Язык си выдержки

Структура программы Си

#include <stdio.h>        // подключаем заголовочный файл stdio.h

int main(void)                  // определяем функцию main

{                               // начало функции

    printf("Hello METANIT.COM!");       // выводим строку на консоль

    return 0;                   // выходим из функции

}                               // конец функции

Переменные

|  |  |
| --- | --- |
| **Тип** | **Спецификатор** |
| **unsigned short** | **hu** |
| **long** | **ld** |
| **unsigned long** | **lu** |
| **long long** | **lld** |
| **unsigned long long** | **llu** |

* **char**: представляет один символ. Занимает в памяти 1 байт (8 бит). Может хранить любое значение из диапазона от -128 до 127
* **unsigned char**: представляет один символ. Занимает в памяти 1 байт (8 бит). Может хранить любой значение из диапазона от 0 до 255
* **signed char**: то же самое, что и char
* **short**: представляет целое число в диапазоне от –32768 до 32767. Занимает в памяти 2 байта (16 бит).

Имеет псевдонимы **short int**, **signed short** и **signed short int**.

* **unsigned short**: представляет целое число в диапазоне от 0 до 65535. Занимает в памяти 2 байта (16 бит).

Имеет псевдоним **unsigned short int**.

* **int**: представляет целое число. В зависимости от архитектуры процессора может занимать 2 байта (16 бит) или 4 байта (32 бита). Если брать основные платформы - 64-разрядные Windows, Linux (вместе с Android) и MacOS, то размер int составляет 4 байта. Диапазон предельных значений соответственно также может варьироваться от –32768 до 32767 (при 2 байтах) или от −2 147 483 648 до 2 147 483 647 (при 4 байтах) и выше.

Имеет псевдонимы **signed int** и **signed**

* **unsigned int**: представляет положительное целое число. В зависимости от архитектуры процессора может занимать 2 байта (16 бит) или 4 байта (32 бита), и из-за этого диапазон предельных значений может меняться: от 0 до 65535 (для 2 байт), либо от 0 до 4 294 967 295 (для 4 байт).

Имеет псевдоним **unsigned**: то же самое, что и **unsigned int**

* **long**: представляет целое число и занимает в памяти 4 байта (32 бита) или 8 байт (64 бита). В зависимости от размера может находиться в в диапазоне от -2 147 483 648 до 2 147 483 647 (4 байта), либо в диапазоне от -9223372036854775807 до +9 223 372 036 854 775 807 (8 байт)

Если брать распростренные платформы, то на 64-разрядном Windows long занимает 4 байта, а на 64-разрядных Linux/MacOS - 8 байт.

Имеет псевдонимы **long int**, **signed long int** и **signed long**.

* **unsigned long**: представляет целое число и занимает в памяти 4 байта (32 бита) или 8 байт (64 бита). В зависимости от размера может находиться в в диапазоне от 0 до 4 294 967 295 (4 байта) или в диапазоне от 0 до 18 446 744 073 709 551 615 (8 байт).

Имеет псевдоним **unsigned long int**.

* **long long**: представляет целое число в диапазоне от -9223372036854775807 до +9 223 372 036 854 775 807. Занимает в памяти, как правило, 8 байт (64 бита).

Имеет псевдонимы **long long int**, **signed long long int** и **signed long long**.

* **unsigned long long**: представляет целое число в диапазоне от 0 до 18 446 744 073 709 551 615. Занимает в памяти, как правило, 8 байт (64 бита).

Имеет псевдоним **unsigned long long int**.

* **float**: представляет вещественное число одинарной точности с плавающей точкой в диапазоне +/- 3.4E-38 до 3.4E+38. В памяти занимает 4 байта (32 бита)
* **double**: представляет вещественное число двойной точности с плавающей точкой в диапазоне +/- 1.7E-308 до 1.7E+308. В памяти занимает 8 байт (64 бита)
* **long double**: представляет вещественное число двойной точности с плавающей точкой в диапазоне +/- 3.4E-4932 до 1.1E+4932. В памяти занимает 10 байт (80 бит). На некоторых системах может занимать 96 и 128 бит.
* **void**: тип без значения

**unsigned int**, **unsigned long int** и **unsigned long long int - u** или **U**

**unsigned long int** и **unsigned long long int - ul** или **UL**

**long long int - ll** или **LL**

**unsigned long long int - ull** или **ULL**

#include <stdio.h>

int main(void)

{

    unsigned number1 = 4294967294u;

    unsigned int number2 = 22U;

    printf("number1 = %u\n", number1);

    printf("number2 = %u\n", number2);

    return 0;

}

**unsigned short - hu**

**long - ld**

**unsigned long - lu**

**long long - lld**

**unsigned long long - llu**

#include <stdio.h>

int main(void)

{

    char letter = 'A';

    printf("letter: %d \n", letter);    // letter: 65

    printf("letter: %c \n", letter);    // letter: A

    return 0;

}

Оператор **typedef** позволяет определить для определенного типа псевдоним.

