, переменные и выражения, связанные знаками отношений: <, <=, ==, !=, >=, >.

Например,

```
if (a > b) y = a - b; else y = b - a;
if (x < 0) x = -x;
if (x != 0) y = 1/x; else y = 0;
```

Замечание 1. В языке С все отличное от нуля – это истина!

Поэтому в последнем примере оператора *if* условие может быть записано короче:

```
if (x) y = 1/x; else y = 0;
```

Замечание 2. Запомните: **равенство задается операцией ==.** Один знак равно '=' определяет присваивание. Поэтому оператор:

```
if (x = 0) y = 1; else y = 2;
```

при **любом x** будет выполнять оператор y = 2, т.к. **x не сравнивается с нулем, а ему присваивается 0** и согласно замечанию 1 – условие всегда будет ложным!

Сложное условие строится из простых с использованием логических операций:

|| - логическое сложение, && - логическое умножение и ! – отрицание

Например,

```
// y = x * x + 1, если x ∈ [0, 2], иначе y = x * x * x.
if (x >= 0 && x <= 2) y = x * x + 1;
else y = x * x * x;
```

3. ***Составной оператор***

В условном операторе в случае истина и в случае ложь выполняется **один** оператор. В тех случаях, когда требуется выполнить не один оператор, а несколько, их объединяют в *составной* оператор, используя *операторные скобки* { }.

Для примера рассмотрим задачу: ввести 2 числа, упорядочить их по возрастанию и вывести.

Например, если введем a = 7, b = 5, то после работы программы a и b должны получить значение a = 5, b = 7; если будут введены значения a = 2 и b = 3, то значения переменных должны сохраниться.

```
#include <stdio.h>
void main()
{ int a, b, c;
  printf("\n Введите 2 числа: ");
  scanf("%d%d", &a, &b);
  if (a > b) { c = a, a = b; b = c; }
  printf("\n a = %d, b = %d", a, b);
}
```

4. *Операторы цикла*

В языке С имеются 3 оператора цикла:

1) Оператор цикла ***for***.

Оператор ***for*** имеет формат:

for (***инициализирующая_часть***; ***условие_работы_цикла***; ***действия***)

циклируемый_оператор

Первая строка называется *заголовком цикла*. Он состоит из трех частей. После первой и второй частей ставятся точки с запятой.

Алгоритм работы оператора:

1. Выполняется *инициализирующая_часть*, включающая действия присваивания начальных значений переменным, которые используются в условии и работе *циклируемого_оператора*;
2. вычисляется *условие_работы_цикла*;
3. если условие ложно, то выполняется переход на п. 7 алгоритма;
4. выполняется *циклируемый_оператор*;
5. выполняются *действия*, заданные в 3-ей части заголовка цикла;
6. переход на п. 2 алгоритма,
7. выход из оператора ***for***.

Заметим, что третья часть, как правило, содержит действия, изменяющие величины, входящие в условие; но это не обязательно, так как эти величины могут меняться и в самом *циклируемом_операторе*. В последнем случае третья часть заголовка может быть пустой. Пустыми могут быть и первая и/или вторая части заголовка. В любом случае в заголовке **обязательны две точки с запятой!**

Пример Рассмотрим следующую задачу, решаемую с использованием оператора ***for***. Требуется найти сумму чисел от 1 до 20.

```
#include <stdio.h>
void main()
{ int i, s,
  for (i = 1, s = 0; i <= 20; i++)
    s = s + i;
  printf("\n s = %d", s);
}
```

Если в цикле повторяется несколько операторов, то их также объединяют в составной оператор фигурными скобками {}.

В языке С не требуется, чтобы переменная цикла была целым числом. Рассмотрим, например, задачу вывода на экран таблицы значений функции $y = sin(x)$ в интервале $[a, b]$ с шагом h.

```
#include <stdio.h>
#include <math.h>
void main()
{ float a, b, h, x, y;
  printf("\n Введите интервал [a,b] и шаг h: ");
  scanf("%f%f%f", &a, &b, &h);
  for ( x = a, x <= b; x += h)
  { y = sin(x);
    printf("\n %7.2f  %7.2f ", x, y);
  }
}
```

2) Оператор цикла ***while.***

Оператор *while* имеет структуру:

***while* (*условие*)**

циклируемый_оператор

Алгоритм работы оператора *while* прост.

1. Вычисляется *условие*;
2. **если условие ложно, выполняется переход на п. 5;**
3. **выполняется** *циклируемый_оператор*;
4. переход на п. 1 алгоритма;
5. выход из оператора *while.*

Данный оператор цикла называют еще "*цикл while с предусловием*". Если условие сразу ложно, то *циклируемый_оператор* не выполняется ни разу.

Рассмотрим пример использования цикла *while* для решения следующей задачи. **необходимо найти наибольший общий делитель** двух чисел. **При решении задачи будем использовать метод:** из большего числа вычитаем меньшее, пока числа не станут равными.

```
#include <stdio.h>
void main()
{ int a, b;
  printf("\n Введите 2 натуральных числа: ");
  scanf("%d%d", &a, &b),
```

```
  while (a != b)
    if (a > b) a = a - b; else b = b - a;
  printf("\n нод = %d", a);
}
```

3) Оператор цикла ***do while***.

