Язык C

2020

Представление данных в языке C

1

Имена данных

- С помощью имен обозначаются константы, переменные, функции, массивы.
- Имя (идентификатор) может содержать буквы, цифры, символы подчеркивания. Имя должно начинаться с буквы или символа подчеркивания. Строчные и прописные буквы в имени различаются.

  a2, max, Max, sort, kol_it, name, str, array.
- Нельзя использовать служебные слова языка:

  auto double int struct break else long switch register tupedef char extern return void case float unsigned default for signed union do if sizeof volatile continue enum short while

Переменные и константы

- Все переменные до их использования должны быть определены (объявлены). При этом задается тип, а затем идет список из одной или более переменных этого типа, разделенных запятыми:

```
int a, b, c;
char x, y;
```

- Переменные могут быть инициализированы при их определении:

```
int a = 25, h = 6;
char g = 'Q', k = 'm';
float r = 1.89;
long double n = r * 123;
```

Глобальные и локальные объекты

- В языке возможны глобальные и локальные объекты. Первые определяются вне функций и доступны для любой из них. Локальные объекты по отношению к функциям являются внутренними. Они начинают существовать, при входе в функцию и уничтожаются после выхода из нее.

```
int a;                                  //Глобальная  переменной

// Объявление функции (т.е. описание ее заголовка)
int function(int b, char c);

void main(void)
{                                     //Тело программы
   int d, e;                           //Локальные переменные
   float f;
}

int function(int b, char c) // Определение функции
{
     char g;    //Определение локальной переменной
}
```

Форматированный вывод данных

- **printf(форматная_строка, список_аргументов);**

```
printf("Привет мир");
printf("num = %i\n",5);
```

- Управляющие символы влияют на расположение на экране выводимых знаков

\b - для перевода курсора влево на одну позицию;
\n - для перехода на новую строку;
\r - для возврата каретки;
\t - горизонтальная табуляция;
\v - вертикальная табуляция;
**** - вывод символа \;
\' - вывод символа ' ;
\" - вывод символа ";

Форматированный вывод данных

%c – подстановка **char**

%d или **%i** – подстановка **int**
%o – то же в восьмеричной системе
%x – то же в шестнадцатеричной системе

%ld – подстановка **long**
%u – подстановка **unsigned**

%f, **%lf**– подстановка **float**, **double** в обычной форме
%e, **%le** – то же в экспоненциальной форме

%s – подстановка строки (**char***)

%p – подстановка значения указателя
%% – вывод символа %

Форматированный ввод данных

- **scanf(форматная_строка, список_аргументов);**

```
float a, b, c;
scanf("%f%f%f", &a, &b, &c);
scanf("%f", &a);
```

- Перед именем каждой переменной ставится значок **&** - «взятие адреса переменной»

Пример

```
#include <stdio.h>
#include <math.h>

void main() {
	float a, b, c, p, s;

	printf("\na="); scanf("%f", &a);
	printf("\nb="); scanf("%f", &b);
	printf("\nc="); scanf("%f", &c);

	p = (a + b + c) / 2;
	s = sqrt(p * (p - a) * (p - b) * (p - c));

	printf("\nПлощадь треугольника=%f", s);
}
```

Арифметические операции

+ сложение
– вычитание
*** умножение**
/ деление

```
float b;
b = 45.0 / 9.0;
b = 3 / 2;

int a = 3;
b = (float)a / 2;
```

% получение остатка от деления нацело - применяется только к данным целого типа.

```
int a = 3;
b = a % 2;
```

Приоритет и порядок вычислений

Операция	Порядок вычислений
( )	Слева направо
- (унарный)	Справа налево
* /	Слева направо
+ -	Слева направо
= (присваивания)	Справа налево

a * b – c /d * f + g +h

a * b – (c / (d * f) + g)+h

Оператор присваивания

- Допускается множественное присваивание, например:

```
a = b = c = d = 1;
```

- Операторы присваивания, в которых и в левой и в правой частях встречается одно и то же имя переменной:

```
a = a + b;
d = d * m;
x = x / y;
```

- могут быть записаны короче:

```
a += b;
d *= m;
x /= y;
```

Операции увеличения и уменьшения

- Предусмотрены две необычные операции для увеличения и уменьшения значений переменных. Операция увеличения **++** (инкремент) добавляет 1 к своему операнду, а операция уменьшения **--(**декремент) вычитает 1.
- Можно использовать либо как префиксные операции (перед переменной: ++n), либо как постфиксные (после переменной: n++).

Если **n = 5**, то

x = n++;

устанавливает **x** равным **5**, а

x = ++n;

полагает **x** равным **6**. В обоих случаях **n** становится равным **6**.

x = i+++j;

Математические библиотечные функции

Функция	Назначение функции
sin(x)	синус x
cos(x)	косинус x
tan(x)	тангенс x
asin(x)	арксинус x
acos(x)	арккосинус x
atan(x)	арктангес x
exp(x)	экспоненциальная функция e в степени x
log(x)	натуральный логарифм ln(x), x>0
log10(x)	десятичный логарифм lg(x), x>0
pow(x,y)	x в степени y
sqrt(x)	квадратный корень от x
abs(n)	абсолютное значение целого аргумента n
fabs(x)	абсолютное значение аргумента x типа double

- x – имеет тип double

Пример

```c
#include <stdio.h>
//Подсчитать количество банкнот в 50, 10 рублей и количество монет в 5 и 1 рубль для выдачи сдачи с введенной суммы
void main(void)
{
    int summa, r50, o50, r10, o10, r5, o5, r1;

    puts("Введите необходимую сумму сдачи");
    scanf("%i", &summa);

    r50 = summa / 50; //результат деления суммы на 50
    o50 = summa % 50; // остаток деления суммы на 50
    r10 = o50 / 10;   //результат деления на 10 остатка от деления на 50
    o10 = o50 % 10;
    r5 = o10 / 5;
    o5 = o10 % 5;
    r1 = o5;

    printf("Чтобы дать %i рублей сдачи, используйте:\n", summa);
    printf("%i банкноты достоинством 50 рублей\n", r50);
    printf("%i банкноты достоинством 10 рублей\n", r10);
    printf("%i монеты достоинством 5 рублей\n", r5);
    printf("%i монеты достоинством 1 рублей\n", r1);
}
```

УСЛОВНЫЕ ОПЕРАТОРЫ

2

Дерективы препроцессора

- Почти все программы на языке С используют специальные команды для компилятора, которые называются директивами. В общем случае директива – это указание компилятору языка С выполнить то или иное действие в момент компиляции программы. Существует строго определенный набор возможных директив, который включает в себя следующие определения:

  #define, #elif, #else, #endif, #if, #ifdef, #ifndef, #include, #undef.