#include <stdio.h>

typedef unsigned char BYTE;

int main(void)

{

    BYTE byte = 22;

    printf("byte = %d", byte);

}

оператор **sizeof()**, который возвращает размер памяти в байтах, которую занимает переменная

printf("sizeof(number) = %d \n", sizeof(number));

**Консольный вывод. Функция printf**

* **\n**: перевод на новую строку
* **\t**: табуляция
* **\r**: возврат каретки (курсора) в начало строки
* **\\**: обратный слеш
* **\'**: одинарная кавычка
* **\"**: двойная кавычка

printf("The book \"The C Programming Language\"\nBrian Kernighan, Dennis Ritchie");

The book "The C Programming Language"

Brian Kernighan, Dennis Ritchie

### **Спецификации преобразования**

Спецификации преобразования позволяют задать формат вывода различных типов данных. Формальное определение спецификаций преобразования выглядит следующим образом:

|  |  |
| --- | --- |
| 1 | % флаги ширина\_поля.точность модификатор спецификатор |

Из этих компонентов обязательными являются только два % и спецификатор.

Какие спецификаторы мы можем использовать:

* **%c**: для вывода отдельных символов (тип char)
* **%s**: для вывода строк
* **%d**: для вывода целых чисел со знаком (тип int)
* **%i**: для вывода целых чисел со знаком (тип int)
* **%u**: для вывода целых положительных чисел (тип unsigned)
* **%f**: для вывода чисел с плавающей точкой (float, double)
* **%F**: для вывода чисел с плавающей точкой в верхнем регистре (float, double)
* **%e**: для вывода экспоненциального представления чисел с плавающей точкой (float, double)
* **%E**: для вывода экспоненциального представления чисел с плавающей точкой в верхнем регистре (float, double)
* **%g**: для вывода чисел используется более короткий из форматов %e или %f
* **%G**: для вывода чисел используется более короткий из форматов %E или %F
* **%o**: для вывода восьмеричных чисел без знака
* **%x**: для вывода шестнадцатеричных чисел
* **%X**: для вывода шестнадцатеричных чисел в верхнем регистре
* **%%**: для вывода знака процента

float r = 71.8986;

printf("x = %4.2f \n", r);  // 71.90

* **h**: для вывода значений **short int**
* **l**: для вывода значений **long int** и **unsigned long int**
* **ll**: для вывода значений **long long int** и **unsigned long long int**
* **L**: для вывода значений **long double**

|  |  |
| --- | --- |
| 1  2 | short r = 7100;  printf("x = %hd", r); |

Необязательно передавать по одному значению для вывода в функцию printf. Мы можем передавать множество значений:

|  |  |
| --- | --- |
| 1 | printf("Name=%s \t Age=%d \t Weight=%3.2f", "Tom", 23, 63.3456); |

Для определения константы используется ключевое слово **const**:

const float PI = 3.14;

При делении стоит быть внимательным, так как если в операции участвуют два целых числа, то результат деления будет округляться до целого числа, даже если результат присваивается переменной float:

|  |  |
| --- | --- |
| 1  2 | float f = 10 / 4;  printf("f = %f \n", f);     // 2.0 |

* Чтобы результат представлял вещественное число, один из операндов также должен представлять вещественное число:

|  |  |
| --- | --- |
| 1  2 | float f = 10.0 / 4;  printf("f = %f \n", f);     // 2.5 |