Оператор имеет структуру

do циклируемый_оператор
while (условие)

Алгоритм работы оператора:

1. Выполняется *циклируемый_оператор*;
2. вычисляется *условие*;
3. если условие **истинно**, то выполняется переход на п. 1 алгоритма;
4. выход из оператора *do while*.

Из алгоритма следует, что *циклируемый_оператор* выполняется обязательно хотя бы один раз, так как условие проверяется после его работы. Поэтому этот оператор цикла называется "*циклом while с постусловием*".

Например, требуется вычислить сумму степеней числа 2, не превосходящих 1000, т.е.

$s = 1 + 2 + 2^2 + 2^3 + \ldots$, пока $2^k <= 1000$.

```
#include <stdio.h>
void main()
{ int  s = 0, k = 1;
  do { s = s + k,
       k = k * 2;
     }
  while (k <= 1000);
}
```

Заметим, что по-прежнему, если в цикл входит более одного оператора, то их нужно объединить в один составной.

5. *Переключатель switch* и *оператор break*.

Переключатель в некоторых случаях может заменить несколько подряд расположенных операторов *if*, в условиях которых проверяются какие-то константные значения. **Например**, при вводе римской *цифры* требуется вывести ее арабское значение. Оператор *if* будет последовательно проверять значения цифры

```
char s;
scanf("%c", &s);
if (s == 'I') printf("\n – это 1");
if (s == 'V') printf("\n – это 5");
и т.д.
```

Оператор *switch* имеет формат:

```
switch (выражение)
{ case константа_1: операторы;
  case константа_2: операторы;
  ...
  case константа_n: операторы;
  default: операторы;
}
```

Фраза *default* необязательна. Выражение должно иметь значение только целого или символьного (не строкового!) типа. Вычисленное значение *выражения* последовательно сравнивается с константами после фраз *case*, поэтому константы должны быть того же типа, что и *выражение*. Выполняются *операторы* после того *case*, в котором *константа* совпала со значением *выражения*. Если значение *выражения* не совпало ни с одной константой, то выполняются операторы после фразы *default*, если она есть, или ничего не выполняется. Особенность оператора заключается в том, что после выполнения операторов найденного *case*, операторы всех нижерасположенных *case* также будут исполняться. Это так называемый *эффект проваливания*. Если это не подходит, т.е. требуется выполнить только операторы найденного *case*, то надо выйти из переключателя с помощью специального оператора ***break***. Оператор *break* выполняет выход из переключателя к оператору, следующему за ним.

Заметим, что оператор *break* также может использоваться для выхода из цикла до его окончания.

Рассмотрим 2 примера использования переключателя - **без эффекта** проваливания и с ним

Пример 1. Реализуем интерпретатор римских цифр.

```
#include <stdio.h>
void main()
{ char s; int k;
```

```
puts("\nРаботает интерпретатор римских цифр");
do
{ printf("\nВведите римскую цифру: ");
  scanf("%c", &s);
  switch (s)
  { case 'I': printf("\n – это 1"); break;
    case 'V': printf("\n – это 5"); break;
    case 'X': printf("\n – это 10"); break;
    case 'L': printf("\n – это 50"); break;
    case 'C': printf("\n – это 100"); break;
    case 'D': printf("\n – это 500"); break;
    case 'M': printf("\n – это 1000"); break;
    default . printf("\n Такой римской цифры нет!");
  }
  printf("\n Повторить? Да – введите 1, нет – введите 0");
  scanf("%d", &k);
} while (k);
}
```

Пример 2. Вводится номер текущего дня недели. Нужно вывести названия всех рабочих дней оставшихся до конца недели.

Решим задачу, используя эффект проваливания

```
# include <stdio.h>
void main()
{ int n;
  printf("Введите номер текущего дня недели: ");
  scanf("%d", &n);
  switch (n)
  { case 1: printf("Понедельник ");
    case 2: printf("Вторник ");
    case 3. printf("Среда ");
    case 4: printf("Четверг ");
    case 5: printf("Пятница "); break;
    default: printf("Это не рабочий день!");
  }
}
```

Так как оператора *break* нет в переключателе, то при совпадении значения переменной *n* с константой одного из *case* будет выведено название соответствующего *n* дня недели и далее всех последующих рабочих дней до конца недели.

6. Функции: определение, вызов, аргументы, возвращаемое значение

Одним из центральных понятий языка С является понятие *функции*. Профессионалы говорят: «язык С – язык функций». Понятие функции состоит из двух составляющих:

- определение функции,
- вызов функции

Определение функции имеет формат:

тип имя_функции(***список_формальных_аргументов***)
{ ***тело_функции*** }

Первая строка называется *заголовком* функции.

Рассмотрим его структуру.

Тип – это тип возвращаемого функцией значения, т.е тип результата, вычисленного в функции. Если функция не возвращает результат (например, просто выводит группу значений на экран), то используется специальный тип ***void***.

Имя_функции – любой идентификатор, не совпадающий с ключевыми словами языка.