- Директива **#define** используется для задания констант, ключевых слов, операторов и выражений, используемых в программе.

  #define <*идентификатор*> <*текст*>

- Символ **;** после директив не ставится.

Примеры использования директивы **#define**.

```c
#include <stdio.h>

#define TWO 2
#define FOUR TWO*TWO
#define PX printf("X равно %d.\n", x)
#define FMT "X равно %d.\n"
#define SQUARE(X) X*X

int main()
{
      int x = TWO;
      PX;
      x = FOUR;
      printf(FMT, x);
      x = SQUARE(3);
      PX;
      return 0;
}
```

Дерективы препроцессора

- Директива #undef отменяет определение, введенное ранее директивой #define.

  #undef FOUR

- Для того чтобы иметь возможность выполнять условную компиляцию, используется группа директив #if, #ifdef, #ifndef, #elif, #else и #endif.

```
#ifdef GRAPH
      #include <graphics.h> //подключение графической библиотеки
#elseif TEXT
      #include <conio.h> //подключение текстовой библиотеки
#else
      #include <io.h> //подключение библиотеки ввода-вывода
#endif
```

Деректива **#include**

- `#include <stdio.h>`
- `#include "C:/Users/dex/Documents/1.txt"`

Условный оператор **if**

- **if (***выражение***)**
  <*оператор***>**

if(a < b) Истинно, если **a** меньше **b** и ложно в противном случае.

if(a > b) Истинно, если **a** больше **b** и ложно в противном случае.

if(a == b) Истинно, если **a** равна **b** и ложно в противном случае.

if(a <= b) Истинно, если **a** меньше либо равна **b** и ложно в противном случае.

if(a >= b) Истинно, если **a** больше либо равна **b** и ложно в противном случае.

if(a != b) Истинно, если **a** не равна **b** и ложно в противном случае.

if(a) Истинно, если **a** не равна нулю, и ложно в противном случае.

Условный оператор if

```
#include <stdio.h>
int main()
{
   float x;
   printf("Введите число: ");
   scanf("%f", &x);

   if (x < 0)
      printf("Введенное число %f является
      отрицательным.\n", x);
   if (x >= 0)
      printf("Введенное число %f является
      неотрицательным.\n", x);

return 0;
}
```

Условный оператор if

- Два условных оператора можно заменить одним, используя конструкцию

```
if (выражение)
        <оператор1>;
else
        <оператор2>;
```

Если «выражение» истинно, то выполняется «оператор1», иначе выполняется «оператор2». В случаях, когда при выполнении какого-либо условия необходимо записать более одного оператора, необходимо использовать фигурные скобки.

```
if (выражение)
{
        <список операторов>
}
else
{
        <список операторов>
}
```

Условный оператор if

- После ключевого слова **else** формально можно поставить еще один оператор условия **if**, в результате получим еще более гибкую конструкцию условных переходов:

```
if(выражение1)       <оператор1>;
else if(выражение2)  <опреатор2>;
else                 <оператор3>;
```

- Имеются три логических операции:

&& - логическое И
|| - логическое ИЛИ
! – логическое НЕТ

- Самый высокий приоритет имеет операция **НЕТ**. Более низкий приоритет у операции **И**, и наконец самый малый приоритет у операции **ИЛИ**

Условный оператор if

if(exp1 > exp2 && exp2 < exp3) Истинно, если значение переменной exp1 больше значения переменной exp2 и значение переменной exp2 меньше значения переменной exp3.

if(exp1 <= exp2 || exp1 >= exp3) Истинно, если значение переменной exp1 меньше либо равно значения переменной exp2 или значение переменной exp2 больше либо равно значения переменной exp3.

if(exp1 && exp2 && !exp3) Истинно, если истинное значение exp1 и истинно значение exp2 и ложно значение exp3.

if(!exp1 || !exp2 && exp3) Истинно, если ложно значение exp1 или ложно значение exp2 и истинно значение exp3.

Условный оператор if

```
#include <stdio.h>
int main()
{
	char c;
	printf("Введите одиночный символ:\n");
	scanf("%c", &c);

	if ((c >= ' a ' && c <= 'z') || (c >= 'A' && c <= 'Z'))
		printf("Это символ алфавита.\n");
	else if (c >= '0' && c <= '9')
		printf("Это цифра.\n");
	else
		printf("Это специальный символ.\n");

return 0;
}
```

Условный оператор if

```
#include <stdio.h>
void main()
{
   float value1, value2;
   char op;
   printf("Введите ваше выражение.\n");
   scanf("%f %c %f", &value1, &op, &value2);

   if (op == '+')
        printf("%.2f\n", value1 + value2);
   else if (op == '-')
        printf("%.2f\n", value1 - value2);
   else if (op =='*')
        printf("%.2f\n", value1 * value2);
   else if (op == '/')
        printf("%.2f\n", value1 / value2);
}
```

Условный оператор **switch**

- Формально можно пользоваться конструкцией **if else if … else**. Однако во многих случаях оказывается более удобным применять оператор **switch**. Синтаксис данного оператора следующий:

```
switch (<переменная>)
{
   case <константа1>:
        <операторы>
   case <константа2>:
        <операторы>
   …
   default:
         <операторы>
}
```

Условный оператор **switch**

```c
#include <stdio.h>
void main()
{
    int x;
    printf("Введите число: ");
    scanf("%i", &x);

    switch (x)
    {
        case 1: printf("Введено число 1\n"); break;
        case 2: printf("Введено число 2\n"); break;
        default: printf("Введено другое число\n");
    }

    char ch;
    printf("Введите символ: ");
    scanf("%c", &ch);

    switch (ch)
    {
        case 'a': printf("Введен символ a\n"); break;
        case 'b': printf("Введен символ b\n"); break;
        default: printf("Введен другой символ\n");
    }
}
```

Условный оператор ?:

- Тернарная операция - это условная операция **?:**

***логическое выражение* ? *Выражение*1 : *выражение*2**

- **j = (i<0) ? (-i) : (i);** Если i меньше нуля, то j присваивается -i. Если i больше или равно нулю, то j присваивается i.
- **min = (a < b) ? a : b;**Вычисляется минимальное значение из чисел

Операторы циклов

- Часто при создании программ требуется много раз выполнить одну и ту же группу операторов.
- **Оператор цикла while.** С помощью данного оператора реализуется цикл, который выполняется до тех пор, пока истинно условие цикла.

```
while (<условие>)
{
    <тело цикла>
}
```

Оператор цикла **while**

```
int N = 20, i = 0;
long S = 0;
while (S < N)
{
      S = S + i;
      i++;
}
```

Цикл *while* реализуется с условием **i < N**. Так как начальное значение переменной **i=0**, а **N=20**, то условие истинно и выполняется тело цикла, в котором осуществляется суммирование переменной **i** и увеличение ее на **1**. На **20** итерации значение **S=20**, условие станет ложным и цикл будет завершен.