* %
* Возвращает результат деления двух чисел по модулю (получение остатка от деления):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | int a = 35 % 3;         // 2 (35 - 33 = 2)  Операции \*, /, % имеют больший приоритет, чем + и - :  Также в Си есть специальные унарные операции над одним числом: ++ (инкремент) и -- (декремент). Каждая из операций имеет две разновидности: префиксная и постфиксная.   * Префиксный инкремент (++x). Увеличивает значение переменной на единицу и полученное значение используется как значение выражения ++x  |  |  | | --- | --- | | 1  2  3  4 | int a = 8;  int b = ++a;  printf("a = %d \n", a); // 9  printf("b = %d", b);    // 9 |  * Постфиксный инкремент (x++). Увеличивает значение переменной на единицу, но значением выражения x++ будет то, которое было до увеличения на единицу:  |  |  | | --- | --- | | 1  2  3  4 | int a = 8;  int b = a++;  printf("a = %d \n", a); // 9  printf("b = %d", b);    // 8 |  * Префиксный декремент (--x). Уменьшает значение переменной на единицу, и полученное значение используется как значение выражения --x  |  |  | | --- | --- | | 1  2  3  4 | int a = 8;  int b = --a;  printf("a = %d \n", a); // 7  printf("b = %d", b);    // 7 |  * Постфиксный декремент (x--). Уменьшает значение переменной на единицу, но значением выражения x— * будет то, которое было до уменьшения на единицу:  |  |  | | --- | --- | | 1  2  3  4 | int a = 8;  int b = a--;  printf("a = %d \n", a); // 7  printf("b = %d", b);    // 8 |   Рассмотрим более сложную ситуацию:   |  |  | | --- | --- | | 1  2  3  4  5  6 | int a = 8;  int b = 7;  int c = a+++b;  printf("a = %d \n", a); // 9  printf("b = %d \n", b); // 7  printf("c = %d \n", c); // 15 |   Здесь операция трактуются таким образом, что вначале идет постфиксный инкремент,  а потом сложение. То есть фактически так:   |  |  | | --- | --- | | 1 | int c = (a++) + b; | |

## Логические операции и операции отношения

### перации сравнения

Операции сравнения (еще называют операциями отношения) позволяют сравнить два значения и возвращают **1**, если выражение истинно, и **0**, если выражение ложно. В языке Си есть следующие операции сравнения:

**==**

#include <stdio.h>

int main(void)

{

    int a = 10;

    int b = 4;

    int c = a == b;                 // 0 - числа НЕ равны

    printf("c = %d \n", c);         // c = 0

    int num = 5;

    int result = num == 2 + 3;      // 1 - числа равны

   printf("result = %d \n", result);     // result = 1

    return 0;

}

**!=**

Операция "не равно". Возвращает 1, если первый операнд не равен второму, и 0, если оба операнда равны:

|  |  |
| --- | --- |
| 1  2 | int c = 10 != 4;    // 1  int d = 4 != 4;     // 0 |

* **>**

Операция "больше чем". Возвращает 1, если первый операнд больше второго, и 0, если первый операнд меньше или равен второму:

|  |  |
| --- | --- |
| 1  2  3 | int a = 10;  int b = 4;  int c = a > b;   // 1 |

* **<**

Операция "меньше чем". Возвращает 1, если первый операнд меньше второго, и 0, если первый операнд больше или равен второму:

|  |  |
| --- | --- |
| 1 | int c = 10 < 4;  // 0 |

* **<=**

Операция "меньше или равно". Возвращает 1, если первый операнд меньше или равен второму, и 0, если первый операнд больше второго:

|  |  |
| --- | --- |
| 1 | int c = 10 <= 4; // 0 |

* **>=**

Операция "больше или равно". Возвращает 1, если первый операнд больше или равен второму, и 0, если первый операнд меньше второго:

|  |  |
| --- | --- |
| 1 | int c = 10 >= 4; // 1 |

### **Логические операции**

Логические операции, как правило, применяются к отношениям и объединяют несколько операций отношения. К логическим операциям относят следующие:

* **!** (операция отрицания)

Если операнд равен 0, то возвращает 1, иначе возвращает 0.

|  |  |
| --- | --- |
| 1  2 | int c = !2; // 0  int d = !0; // 1 |

* **&&** (конъюнкция, логическое умножение)

Возвращает 1, если оба операнда не равны 0. Возвращает 0, если хотя бы один операнд равен 0.

|  |  |
| --- | --- |
| 1  2 | int c = -2 && 5;    // 1  int d = 0 && 7;     // 0 |

* **||** (дизъюнкция, логическое сложение)

Возвращает 1, если хотя бы один операнд не равен 0. Возвращает 0, если оба операнда равны 0.

|  |  |
| --- | --- |
| 1  2  3 | int c = -2 || 5;    // 1  int d = 0 || 7;     // 1  int e = 0 || 0;     // 0 |