Список_формальных_аргументов или часто говорят *параметров*, это перечень данных, которые передаются в функцию при её вызове, с указанием типа и произвольного идентификатора.

Формат задания параметров:

тип идентификатор

Особо отметим, что список параметров может быть пустым, но круглые скобки все равно остаются.

Тело_функции – группа описаний и операторов, выполняющих определенные действия. Переменные, описанные в теле функции, называются *локальными* и вне функции не существуют (недоступны). Если функция имеет возвращаемое значение, отличное от *void*, то среди операторов должен быть оператор:

return возвращаемое_значение

Типы значений, указанных в заголовке, и в операторе *return* должны совпадать или быть приводимыми по умолчанию. Например, если в заголовке указан тип возвращаемого значения *float*, а возвращается *int*, то это допустимо.

Оператор возврата значения *return* выполняет выход из функции в том месте, где он записан, поэтому он может использоваться и просто для выхода из функции в некотором месте. Если оператор

return не задан, то выход происходит после выполнения последнего оператора функции.

Рассмотрим примеры функций.

Пример 1. Функция вычисляет минимум из 3 целых чисел

```
int  Min3(int a, int b, int c)
{ // локальная переменная min – существует только в этой функции!
  int min = a;
  if (b < a) min = b;
  if (c < min) min = c;
  return min;
}
```

Пример 2. Функция вычисляет площадь круга радиуса *r*.

```
float sq_circle(float r)
{ return 3.14159 * r * r; }
```

Пример показывает, что в операторе return в качестве возвращаемого значения может быть задано выражение.

Пример 3. Функция выводит на экран 20 символов '*'.

```
void star20( )
{ int i;
  for (i = 0; i < 20; i++)
    printf("*");
}
```

Пример 4. Функция выводит на экран *k* заданных символов *s*.

```
void prints(int k, char s)
{ while (k--)
    printf("%c", s);
}
```

Пример 5. Вычисление функции sign(x).

```
int Sign(float x)
{ if (x > 0) return 1;
  if (x) return -1;
  return 0;
}
```

2 замечания к последнему примеру.

Замечание 1. Так как оператор *return* не только возвращает значение, но и выполняет выход из функции, то опция *else* излишняя

Звмечание 2. Так как ко второму опервтору *if* произойдет переход при двух возможных значениях x: $x < 0$ или $x == 0$, то условие x будет истинным при $x < 0$.

Рассмотрим теперь, каким образом можно использовать функции. Обращение к функции происходит через *вызов функции.*

Вызов функции имеет формат:

имя_функции(список_фактических_аргументов)

Список фактических аргументов – это константы, переменные, выражеиия. Соответствие между формальными и фактическнми аргументами выполняется по порядку их следования в том и другом списке (первый фактический аргумент соответствует первому формальиому и т.д.). Если список параметров в определении функции пуст, то **круглые скобки ( ) опускать нельзя**. Вызов функции может стоять там, где можно использовать выражения: в операторах вывода, прнсваивания, условных операторах, циклах, в спнсках фактических параметров при вызовах других функций и т.д.

Необходимое условие при вызове функции – функция должна быть описана ранее вызова функции. Поэтому, как правило, определения функций предшествуют главной функции Этот порядок может быть изменен, если задать перечень заголовков функций ранее вызова функций, а их определения затем разместить после функции main(). В этом случае списки аргументов могут содержать лишь типы аргументов без имен.

Напрнмер

```
int Min3(int, int, int);
float sq_cirle(float);
void star20( );
void prints(int, char);
int Sign(float),
```

Замечанне. В языке С не допускается вложенность функций!

Рассмотрим примеры использования (вызова) функций, определенных выше.

```
#include <stdio.h>
// здесь находятся определения функций, приведенные выше
void main()
{ int a = 4, b = 7, c = 1, i;
  float r, y;
  star20( );                    // вывод на экран 20 символов '*'
  printf("\n Введите радиус круга: ");
  scanf("%f",&r),
  printf(" Площадь круга радиуса %4.1f равна %7.2f", r, sq_circle(r));
                                // вызов функции в операторе вывода
  y = Sign(a);                  // вызов функции в операторе присваивания
  printf("\n sign(%d) = %d\n", a, y),
  // цикл выводит символы '-' в 20 строках по i знаков -
  // (получится треугольник)
  for (i = 0; i < 20; i++)
  { prints(i+1, '-'); printf("\n"); }
  // вычисляется площадь круга с вдвое меньшим радиусом,
  // т.о., фактическим аргументом может быть выражение
  y = sq_circle(r/2),
  star20();                     // и в заключение опять выводятся 20 '*'
}
```

7. Массивы данных

Массивы данных – это группы **однотипных** элементов, расположенные в оперативной памяти в последовательных ячейках. Рассмотрим отдельно *числовые массивы* и *символьные*.

7.1. Числовые массивы

Числовой массив имеет описание

***тип имя_массива*[*количество_элементов*]**

где тип – это тип элементов массива, имя – произвольный идентификатор. Количество элементов в массиве может быть задано только константой, но в пределах этой константы можно использовать произвольное количество элементов.

Например, описание

int a[10];

определяет целочисленный массив из 10 элементов.

Нумерация элемеитов массива начинается с 0.