Оператор цикла **while**

```
long S = 0;
int N = 20, i = 0;
while ((S = S + i++) < N);

//-------------------------------
```

```
int num;
while (scanf("%i", &num) == 1)
{
   printf("Вы ввели значение %i\n", num);
}
```

Данный цикл будет работать, пока пользователь вводит целочисленные значения и останавливается, если введена буква или вещественное число.

Оператор цикла **while**

Любой цикл можно принудительно завершить даже при истинном условии цикла. Это достигается путем использования оператора **break**.

```
int num;
while (scanf("%i", &num) == 1)
{
      if (num == 0) break;
      printf("Вы ввели значение %i\n", num);
}

//-----------------------------------------

while (scanf("%i", &num) == 1 && num != 0)
{
       printf("Вы ввели значение %i\n", num);
}
```

Оператор цикла **for**

```
for (<инициализация счетчика>; <условие>; <изменение счетчика>)
{
   <тело цикла>
}
```

Пример вывода таблицы кодов ASCII символов.

```
char ch;
for (ch = 'a'; ch <= 'z'; ch++)
     printf("Значение ASCII для %c - %i.\n", ch, ch);
```

В качестве счетчика цикла выступает переменная **ch**, которая инициализируется символом **‘a’**. Это означает, что в переменную **ch** заносится число **97** – код символа **‘a’**. Именно так символы представляются в памяти компьютера. Код символа ‘**z**’ – **122**, и все малые буквы латинского алфавита имеют коды в диапазоне [**97**; **122**]. Поэтому, увеличивая значение **ch** на единицу, получаем код следующей буквы, которая выводится с помощью функции **printf**().

Оператор цикла **for**

```
for (char ch = 97; ch <= 122; ch++)
     printf("Значение ASCII для %c - %i.\n", ch, ch);
```

Здесь следует отметить, что переменная **ch** объявлена **внутри** оператора **for**

```
int line = 1;
double debet;
for (debet = 100.0; debet < 150.0; debet = debet * 1.1, line++)
     printf(" %i.Ваш долг теперь равен % .2f.\n", line, debet);
```

Оператор цикла **for**

```
int exit = 0, mov;
for (int num = 0; num < 100 && !exit; num += 1)
{
    scanf("%i", &mov);
    if (mov == 0) exit = 1;
    printf("Произведение num * mov = %i.\n", num * mov);
}
```

Оператор цикла **for**

for с одним условием:

```
int i = 0;
for (; i < 100;) i++;
```

и без условия:

```
int i = 0;
for (;;) { i++;  if (i > 100)  break; }
```

Оператор цикла **do while**

- Представленные выше операторы циклов проверяют условие перед выполнением цикла, поэтому существует вероятность, что операторы внутри цикла никогда не будут выполнены – это циклы с **предусловием**.
- Иногда целесообразно выполнять проверку условия после того, как будут выполнены операторы, стоящие внутри цикла. Это достигается путем использования операторов **do while**, которые реализуют цикл с **постусловием**.

Оператор цикла **do while**

```
const int secret_code = 13;
int code;
do
{
    printf("Введите секретный код : ");
    scanf("%i", &code);
} while (code != secret_code);
```

Программирование вложенных циклов

Внутри одного цикла может находиться другой. Вложенные циклы необходимы для решения большого числа задач, например, вычисления двойных, тройных и т.д. сумм, просмотр элементов двумерного массива.

```
long S = 0;
int M = 10, N = 5;
for (int i = 0; i <= N; i++)
    for (int j = 0; j <= M; j++)
        S += i * j;
```

Операторы **break** и **continue**

В теле цикла можно использовать оператор **break**, который позволяет выйти из цикла, не завершая его. Оператор **continue** позволяет пропустить часть операторов тела цикла и начать новую итерацию.

```
while (<выражение>)
{
    . . .
    break;
    . . .
}
```

```
while (<выражение>)
{
    . . .
    continue;
    . . .
}
```

Оператор безусловного перехода **goto**

```
goto метка;
   . . .
метка: <оператор>;
```

Использование **goto** в программе крайне нежелательно, так как он усложняет логику программы

```
if (size > 12)
     goto A;
else
     goto B;
A: cost = cost * 3;
  flag = 1;
B: s = cost* flag;
```

Генерация псевдослучайного числа

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

void main()
{
    const int max = 100;
    const int min = 50;
    const int N = 20;

    srand((unsigned)time(0));

    for (int i = 1; i <= N; i++) {
        int u = (double)rand() / (RAND_MAX + 1) * (max - min) + min;
        printf("%i. Псевдослучайное число в диапазоне [%i, %i] = %i\n",
        i, min, max, u);
    }
}
```

МАССИВЫ

3

Одномерные массивы

Массив представляет собой совокупность значений одного и того же типа. Каждый элемент массива определяется именем и порядковым номером, который называется индексом. Индексы могут быть только целыми числами.

тип элемента имя [размер];

```
char str[80];
float b[30];
int a[8];
```

При объявлении массива под него отводится память. Число байт, отводимое под массив определяется, как произведение размера типа элемента массива на количество элементов массива:

Общее число байт = sizeof (базовый тип) *количество элементов массива.

Размер массива

sizeof (char) * 80 = 1*80 = 80 байт.

sizeof (float) * 30 = 4*30 = 120 байт

Первый элемент массива имеет индекс **0**, а последний имеет индекс на единицу меньше, чем количество элементов массива. Таким образом, первый элемент массива **str** - это **str[0],** а последний – это **str[79].**

Отводимая под массив память представляет собой непрерывную область (линейную последовательность байт). Элементы массива хранятся в памяти, непосредственно примыкая, друг к другу.

Для обращения к элементу массива следует записать имя массива и индекс элемента. Например, **str[5]** – это шестой элемент массива **str**, **a[4]** – пятый элемент массива **a**.