Используем одновременно несколько логических операций и операций сравнения:

|  |  |
| --- | --- |
|  | int a = -2 > 5;          // 0  int b = 0 < 7;           // 1  int c = 0 == 0;          // 1  int d = a && b || c;        // 1    printf("a = %d \n", a);  printf("b = %d \n", b);  printf("c = %d \n", c);  printf("d = %d \n", d); |

Операции сравнения имеют больший приоритет, чем логические операции, поэтому мы могли бы напрямую подставить операции сравнения:

|  |  |
| --- | --- |
| 1 | int d = -2 > 5 && 0 < 7 || 0 == 0;        // 1 |

### **Операции сдвига**

Каждое целое число в памяти представлено в виде определенного количества разрядов. И операции сдвига позволяют сдвинуть битовое представление числа на несколько разрядов вправо или влево. Операции сдвига применяются только к целочисленным операндам. Есть две операции:

* **<<**

Сдвигает битовое представление числа, представленного первым операндом, влево на определенное количество разрядов, которое задается вторым операндом.

* **>>**

Сдвигает битовое представление числа вправо на определенное количество разрядов.

Применение операций:

|  |  |
| --- | --- |
| 1  2 | int a = 2 << 2;           // 10  на два разрядов влево = 1000 - 8  int b = 16 >> 3;          // 10000 на три разряда вправо = 10 - 2 |

### **Поразрядные операции**

Поразрядные операции также проводятся только над разрядами целочисленных операндов:

* **&**: поразрядная конъюнкция (операция И или поразрядное умножение). Возвращает 1, если оба из соответствующих разрядов обоих чисел равны 1
* **|**: поразрядная дизъюнкция (операция ИЛИ или поразрядное сложение). Возвращает 1, если хотя бы один из соответствующих разрядов обоих чисел равен 1
* **^**: поразрядное исключающее ИЛИ. Возвращает 1, если только один из соответствующих разрядов обоих чисел равен 1
* **~**: поразрядное отрицание. Инвертирует все разряды операнда. Если разрядравен 1, то он становится равен 0, а если он равен 0, то он получает значение 1.

int a = 5 | 2;          // 101 | 010 = 111  - 7

int b = 6 & 2;          // 110 & 010 = 10  - 2

int c = 5 ^ 2;          // 101 ^ 010 = 111 - 7

int f = 12;              // 00001100

int d = ~f;         // 11110011   или -13

printf("a = %d \n", a);

printf("b = %d \n", b);

printf("c = %d \n", c);

printf("d = %d \n", d);

## Операции присваивания

|  |  |
| --- | --- |
| 1  2 | int a, b, c;  a = b = c = 34 +7; |

Здесь сначала вычисляется значение выражения c = 34 +7. Значение правого операнда - 34 +7 присваивается левому операнду с. То есть переменная c получает значение 41. Далее вычисляется выражение b = c: значение правого операнда **c** (41) присваивается левому операнду **b**. И в конце вычисляется выражение a = b: значение правого операнда **b** (41) присваивается левому операнду **a**.

Все остальные операции присваивания являются сочетанием простой операции присваивания с другими операциями:

* **+=**: присваивание после сложения. Присваивает левому операнду сумму левого и правого операндов: **A += B** эквивалентно **A = A + B**
* **-=**: присваивание после вычитания. Присваивает левому операнду разность левого и правого операндов: **A -= B** эквивалентно **A = A - B**
* **\*=**: присваивание после умножения. Присваивает левому операнду произведение левого и правого операндов: **A \*= B** эквивалентно **A = A \* B**
* **/=**: присваивание после деления. Присваивает левому операнду частное левого и правого операндов: **A /= B** эквивалентно **A = A / B**
* **%=**: присваивание после деления по модулю. Присваивает левому операнду остаток от целочисленного деления левого операнда на правый: **A %= B** эквивалентно **A = A % B**
* **<<=**: присваивание после сдвига разрядов влево. Присваивает левому операнду результат сдвига его битового представления влево на определенное количество разрядов, равное значению правого операнда: **A <<= B** эквивалентно **A = A << B**
* **>>=**: присваивание после сдвига разрядов вправо. Присваивает левому операнду результат сдвига его битового представления вправо на определенное количество разрядов, равное значению правого операнда: **A >>= B** эквивалентно **A = A >> B**
* **&=**: присваивание после поразрядной конъюнкции. Присваивает левому операнду результат поразрядной конъюнкции его битового представления с битовым представлением правого операнда: **A &= B** эквивалентно **A = A & B**
* **|=**: присваивание после поразрядной дизъюнкции. Присваивает левому операнду результат поразрядной дизъюнкции его битового представления с битовым представлением правого операнда: **A |= B** эквивалентно **A = A | B**
* **^=**: присваивание после операции исключающего ИЛИ. Присваивает левому операнду результат операции исключающего ИЛИ его битового представления с битовым представлением правого операнда: **A ^= B** эквивалентно **A = A ^ B**