Операторы программы работают с элементами массива. Элемент массива задается переменной с индексом:

***имя_массива*[*индекс*]**

Индекс может быть константой, переменной или выражением целого типв.

В нашем примере можно задать элементы массива таким образом:

a[0] – первый элемент массива с номером 0,

a[9] – последний элемент,

a[i] - *i*-ый элемент, причем значение *i* должно быть известно и быть >=0 и <10,

a[2*i + 1] – индекс элемента задан выражением,

a[a[0]] – индекс задан значением 0-го элемента этого же массива.

При описании массива значения его элементов не определены

Значения элементов массива могут быть введены или вычислены. Можно также весь массив или несколько элементов проинициализировать при описании массива.

Инициализация массива имеет вид.

***тип имя_массива*[*количество_элементов*] =**

{ *значения_элементов_массива_через_запятую* }

Например.

```
int a[5] = { 1, 2, 3, 4, 5 };
```

Если заданных значений меньше, чем размер массива, то остальные элементы заполняются нулями. Так при описании массиаа:

```
int b[10] = { 2, 3, -5 };
```

первые 3 элемента будут определены b[0] = 2, b[1] = 3, b[2] = -5, а остальные элементы от b[3] до b[9] будут заполнены нулями. Такое правило инициализации позволяет «обнулить» асе элементы любого массива простым способом

```
int c[100] = { 0 };          // все 100 элементов массива с будут
                             // проинициализированы нулевыми значениями
```

Пример. В качестве примера работы с массивом рассмотрим следующую задачу: нужно ввести массив целых чисел и «обернуть его», т.е. получить элементы а обратном порядке. Например, исходный массив: 3 5 7 9, результат: 9 7 5 3.

```
# include <stdio.h>
void main()
{ int a[20], n, i, j, b;
  printf("\nВведите количество элементов <=20: ");
  scanf("%d", &n);
  printf("\nВведите %d чисел: ", n);
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);              // числа вводятся через пробел или
                                     // Enter, но не через запятую
  // оборачиваем массив, меняя местами симметричные элементы
  // с начала массива и с конца
  for (i = 0, j = n - 1; i < j; i++, j--)
  { b = a[i], a[i] = a[j]; a[j] = b, }
  printf("\nРезультат: ");
  for (i = 0; i < n; i++)
    printf("%d ",a[i]);              // после спецификатора %d пишем
                                     // пробел, иначе числа «слипнутся»
}
```

Если параметром функции является массив, то он задается следующим образом:

тип имя_массива[ ]

т.е квадратные скобки остаются пустыми.

Функция «обращения» массива будет выглядеть так:

```
void reverse(int a[], int n)
{ int i, j, c;
  for (i = 0, j = n - 1; i < j; i++, j--)
  { c = a[i]; a[i] = a[j]; a[j] = c; }
}
```

Обращение к функции после ввода массива выполняется следующим образом:

```
reverse(a, n);
```

Обратите внимание, передается имя массива и количество элементов.

7.2. Символьные массивы – строки

Массив символов называют чаще *строкой*. Строка имеет следующее описание:

char имя_строки[количество_символов]

В оперативной памяти символы строки представляются кодами в последовательных байтах памяти и **ограничиваются байтом, состоящем из всех нулевых битов.** Этот байт называется *нуль-кодом* и имеет изображение '\0'. Нуль-код формируется автомвтически при вводе строки после нажатия *Enter*. Вывод строки осуществляется до нуль-кода. Нуль-код используется также при работе других функций над строками, о которых будет рассказано ниже.

Строку, как и числовой массив, можно проинициализировать при описании, но инициализвция, как правило, выполняется иначе, в имеино.

char имя_строки[количество_символов] = "значение"

Заметим, что в количество символов строки должеи включаться и нуль-код.

Например, при определеиии строки:

```
char str[25] = "Язык С – язык функций!";
```

для строки *str* будет выделеио 25 байт памяти и сформироваио 23 кода символа, включая нуль-код. Остальные 2 байта будут иеопределены

Ввод и вывод строки может выполняться теми же функциями форматного ввода-вывода *scanf* и *printf* со спецификатором %s. При вводе символы строки вводятся до *Enter*, причем в функции ввода

для указания адреса строки операцию & задавать не нужно, т к. имя строки – это и есть ее адрес

Для ввода и вывода строк в библиотеке ***stdio*** имеются также специальные функции.

gets(имя_строки) – вводит строку до *Enter*, в конец строки автоматически добавляется нуль-код.

puts(имя_строки) – выводит строку; символы выводятся до *нуль-кода*, после вывода курсор переводится на новую строку.

В стандартной библиотеке с заголовочным файлом *<string.h>* имеется группа функций для работы со строками. Рассмотрим некоторые из них.

strlen(имя_строки) – возвращает длину строки, т е. количество символов до нуль-кода;

strcpy(имя_строки1, имя_строки2) – копирует *строку2* в *строку1*, при этом старое содержимое *строки1* теряется.

Замечание. Для строк операция присваивания не определена, ее заменяет функция копирования.