```
char ch = str[5];
```

Доступ к элементам массива

Следующий фрагмент программы демонстрирует запись в массив значений линейной функции $f(x) = kx + b$ и вывода значений на экран:

```
#include <stdio.h>

void main()
{
   double k = 0.5, b = 10.0;
   double f[100];
   for (int x = 0; x < 100; x++)
   {
      f[x] = k * x + b;
      printf("%.2f ", f[x]);
   }
}
```

Инициализация массива

Одновременно с объявлением элементам массива могут быть присвоены значения. Значения, которые присваиваются элементам массива должны представлять собой константы того же типа, с которым объявлен массив.

```
int powers[4] = { 1, 2, 4, 6 };
int data[] = { 2, 16, 32, 64, 128, 256 };
```

Размер можно задавать только константами:

```
const int N = 100;
float array_f[N];
```

Пример использования

```c
#include <stdio.h>
#define MONTHS 12

void main(void)
{
    //объявление и инициализация массива days
    //каждая константа в списке инициализации представляет собой
    //количество дней в соответствующем месяце
    int days[MONTHS] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
    int index;

    //вывод массива
    for (index = 0; index < MONTHS; index++)
         printf("Месяц %i имеет %i дней \n", index + 1, days[index]);

}
```

Двумерные массивы

Язык С позволяет использовать массивы любой размерности, но наиболее часто встречаются двумерные массивы.

Двумерный массив можно представить как таблицу, имеющие столбцы и строки. Такой массив следует объявлять с двумя индексами, один из которых определяет количество строк, а второй количество столбцов таблицы.

```
int table[3][4];
```

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]

В памяти двумерный массив занимает непрерывную область и располагается по строкам, как одномерный.

Инициализация двумерных массивов

```
int table[3][4] = {  {5 ,8 ,9 ,7},
                     {3, 6, 4, 7},
                     {6, 2, 1, 0} };
```

5	8	9	7
3	6	4	7
6	2	1	0

```
int table[3][4] = {  {5 ,8},
                     {3, 6, 4, 7},
                     {6, 1, 0}  };
```

5	8		
3	6	4	7
6	1	0	

Пример использования

```
#include <stdio.h>

int room[2][10] = {  {102,107,109,112,115,116,123,125,127,130},
                     {12,43,23,12,20,15,16,23,12,15} };

void main(void) {
    int i, j, flag = 0, num;

    puts("Вместимость всех комнат гостиницы:");
    for (j = 0; j < 10; j++)
            printf("Комната #%i рассчитана на %i мест\n", room[0][j], room[1][j]);

    puts("Введите минимальное необходимое количество мест");
    scanf("%i", &num);

    // поиск списка комнат с заданной вместимостью
    for (j = 0; j < 10; j++)
        if (room[1][j] >= num)
        {
            flag = 1;
            printf("Комната #%i рассчитана на %i мест\n", room[0][j], room[1][j]);
        }

    if (flag == 0)
           puts("Комнат с таким количеством мест нет");
}
```

СТРОКИ

4

Строки

В языке С нет специального типа данных для строковых переменных. Для этих целей используются массивы символов (тип char).

```
char str_1[100] = { 'П','р','и','в','е','т','\0' };
char str_2[100] = "Привет";
char str_3[] = "Привет";
printf(" %s\n  %s\n  %s\n ", str_1, str_2, str_3);
```

В частности символ **‘\0’** означает в языке С++ конец строки и все символы после него игнорируются как символы строки.

```
char str1[10] = { 'H','e','l','l','o','\0' };
char str2[10] = { 'H','e','l','l','\0','o' };
char str3[10] = { 'H','e','\0','l','l','o' };
printf(" % s\n  %s\n  % s\n", str1, str2, str3);
```

strlen

Таким образом, чтобы подсчитать длину строки (число символов) необходимо считать символы до тех пор, пока не встретится символ **‘\0’** или не будет достигнут конец массива. Функция вычисления размера строк уже реализована в стандартной библиотеке языка C **string.h**:

```
#include <stdio.h>
#include <string.h>

void main(void) {
	char str[] = "Hello world!";
	int length = strlen(str);
	printf("Длина строки = %i.\n", length);
}
```

strcpy

Теперь рассмотрим правила присваивания одной строковой переменной другой.

```
char str1[] = "First string";
char str2[] = "Second string";
```

При записи такого оператора присваивания **str1 = str2;** компилятор выдаст сообщение об ошибке.

```
#include <stdio.h>
#include <string.h>

void main(void) {
	char src[] = "Hello world!";
	char dest[100];
	strcpy(dest, src);
	printf("%s\n", dest);
}
```

strcmp

Сравнения двух строк между собой:

```
#include <stdio.h>
#include <string.h>
```

```
void main(void) {
   char str1[] = "First string";
   char str2[] = "Second string";

   if (strcmp(str1, str2) == 0)
        printf("Срока %s равна строке % s\n", str1, str2);
   else
        printf("Срока %s не равна строке %s\n", str1, str2);
}
```

Ввод строк с клавиатуры

```
#include <stdio.h>
#include <string.h>

#define DEF "First string"

void main(void) {
	char str[100];
	//scanf("%s", str);
	gets_s(str);

	puts(str);
	puts(DEF);
	puts(&str[4]);
}
```

sprintf

```
#include <stdio.h>
#include <string.h>

void main(void) {
    int age;
    char name[100], str[100];
    puts("Введите Ваше имя : ");
    scanf("%s", name);
    printf("Введите Ваш возраст : ");
    scanf("%i", &age);
    sprintf(str, "Здраствуйте %s.Ваш возраст %i
    лет", name, age);
    puts(str);
}
```

Массивы строк

```
#include <stdio.h>

void main(void)
{
	char name[10][20];
	int index;

	// ввод строк
	// цикл по индексу строки
	for (index = 0; index < 10; index++)
	{
		printf("Введите строку #%i: ", index);
		gets_s(name[index]);
		puts("");
	}

	for (index = 0; index < 10; index++)
		puts(name[index]);
}
```

УКАЗАТЕЛИ

5

Объявление указателей

Указатель – это переменная, хранящая адрес некоторого данного (объекта).

Память компьютера делится на 8-битовые байты. Каждый байт пронумерован, нумерация байт начинается с нуля. Номер байта называют адресом;

Таким образом, указатель является просто **адресом** байта памяти компьютера.

Использование указателей в программах позволяет:

- Упростить работу с массивами;
- Распределять память под данные динамически, то есть в процессе исполнения программы;
- Выполнить запись и чтение данных в любом месте памяти.

Значение указателя сообщает о том, где размещен объект

Объявление указателей

Указатели, как и другие переменные, должны быть объявлены в программе. При объявлении указателя перед его именем ставится *

```
int* iptr; // * - означает «указатель на»
```

iptr - это указатель на объект типа **int**. Этим объектом может быть простая переменная, типа int, массив элементов типа int или блок памяти, полученный, например, при динамическом распределении памяти.