Примеры операций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int a = 5;  a += 10;        // 15  a -= 3;         // 12  a \*= 2;         // 24  a /= 6;         // 4  a <<= 4;      // 64  a >>= 2;      // 16 |

## Преобразование типов

int a = 10;

int b = 4;

int c = a / b;                     // 2

double d = a / b;               // 2.00000

double e = (double)a / (double)b;   // 2.50000

printf("c = %d \n", c);

printf("d = %f \n", d);

printf("e = %f \n", e);

Без потери информации проходят следующие цепочки преобразований:

**char -> short -> int -> long**

**unsigned char -> unsigned short -> unsigned int -> unsigned long**

**float -> double -> long double**

## Условные конструкции

Оператор if проверяет истинность условия, и если оно истинно, выполняет блок инструкций. Этот оператор имеет следующую сокращенную форму:

|  |  |
| --- | --- |
| 1 | if (условие) инструкция; |

В качестве *условия* может выступать любое выражение, которое возвращает число. Если это выражение возвращает ненулевое значение (то есть выражение истинно), то выполняется последующая инструкция:

|  |  |
| --- | --- |
|  | include <stdio.h>  int main(void)  {   int n = 10;      if(n == 20) printf("n = 20 \n");      if(n == 10) printf("n = 10 \n");      return 0;  } |

Если в конструкции if необходимо выполнить не одну, а несколько инструкций, то эти инструкции оформляются в блок кода с помощью фигурных скобок:

|  |  |
| --- | --- |
|  | #include <stdio.h>   int main(void)  {    int x = 60;       if(x > 50)      {  printf("if statement \n");          printf("x is greater than 50 \n");      } return 0; } |

**Два условия:**

int x = 60;

if(x > 60)

    printf("x is greater than 60 \n");

else if (x < 60)

    printf("x is less than 60 \n");

else

    printf("x is equal 60 \n")

И также если после операторов должен идти набор инструкций, то эти инструкции оформляются в блок кода:

|  |  |
| --- | --- |
|  | #include <stdio.h>   int main(void)  {      int x = 60;      if(x > 60)      {          printf("if statement \n");          printf("x is greater than 60 \n");      }      else if (x < 60)      {          printf("else if statement \n");          printf("x is less than 60 \n");      }      else      {          printf("else statement \n");          printf("x is equal 60 \n");      }      return 0;  } |

### Конструкция switch

Другую форму организации ветвления программ представляет конструкция **switch...case**. Она имеет следующую форму:

|  |  |
| --- | --- |
|  | switch(выражение)  {  case константа\_1: инструкции\_1;      case константа\_2: инструкции\_2;       default: инструкции;  } |

После ключевого слова **switch** в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями после оператора **сase**. И если совпадение будет найдено, то будет выполняться данный блок **сase**.

В качестве констант после оператора **case** могут выступать значения типов char, int и unsigned. Например, проверим с помощью конструкции **switch** значение некоторого числа:

|  |  |
| --- | --- |
|  | #include <stdio.h>   int main(void)  {     int x = 2;      switch(x)      {          case 1:              printf("x = 1 \n");              break;          case 2:              printf("x = 2 \n");              break;          case 3:              printf("x = 3 \n");              break;      }      return 0;  } |

В конце конструкции switch может стоять блок **default**. Он необязателен и выполняется в том случае, если ни одна совпадения в блоках case не было найдено. Например, сравним значение переменной с набором значений:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int x = 5;        switch(x)      {          case 1:              printf("x = 1 \n");              break;          case 2:              printf("x = 2 \n");              break;          case 3:              printf("x = 3 \n");              break;          default:              printf("x is undefined \n");              break;      }      return 0; |

### **Тернарный оператор**

Тернарный оператор **?:** позволяет сократить определение простейших условных конструкций if и имеет следующую форму:

|  |  |
| --- | --- |
| 1 | [первый операнд - условие] ? [второй операнд] : [третий операнд] |