Например,

```
char s1[15], s2[15] = "Информатика";
s1 = s2;                    // Неверно! Присваивание не определено!
```

Правильно применить функцию *strcpy*:

```
strcpy(s1, s2);
```

strcmp(имя_строки1, имя_строки2) – сравнение двух строк. Выполняется сравнение символов двух строк в одинаковых позициях до тех пор, пока символы равны. При появлении несовпадающих символов та строка считается больше другой, в которой не совпавший код символа больше кода символа другой строки. Функция возвращает 3 значения.

> 0, если первая строка больше второй,

< 0, если первая строка меньше второй,

0, если строки равны.

Такое сравнение строк называется *лексикографическим*.

Например, пусть заданы 2 строки

```
char s1[10] = "маша", s2[10] = "мама";
if (strcmp(s1,s2) > 0) puts("строка 1 > строки 2");
else puts("строка 1 <= строки 2"),
```

В примере сравнение строк выполняется до третьей буквы, и, поскольку код буквы 'ш' больше кода буквы 'м', функция вернет

положительное значение. В нашем примере будет выведено сообщение "строка 1 > строки 2".

strcat(имя_строки1, имя_строки2) – возвращается строка, являющаяся слиянием (***конкатенацией***) двух строк. Результат замещает первую строку.

Например, заданы 2 строки

```
char s1[10] = "тюль", s2[10] = "пан";
strcat(s1, s2);
puts(s1);                          // будет выведена строка "тюльпан"
```

7.3. Двумерные массивы – матрицы

Матрица или *двумерный массив* представляет собой таблицу, состоящую из строк и столбцов. Элементы матрицы имеют одинаковый тип. Описание матрицы имеет вид

тип_элементов

***имя_матрицы*[*количество_строк*][*количество_столбцов*]**

Например, описание

```
int a[3][4];
```

определяет матрицу из трех строк и четырех столбцов:

$$\begin{matrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \end{matrix}$$

Работа с матрицей выполняется поэлементно. Элемент матрицы задается следующим образом:

***имя_матрицы*[*номер_строки*][*номер_столбца*]**

Заметим, что нумерация строк и столбцов начинается с нуля.

При описании матрицы её элементы можно проинициализировать. Это делается следующим образом.

```
int a[3][4] = { { 1, 1, 1, 1 },          // первая строка с номером 0
                { 2, 2, 2, 2 },          // вторая строка с номером 1
                { 3, 3, 3, 3 }           // третья строка с номером 2
              };
```

Если количество значений инициализации меньше, чем размер матрицы, то остальные значения элементов полагаются равными 0.

Например, единичную матрицу размером 3*3 можно проинициализировать следующим образом:

```
int E[3][3] = { { 1 },
                { 0, 1 },
                { 0, 0, 1 } };
```

Если матрица не инициализируется при описании, то ее элементы содержат неопределенные значения.

Количество строк и столбцов определяется **константами,** но в пределах заданных размеров можно использовать произвольное количество строк и столбцов.

Рассмотрим пример. Требуется найти максимальный элемент главной диагонали квадратной матрицы размера ***m*m***.

```
#include <stdio.h>
void main()
{ int a[10][10], m, i, j, max;
  printf("\n Введите m <= 10: ");
  scanf("%d", &m);
  printf("\n Вводите матрицу построчно по %d элементов в строке:\n");
  for (i = 0, i < m; i++)
    for (j = 0; j < m; j++)
      scanf("%d", &a[i][j]);
  for (max = a[0][0], i = 1; i < m; i++)
    if (a[i][i] > max ) max = a[i][i];    // на главной диагонали индексы
                                          // строки и столбца равны
  printf("\n max = %d", max);
}
```

Матрицы символов также можно рассматривать как *массивы строк*. Представим, что мы задаем список фамилий. Одна фамилия – это строка, а весь список - это матрица символов фамилий. При работе со строками, в отличие от числовой матрицы, можно работать не только с отдельными символами, но и с целой строкой. В первом случае будет использоваться два индекса, во втором – один индекс. Кроме того, можно иначе выполнить инициализацию строк при описании массива строк. Инициализация на примере списка фамилий может иметь вид.

```
char fam[10][20] = { "Сергеев",    // максимально можно задать
                     "Иванов",     // 10 фамилий длиной
                     "Сидоров",    // не более 19 букв, 1 символ
                     "Андреев",    // строки отводится для нуль-кода
                     "Ивлев" };
// остальные 5 фамилий не проинициализированы
```

Пример. Для примера рассмотрим такую задачу. Задан список фамилий. Вывести фамилии на заданную букву.

```
#include <stdio.h>
void main( )
{ char fam[10][20] = { "Сергеев",
                      "Иванов",
                      "Сидоров",
                      "Андреев",
                      "Ивлев" }, b;
  int i, n = 5;
  puts("Задан список фамилий: "),
  for (i = 0; i < n; i++)
    puts(fam[i]);                    // обращаемся к каждой строке целиком
  puts("Введитв букву: "); scanf("%c",&b);
  printf("\nФамилии на букву %c: \n", b);
  for (i = 0; i < n, i++)
  // обращаемся к первой букве фамипии как к элементу матрицы
    if (fam[i][0] == b) puts(fam[i]);
}
```

Замечание. В том случае, когда аргументом функции является матрица, он задается таким образом.