```
static float* f;
extern double* z;
extern char* ch;
```

Каждое из этих объявлений выделяет память для переменной типа указатель, но каждый из указателей пока ни на что не указывает
До тех пор, пока указателю не будет присвоен адрес какого – либо объекта, его нельзя использовать в программе.

Классы памяти

Класс памяти переменной (**Storage class**) —определяет область видимости переменной, а также как долго переменная находится в памяти.
C/C++ располагает четырьмя спецификаторами класса памяти:

```
auto
register
static
extern
```

Классы памяти

auto — автоматическая (локальная), динамическая переменная. Автоматические переменные создаются при входе в функцию и уничтожаются при выходе из неё. Они видны только внутри функции или блока, в которых определены. Этот класс памяти используется, если не указан ни один из четырёх модификаторов, и в C++0x значение слова auto изменили.

static — статическая переменная (локальная). Статические переменные имеют такую же область действия, как автоматические, но они не исчезают, когда содержащая их функция закончит свою работу. Компилятор хранит их значения от одного вызова функции до другого.

extern — внешняя (глобальная) переменная. Внешние переменные доступны везде, где описаны, а не только там, где определены. Включение ключевого слова extern позволяет функции использовать внешнюю переменную, даже если она определяется позже в этом или другом файле.

register — регистровая переменная (локальная). Это слово является всего лишь «пожеланием» компилятору помещать часто используемую переменную в регистры процессора для ускорения про-граммы.

Доступ к объекту через указатель

Для получения адреса какого – либо объекта используется операция &. Эта операция позволяет присваивать указателям адреса объектов.

```
int a, * aptr;
char c, * cptr;
aptr = &a;
cptr = &c;
```

Доступ к объекту через указатель

- Операция * называется операцией разадресации или операцией обращения по адресу. Операция * рассматривает свой операнд как адрес некоторого объекта и использует этот адрес для выборки содержимого.
- Для вывода адреса памяти, содержащегося в указателе, необходимо использовать спецификацию формата **%p** (значение адреса выводится в шестнадцатеричной системе счисления).

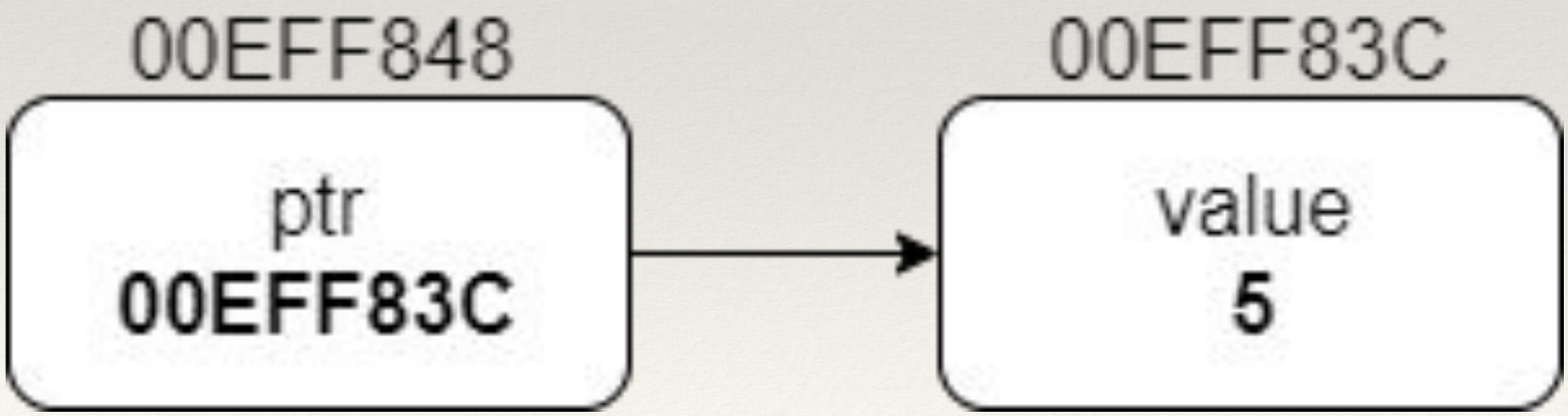
Доступ к объекту через указатель

```
#include <stdio.h>

void main(void)
{
    int x, *px;
    px = &x;
    x = 35;
    printf("Адрес x: %p\n", &x);
    printf("Значение указателя: %p\n", px);
    printf("Значение x: %i\n", x);
    printf("Значение, адресуемое указателем: %i\n", *px);
}
```

Адрес x: **0133FC94**
Значение указателя: **0133FC94**
Значение x: **35**
Значение, адресуемое указателем: **35**

Доступ к объекту через указатель

```
int value = 5;
int* ptr = &value;
```



Инициализация указателей

Следует избегать использования в программе неинициализированных указателей. Всегда должен существовать объект, адрес которого содержит указатель.

Правильно

```
float f, *fp = &f;
int s[80], *sp = s;
```

Не правильно

```
float f;
int* pi;
f = *pi;
```

Указатель на неопределенный тип

Существует специальный тип указателя, называемый указателем на неопределенный тип: **void * имя;**

```
#include <stdio.h>

void main()
{
        int a = 123;
        double d = 3.45678;
        void* vp;

        vp = &a;
        printf("a=%d \n", *((int*)vp));

        vp = &d;
        printf("d=%lf \n", *((double*)vp));
}
```

Выражения с указателями

Можно присваивать один указатель другому:

```
int a = 123, *a1 = &a, *a2;
a2 = a1;
printf("%p %p", a1, a2);
```

Приоритет у унарной операции разадресации * выше, чем у бинарных арифметических операций:

```
int  x = 5, y, *px = &x;
y = *px + 5;
```

Операции с указателями

С указателями можно использовать только следующие операции:

- `++` инкремента
- `--` декремента
- `+, -` сложения и вычитания

Если к указателю применяются операции ++ или --, то указатель увеличивается или уменьшается на **размер** объекта, который он адресует:

```
int   i, *pi = &i;
float a, *pa = &a;
pi++;
pa++;
pa = pa + 3;
```

Операции с указателями

```
int x = 5, y, *px = &x;

y = *px + 2;
printf("y = %i значение указателя = %i\n", y, px);

y = *px++;
printf("y = %i значение указателя = %i\n", y, px);
```

```
px = &x;
y = (*px)++;

printf("y = %i значение указателя = %i. Значение, адресуемое
указателем *px = %i\n", y, px, *px);

y = ++ *px;
printf("y = %i значение указателя = %i\n", y, px);
```

y = 7 значение указателя = 14350828
y = 5 значение указателя = 14350832
y = 5 значение указателя = 14350828. Значение, адресуемое указателем *px = 6
y = 7 значение указателя = 14350828

Операции с указателями

```
int x = 5, y, *px = &x;

y = *px + 2;
printf("y = %i значение указателя = %i\n", y, px);

y = *px++;
printf("y = %i значение указателя = %i\n", y, px);
```

```
px = &x;
y = (*px)++;

printf("y = %i значение указателя = %i. Значение, адресуемое
указателем *px = %i\n", y, px, *px);

y = ++ *px;
printf("y = %i значение указателя = %i\n", y, px);
```

y = 7 значение указателя = 14350828
y = 5 значение указателя = 14350832
y = 5 значение указателя = 14350828. Значение, адресуемое указателем *px = 6
y = 7 значение указателя = 14350828

УКАЗАТЕЛИ И МАССИВЫ

6

Константный указатель

Между указателями и массивами существует прямая связь. Когда объявляется массив:

int arr [5];

то идентификатор массива **arr** определяется как константный указатель на первый (с индексом 0) элемент массива. Это означает, что имя массива содержит адрес элемента массива с индексом 0:

arr == &arr[0];

Константный указатель

Так как идентификатор массива содержит адрес, то можно, например, записать:

```
int arr[5];
int *parr;
parr = arr; // parr = &arr[0];
```

Адрес элемента массива с индексом **i** может быть записан:

&arr[i] эквивалентно: **parr+i**

Значение элемента массива с индексом i может быть записано:

arr[i] эквивалентно: ***(parr+i)** или ***(arr+i)**

Указатель – это переменная, и ее значение можно изменять в программе

Указатель на массив

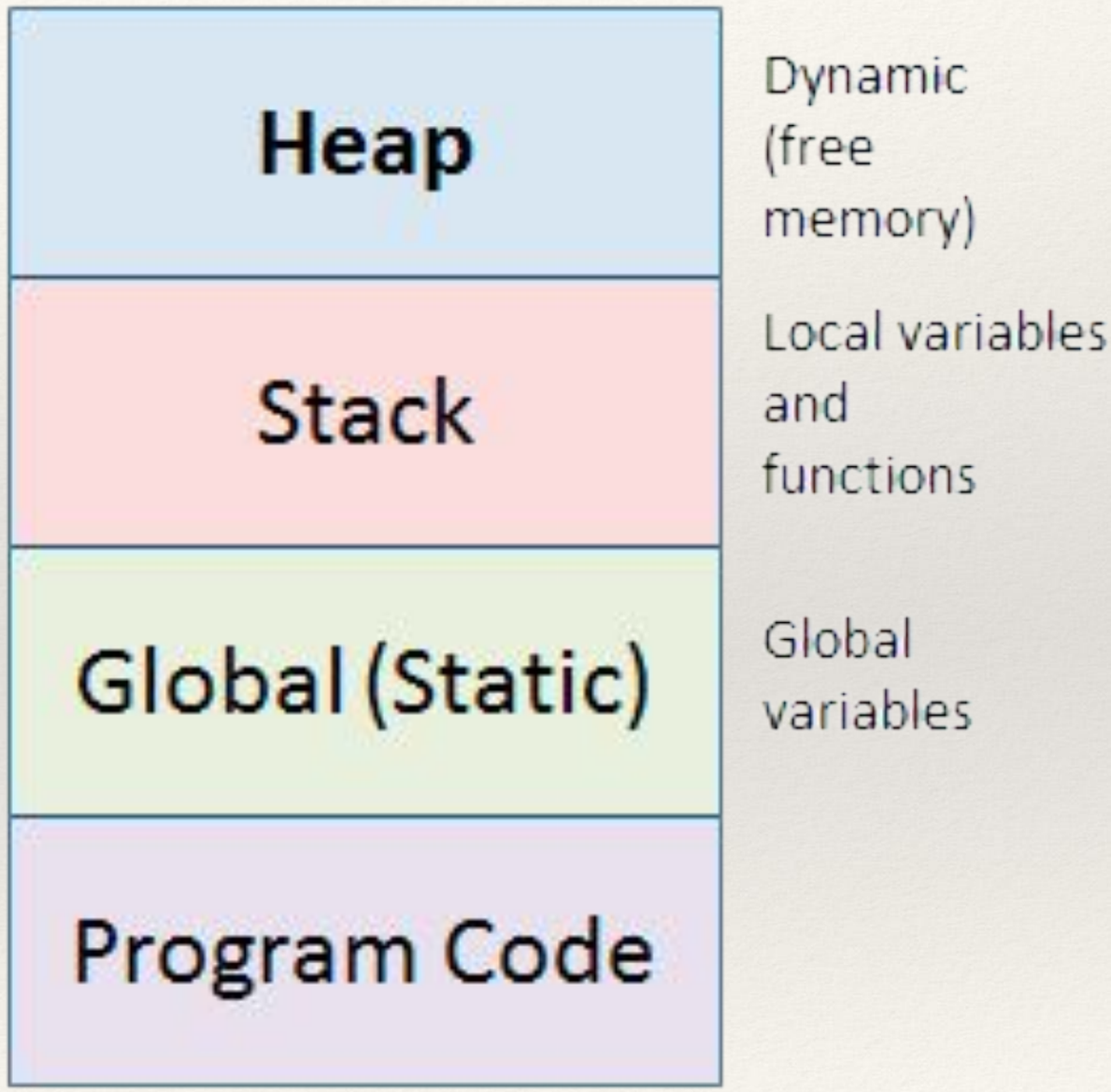
```c
#include <stdio.h>
#include <stdlib.h>