***тип имя_матрицы*[][*количество_столбцов*]**

Первые квадратные скобки пустые, а во вторых скобках должно быть задано то же количество столбцов, что и у фактического аргумента-матрицы.

Пример, рассмотрим функцию вычисления произведения иеиулевых элементов матрицы.

```
int mult(int a[][4], int m, int n)
{ int i, j, p = 1;
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      if (a[i][j]) p = p * a[i][j],
  return p;
}
void main()
{ int a[3][4] ={ { 2, 3 },              // a[0][2] = a[0][3] = 0
                 { 0, 1, 4, 8 },
                 { 0, 2, -4 } };        // в[2][3] = 0
  printf("\nПроизведение не равных 0 элементов = %d", mult(a,3,4));
}
```

8. Указатели. Операции * и &

Всякий элемент данных, расположенный в оперативной памяти, характеризуется двумя величинами: *значением* и *адресом*, по которому он находится. Чаще всего, используя имена данных, мы оперируем с их значениями, но во всех современных языках имеется возможность работать с адресами данных. В языке С для этого используется тип данных *указатель*. Значениями переменных, описанных как указатели, являются адреса данных. Через указатели можно обращаться как к самим данным, так и оперировать с адресами. В последнем случае говорят об *адресной арифметике*.

Указатель определяется описанием

тип * идентификатор

Например, описание

```
int *y;
```

говорит о том, что переменная *y* будет содержать *адреса данных целого типа*

Предположим, что имеются также описания переменных.

```
int a = 5, b = 7;
```

Адреса переменных можно присваивать переменной-указателю *y*. Для этого используется специальная операция **&** - *получить адрес*.

Если предположить, что переменная *b* расположена в ячейке с адресом 104, то после выполнения операции

```
y = &b;
```

в переменную *y* запишется число 104 - адрес переменной *b*, а не её значение 7.

Возникает необходимость и в обратном действии. Как, зная, что в некоторой переменной-указателе находится адрес некоторого значения, использовать это значение в операциях? Для этого предназначена специальная операция * - *получить значение по адресу* или операция *разадресации*.

В нашем примере сложить переменную *a* со значением ячейки с адресом в переменной *y* (*y* содержит адрес переменной *b*) можно оператором:

```
a = a + *y;                    // a = 5 + 7
```

Рассмотрим описание некоторого массива:

```
int a[5],
```

В оперативной памяти элементы массива располагаются в последовательных ячейках, а адрес первого элемента (с номером 0) присваивается переменной *a* – имени массива. Значение (адрес)

переменной *a* присваивается на этапе компиляции при распределении памяти для данных программы и при работе программы его изменить нельзя. Говорят, что имя массива *a – это константный указатель* Используя переменную-указатель можно «перемещаться по массиву», используя адресную арифметику.

Рассмотрим несколько примеров Итак, пусть имеются описания:

```
int a[5] = { 1, 2, 3, 4 }, b = 10, c, *y,
y = a;                    // переменная y получила адрес элемента a[0]
c = *y + b;               // c = a[0] + b = 1 + 10 = 11
y++;                      // переменная y получила адрес элемента a[1]
*(y + 3) = *y  - *(y + 1); // a[4] = a[1] – a[2] = 2 - 3 = -1
(*y)++;                   // a[1]++ = 2 + 1 = 3, т.к
                          // y по-прежнему содержит адрес a[1]
*y++;                     // значение a[1] не изменится,
                          // но переменная y получит адрес a[2]
c = y – a;        // c = 2 - разность адресов a[2] и a[0] деленная на 4,
                  // т.е. разность номеров 2-го и 0-го элементов
```

Рассмотрев эти примеры, следует уточнить понятие адресной арифметики. При увеличении или уменьшении переменной-указателя на единицу, фактически адрес изменяется компилятором на размер в байтах единицы данных в зависимости от типа указателя. В нашем примере указатель *y* определен как указатель на целое (*int*), поэтому значение указателя будет меняться на 4

Пример. Рассмотрим пример использования указателя для решения задачи является ли заданное слово *палиндромом*

```
#include <stdio.h>
#include <string.h>
void main( )
{ char word[10] = "казак", *y1 = word, *y2;
  y2 = y1 + strlen(word) - 1;                  // y2 – адрес последней буквы
  // пока адреса не «встретились» и буквы равны!
  while (y1 < y2 && *y1 == *y2 )
    { y1++; y2–; }                             // меняем адреса
  if (y1 >= y2)
    printf("\nДа, слово - палиндром"),         // y1 и y2  встретились
  else printf("\nнет, слово - не палиндром");
}
```

Заметим, что поскольку адреса – это числа, то их можно сравнивать обычными операциями отношений

9. Динамические массивы и матрицы. Операции *new*, *delete*

Недостатком рассмотренного в п. 7 способа представления массивов и матриц является то, что их размеры определяется константами. Фактический размер может оказаться значительно меньше, и поэтому часть оперативной памяти не будет использоваться, но будет считаться занятой. В языке С имеется возможность для массивов и матриц использовать тот объем оперативной памяти, который требуется в данном варианте работы программы. Для каждой программы выделяется память, в которой размещается сама программа и ее данные. Остается также свободная, незанятая память. Операцией ***new*** можно запросить требуемый объем свободной памяти во время выполнения программы. Если свободная память есть в наличии, то она выделяется и возвращается адрес первого байта выделенной памяти.

Формат операции:

new тип[количество_единиц_памяти_данного_типа]

Так, при выполнении операции

```
new int [10];
```

будет выделено 10 * 4 = 40 байт памяти, а операции

```
new char[10];
```

10*1 = 10 байт памяти

В этих примерах выделенная память никак не использовалась. На практике операция *new* может быть использована для запроса свободной памяти конкретного размера под массивы и матрицы. Рассмотрим, как это делается.

Массивы

Пусть требуется определить массив из *n* элементов, причем *n* вводится при выполнении программы. Для этого определим переменную-указатель, которая будет содержать адрес начала массива, т.е. элемента с номером 0.

```
int *a, n;                  // пока значение указателя a неопределенно
printf("\nВведите n: ");
scanf("%d", &n);            // вводим размер массива
a = new int[n],
```

Запрашиваем в свободной памяти n*4 байта. Если памяти требуемого размера достаточно, то она выделяется, и возвращается адрес первого байта выделенной памяти. Этот адрес записывается в

переменную-указатель *a*. Если саободной памяти требуемого размера нет, то аозвращается *нуль*. Поэтому для корректной работы программ требуется проверять результат работы операции *new*. Например, следующим образом,

```
if (!a) { printf("Недостаточно памяти! Выход"); return; }
```

Если операция *new* завершит работу нормально, то далее с массивом можно работать как обычно: можно задавать элемент массива индексом *a*[*i*], а можно, используя адресную арифметику, *(*a* + *i*), где (*a* + *i*) – адрес *i*-го элемента массива *a*, *(*a* + *i*) – его значение.

Операция ***delete*** используется для того, чтобы освободить память, которая была взята в программе с использованием операции *new*. Освободить память можно в любом месте программы, если для дальнейшей работы массив не нужен.

Формат операции:

delete [] ***a***

здесь пустые квадратные скобки '[]' информируют о том, что освобождается память не от одного элемента по адресу *a*, а от массива данных. Если память бралась только для одного элемента, то формат освобождения:

delete a

При выполнении операции ***delete*** память, занятая массивом, присоединяется к списку свободной памяти, связь указателя *a* с ней разрывается, и указатель *a* далее использовать нельзя, до тех пор, пока он снова не будет связан с каким-либо участком памяти

Матрицы

Матрица в языке С рассматривается как *массив указателей на строки*. Поэтому и описание матрицы имеет такой вид

```
a[m][n],
```

который говорит о том, что определены *m* указателей *a*[*i*] на строки, т.е. *a*[*i*] соответствует адресу начала *i*-ой строки. А каждая строка, в нашем примере, состоит из *n* элементов. Это позволяет с отдельной строкой оперировать просто, как с массивом.

Рассмотрим для иллюстрации сказанного такую задачу. Определена функция нахождения максимального элемента в массиве *a* размерности *n*.

```
int Max(int a[], int n)
{ int i, max = a[0];
  for (i = 1; i < n; i++)
```

```
    if (a[i] > max ) max = a[i];
  return max;
}
```

Эту функцию можно использовать для нахождения максимального элемента строки матрицы, передавая в функцию адрес строки и количество элементов в ней.

```
void main()
{ int a[3][4] = { { 1, 2, 3, 4 },
                  { 5, 7, 3 , 2 },
                  { 0, 7 } }, i,                    // a[2][2] = a[2][3] = 0
  puts("\n Максимумы строк матрицы: ");
  for (i = 0; i < 3; i++)
   printf("\n %d   %d", (i + 1), Max(a[i], 4));
                        // передаем в функцию адрес i-ой строки a[i] и
                        // количество элементов в ней
}
```

Рассматривая такой способ представления матрицы, место для матрицы можно запрашивать в свободной памяти после ввода размерностей. Это делается в 2 этапа:

1) Запрашивается память под массив указателей на строки:

```
a = new int *[m];              // m – количество строк
```

Обратите внимание, что запрашивается память не под числа, а под адреса, поэтому тип - **int***.

Операция *new* возвращает адрес начала массива адресов, поэтому переменная *a* будет иметь тип ***двойной указатель***:

```
int **a
```

После нормального завершения операции *new* будет выделена память под массив адресов и только. Память под элементы строк не будет выделена, то есть элементы массива указателей *a* будут неопределены. Память под элементы строк выделяются на втором этапе.

2) В цикле для каждой строки матрицы запрашивается память под массив элементов строки:

```
for (i = 0; i < n; i++)
    a[i] = new int[n];          // n – количество элементов в строке
```

Далее с элементами матрицы работаем как обычно, используя 2 индекса

Освободить память от матрицы, используя операцию *delete*, можно также в 2 этапа, которые обратны процедуре запроса памяти операцией *new*:

1. Сначала надо освободить память, занятую каждой строкой матрицы:

```
for (i = 0; i < m; i++)
  delete [] a[i];
```

2. Затем освобождается память от массива указателей на строки

```
delete [] a;
```

Замечание 1. Так как размеры массивов и матриц при таком представлении могут меняться от варианта к варианту работы программы, то такие массивы и матрицы часто называют *динамическими*, а свободную память - *динамической*.