#define N 100

int main(void)
{
	int a[N];
	int* pa = a;

	for (int i = 0; i < N; i++)
		*(pa + i) = rand() % 50;

	for (int i = 0; i < N; i++)
		printf("%4i", *(pa + i));
}
```

Распределение памяти в С

- Память под переменные можно распределять в процессе исполнения программы.

- Переменные, память под которые выделяется в процессе исполнения программы, называются **динамическими**.

- Они запоминаются в блоках памяти переменного размера, известных как «**куча**» (**Heap**).



Динамические массивы

- Динамические переменные используются, если в программе нужно работать с массивом, размер которого заранее не известен.
- Для динамического выделения свободной памяти можно использовать функции **malloc()** из **<malloc.h>** или оператор **new**

malloc() и new

Для **malloc()** в круглых скобках задается число байт для выделения из кучи:

```
double *fp = (double*)malloc(16);

int *ip = (int*)malloc(4 * sizeof(int));
```

Для **new** задается размер массива:

```
float *fp = new float[5];
```

Нехватка памяти

Функция **malloc**() в случае ошибки (например, в куче не хватает объема запрашиваемой памяти) возвращают **NULL**

```
float* fp;
fp = (float*)malloc(4 * sizeof(float));
if (fp == NULL)
    printf("Ошибка распределения памяти!");
```

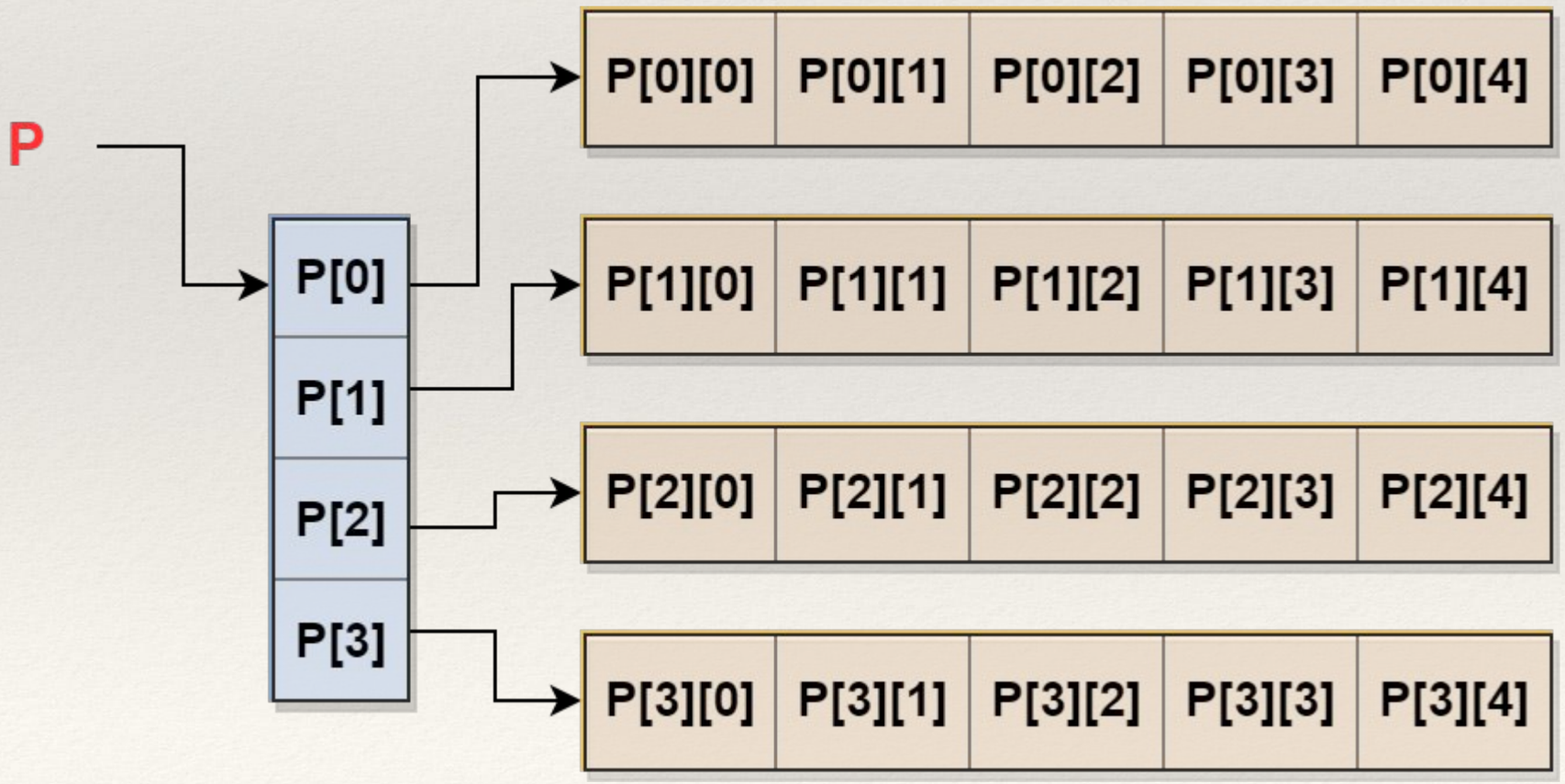
Удаление памяти в куче

- По окончании работы с выделенным блоком памяти его следует освободить.
- Если не освободить выделенную в куче память, которая больше не нужна, ее нельзя будет использовать до конца работы программы.

```
int *mp = (int*)malloc(50 * sizeof(int));
free(mp);


int *np = new int[50];
delete [] np;
```

Двумерные массивы

`int P[4][5];` - можно рассматривать как объявление одномерного массива **P**, состоящего из четырех элементов, каждый элемент которого, в свою очередь, состоит из пяти элементов.



Двумерные динамические массивы

Псевдодвумерный:

```
int *pa, m = 5, n = 4;
```

```
pa = (int)malloc(m * n * sizeof(int));
```

Двумерный:

```
int **pa, m = 5, n = 4
```

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

В памяти двумерный массив представлен как одномерный

m – количество строк в массиве, **n** – количество столбцов. Память под массив можно распределить динамически

Адрес элемента массива, находящегося в строке с индексом **i** и столбце с индексом **j** можно вычислить так:

```
pa +i*m +j;
```

```
*(pa+i)+j
```

Значение элемента массива, находящегося в строке с индексом **i** и столбце с индексом **j** можно вычислить так:

```
*(pa +i*m +j );
```

```
*(*(pa+i)+j)
```

Двумерные динамические массивы

```
int rows = 2;
int cols = 5;

int **rooms;

// создание
rooms = new int *[rows];          // массив указателей
for (int i = 0; i < rows; i++) {
    rooms[i] = new int[cols];       // инициализация указателей
}
```

```
for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++)
        rooms[i][j] = rand()%10;


for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        printf("%4i", rooms[i][j]);
    }
    puts("");
}

// уничтожение
for (int i = 0; i < rows; i++) {
    delete[] rooms[i];
}
delete[] rooms;
```

Свободные массивы

Свободный массив – это массив, в котором количество элементов в строке может быть различными. Свободные массивы могут быть любого типа. Для работы со свободными массивами используют массивы указателей, содержащие количество элементов, равное количеству строк свободного массива. Оперативная память для каждой строки может выделяться либо статически, либо динамически.