Замечание 2. При динамическом представлении массивов и матриц формальные аргументы функций, являющиеся массивами или матрицами, могут быть заданы через указатели.

Формальный аргумент – динамический массив задается следующим образом:

```
int ff(int *a, int n)          // a – указатель на массив,
                               // n – размерность массива
```

Формальный аргумент – динамическая матрица определяется следующим образом:

```
int ff( int **a, int m, int n)  // a – указатель на матрицу,
                                // m, n – число строк и столбцов
```

Пример. Рассмотрим решение той же задачи о нахождении максимальных элементов строк.

```
int Max(int *a, int n)
{ int i, max = a[0];
  for (i = 1; i < n; i++)
    if (a[i] > max ) max = a[i];
  retum max,
}

void main()
{ int **a, m, n, i, j;
  printf("\nВведите m и n: ");
  scanf("%d%d", &m, &n);
```

```
  a = new int *[m];
  if (!a) { printf("\n Недостаточно памяти"), return; }
  for (i = 0; i<m; i++)
  { a[i] = new int[n];
    if (!a[i]) { printf("\n Недостаточно памяти"); return; }
  }
  printf("\nВводите элементы матрицы построчно\n");
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &a[i][j]);
  puts("\nМаксимумы строк матрицы: ");
  for (i = 0; i < m; i++)
    printf("\n %d . %d", (i + 1), Max(a[i], n));   // передаем в функцию адрес
                                                   // i-ой строки и n – количество
                                                   // элементов в ней.
  for (i = 0; i < m; i++)
    delete [] a[i];
  delete [] a;
}
```

Рассмотрим еще две задачи на тему динамические матрицы.

Пример. Удалим из матрицы нулевые строки.

```
#include <stdio.h>
void main()
{ int **a, m, n, i, j, k;
  printf("\nВведите m и n: ");
  scanf("%d%d", &m, &n);
  a = new int *[m];
  if (!a) { printf("\n Недостаточно памяти"); return; }
  for (i = 0; i<m; i++)
  { a[i] = new int[n];
    if (!a[i]) { printf("\n Недостаточно памяти"); return, }
  }
  printf("\nВводите элементы матрицы построчно\n");
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &a[i][j]);
```

```
  for (i = 0; i < m;)
  {
    j = 0; while (j < n && !a[i][j]) j++;
    if (j == n)                          // если строка состоит только из нулей,
    {
      delete [] a[i]; m--;                          // освобождаем занятую этой
      for (k = i; k < m; k++) a[k] = a[k+1];        // строкой память и сдвигаем
                                                    // на одну позицию вверх
                                                    // указатели на строки,
                                                    // расположенные ниже
    }
    else i++;
  }
  puts("\nМатрица без нулевых строк: ");
  for (i = 0; i < m; i++)
  {
    for (j = 0; j < n; j++)
      printf("%d ", a[i][j]);
    printf("\n");
  }

  for (i = 0; i < m; i++)
    delete [] a[i];
  delete [] a,
}
```

Пример. Создадим матрицу b из строк матрицы a, не содержащих нулей.

```
#include <stdio.h>
void main()
{ int **a, m, n, i, j, k, **b,
  printf("\nВведите m и n: ");
  scanf("%d%d", &m, &n);
  a = new int *[m];
  b = new int *[m];
  if (!a || !b) { printf("\n Недостаточно памяти"); return, }

  for (i = 0; i<m; i++)
  { a[i] = new int[n];
```

```
    if (!a[i]) { printf("\n Недостаточно памяти"), return, }
  }
  printf("\nВводите элементы матрицы построчно\n"),
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &a[i][j]);
  // создаем матрицу b из строк матрицы a, не содержащих нулей
  for (i = 0, k = 0; i < m; i++)
  {
    j = 0; while (j < n && a[i][j]) j++;
    if (j == n)                  // если в строке нет нулей,
      b[k++] = a[i];             // то указатель на эту строку присваиваем
                                 // очередному элементу массива указателей
                                 // на строки матрицы b
  }
  puts("\nМатрица b:\n");
  for (i = 0; i < k; i++)
  {
    for (j = 0; j < n; j++)
      printf("%d ", b[i][j]);
    printf("\n");
  }

  for (i = 0; i < m; i++)
    delete [] a[i];
  delete [] a;
  delete [] b;
}
```

Литература

1. В В. Подбельский. Язык Си++ - М.: Финансы и статистика, 2000. – 560 с.
2. Т.А. Павловская. C/C++ Программирование на языке высокого уровня – СПб.: Лидер, 2010. – 461 с.
3. М. Уэйт, С. Прата, Д. Мартин Язык Си – М: Мир, 1988. – 512 с.

Содержание

Отпечатано на участке оперативной полиграфии
редакционно-издательского отдела ТГУ

Заказ №53 от «16» 04 2010 г. Тираж 150 экз.