```
int *a[100];
```

Объявлен массив указателей (в нем 100 элементов) на данные типа **int**. Память выделяется для **100** указателей, по одному указателю на каждую из 100 строк свободного массива. Память для строк массива не выделена. Это можно сделать динамически. Например, если m - количество элементов в строке с индексом i, то память для этой строки может быть выделена так:

```
a[i] = new int [m];
```

Наиболее часто свободные массивы в программах на С используются при работе со строками:

```
char *errors[] = { "Невозможно открыть файл",
                   "Ошибка распределения памяти",
                    "Системная ошибка“
                     };
```

ФУНКЦИИ

7

Функции

Программы на языке Си обычно состоят из большого числа отдельных функций (подпрограмм). Все функции являются глобальными. В языке запрещено определять одну функцию внутри другой. Связь между функциями осуществляется через аргументы, возвращаемые значения и внешние переменные.

Определение функции имеет следующий вид:

тип имя_функции(тип параметр_1**, тип** параметр_2**,...)**
```
{
    тело функции
}
```

Передача значения из вызванной функции в вызвавшую происходит с помощью оператора возврата **return**, который записывается в следующем формальном виде:

```
return выражение;
```

Функции

```
#include <stdio.h>

int fmax(int a, int b)
{
    if (a > b) {
            printf("max = %i\n", a);
            return a;
    }

    printf("max = %i\n", b);
    return b;
}

//-------------------------------------

void main()
{
    int c, d = 2, g = 8;

    c = fmax(15, 5);
    c = fmax(d, g);
    fmax(d, g);
}
```

Вызвавшая функция может при необходимости игнорировать возвращаемое значение. После слова **return** можно ничего не записывать; в этом случае вызвавшей функции никакого значения не передается. Управление передается вызвавшей функции.

Аргументы по умолчанию

Можно задавать значения аргументов функции, которые будут использоваться по умолчанию, т.е. если программист не введет свое значение.

```
void some(int a = 1, int b = 2, int c = 3)
{
   printf("a = %i, b = %i, c = %i\n", a, b, c);
}

void main()
{
     some();
     some(10);
     some(10, 20);
     some(10, 20, 30);
}
```

Передача значений

В языке Си аргументы функции передаются по значению, т.е. вызванная функция получает свою временную **копию** каждого аргумента, а не его адрес. Это означает, что вызванная функция не может изменить значение переменной вызвавшей ее программы.

Однако это легко сделать, если передавать в функцию не переменные, а их адреса:

```
void swap(int* a, int* b)
{
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
```

```
void main()
{
   int a = 5, b = 4;

   swap(&a, &b);

   printf("a = %i, b = %i", a, b);
}
```

Передача массива

Если же в качестве аргумента функции используется имя массива, то передается только адрес начала массива, а сами элементы не копируются. Функция может изменять элементы массива.

Функции можно передать массив в виде параметра:

- Параметр задается как массив (например: **int m[100]**).
- Параметр задается как массив без указания его размерности (например: **int m[]**).
- Параметр задается как указатель (например: **int *m**). Этот вариант используется наиболее часто.

Передача массива

```
#include <stdio.h>

int minimum(int values[], int numberOfElements)
{
   int minValue, i; minValue = values[0];

   for (i = 1; i < numberOfElements; ++i)
         if (values[i] < minValue)
            minValue = values[i];

   return minValue;
}


int main(void)
{
   int array1[5] = { 157, -28, -37, 26, 10 };
   int array2[7] = { 12, 45, 1, 10, 5, 3, 22 };

   printf("array1 minimum: %i\n", minimum(array1, 5));
   printf("array2 minimum: %i\n", minimum(array2, 7));

return 0;
}
```

Перегрузка функций

- В языке C++ предусмотрена перегрузка функций, то есть функция с одним именем описана для разных типов параметров, с разными результатами, если написать функцию с разным результатом, но с одним списком параметров, то компилятор выдаст ошибку.)
- Перегрузка функций позволяет программам определять несколько функций с одним и тем же именем и типом возвращаемого значения.

Перегрузка функций

```c
#include <stdio.h>

int add(int a, int b)
{
     return(a + b);
}
```

```c
int add(int a, int b, int c)
{
     return(a + b + c);
}


void main(void)
{
     printf("%i\n", add(200, 801));

     printf("%i\n", add(100, 201, 700));

}
```

Аргументы функции main( )

В программы на языке Си можно передавать некоторые аргументы. Когда вначале вычислений производится обращение к **main( )**, ей передаются три параметра:

- Первый из них определяет число командных аргументов при обращении к программе.
- Второй представляет собой массив указателей на символьные строки, содержащие эти аргументы (в одной строке - один аргумент).
- Третий тоже является массивом указателей на символьные строки, он используется для доступа к параметрам операционной системы (к переменным окружения).

Аргументы функции main( )

```
#include <stdio.h>

int main(int argc, char* argv[], char* env[]) {

   printf("Количество аргументов командной строки %i \n", argc);

   printf("Аргументы командной строки:\n");
   for (int i = 0; i < argc; i++)
        printf("%s\n", argv[i]);

   printf("\nАргументы состояния среды:\n");
   for (int i = 0; env[i] != NULL; i++)
        printf("%s\n", env[i]);

return 0;
}
```

C:\>prog.exe file1 file2 file3 <Enter>

Рекурсия

Рекурсией называется такой способ вызова, при котором функция вызывает сама себя.

```
#include <stdio.h>

void up_and_down(int n)
{
   printf("Уровень вниз %i\n", n);

   if (n < 4) up_and_down(n + 1);

   printf("Уровень вверх %i\n", n);
}

void main() {
   up_and_down(1);
}
```