Оператор использует сразу три операнда. В зависимости от условия тернарный оператор возвращает второй или третий операнд: если условие равно 1 (то есть истинно), то возвращается второй операнд; если условие равно 0 (то есть ложно), то третий. Например:

|  |  |
| --- | --- |
|  | #include <stdio.h>   int main(void)  {      int x=5;      int y=2;     int z = x > y ? x-y : x+y;      printf("z = %d", z);        // z = 3      return 0;  } |

Здесь результатом тернарной операции является переменная z. Она имеет следующие операнды:

**Условие**

|  |  |
| --- | --- |
| 1 | x > y |

Действия, выполняемые, если условие **истинно**

|  |  |
| --- | --- |
| 1 | x - y |

Действия, выполняемые, если условие **ложно**

|  |  |
| --- | --- |
| 1 | x + y |

Поскольку в данном случае условие x < y истинно (то есть x больше чем y), то будет выполняться вычитание x - y, и переменная z получит его результат - число 3.

## Циклы

* **for**
* **while**
* **do...while**

Цикл for имеет следующее формальное определение:

|  |  |
| --- | --- |
|  | for (инициализация; условие; приращение;)  {      // тело цикла  } |

заголовок цикла состоит из трех частей. Первая часть - *инициализация* выполняется один раз при начале выполнения цикла и представляет установку начальных условий, как правило, это инициализация счетчиков - специальных переменных, которые используются для контроля за циклом.

Вторая часть - *условие*, при соблюдении которого выполняется цикл. Зачастую в качестве условия используется операция сравнения, и если она возвращает ненулевое значение (то есть условие истинно), то выполняется тело цикла, а затем вычисляется *выражение\_3*.

Третья часть - *приращение* задает изменение параметров цикла. Обычно здесь происходит увеличение счетчиков цикла.

Рассмотрим стандартный цикл **for** и для этого выведем числа с 0 до 5 на консоль:

|  |  |
| --- | --- |
|  | #include <stdio.h>  int main(void)  {      for (int i = 0; i < 6; i++)      {          printf("%d", i);      }      return 0;  } |

### **Цикл do..while**

В цикле **do..while** сначала выполняется код цикла, а потом происходит проверка условия в инструкции **while**. И пока это условие истинно, то есть не равно 0, то цикл повторяется.

|  |  |
| --- | --- |
| 1  2  3  4  5 | do  {      // действия цикла  }  while (условие); |

### **Цикл while**

В отличие от цикла do цикл while сразу проверяет истинность некоторого условия, и если условие истинно, то есть не равно 0, то код цикла выполняется:

|  |  |
| --- | --- |
| 1  2  3 | while(условие){      // выполняемые инструкции, если условие истинно  } |

Пример:

include <stdio.h>

int main(void)

{

    int i = 6;

    while (i > 0)

    {

        printf("%d \n", i);

        i--;

    }

    return 0;

}

### **Операторы continue и break**

Иногда возникает необходимость выйти из цикла до его завершения. В этом случае можно воспользоваться оператором **break**. Например:

|  |
| --- |
| int i = 1;  for ( ; ; )  {      printf("%d \* %d = %d \n", i, i,  i \* i);      i++;      if (i > 5) break;  } |

Здесь когда значение переменной i достигнет 5, осуществляется выход из цикла с помощью оператора **break**.

В отличие от оператора break, оператор **continue** производит переход к следующей итерации. Например, нам надо посчитать сумму только нечетных чисел из некоторого диапазона:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #include <stdio.h>  int main(void)  {      int result = 0;      for (int i=0; i<10; i++)      {          if (i % 2 == 0) continue;          result +=i;      }      printf("result = %d", result);  // 25      return 0;  } |

Чтобы узнать, четное ли число, мы получаем остаток от целочисленного деления на 2, и если он равен 0, то с помощью оператора continue переходим к следующей итерации цикла. А если число нечетное, то складываем его с остальными нечетными числами.

**Массив** представляет набор однотипных значений. Объявление массива выглядит следующим образом:

|  |  |
| --- | --- |
| 1 | тип\_переменной название\_массива [длина\_массива] |

После типа переменной идет название массива, а затем в квадратных скобках его размер. Например, определим массив из 4 чисел:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int main(void)  {      int numbers[4];      return 0;  } |

Используя номера элементов массиве, которые называются индексами, мы можем обратиться к отдельным элементам. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <stdio.h>    int main(void)  {      int numbers[4];      numbers[0] = 1;      numbers[1] = 2;      numbers[2] = 3;      numbers[3] = 4;        printf("numbers[0] = %d \n", numbers[0]);   // 1 - первый элемент      printf("numbers[2] = %d \n", numbers[2]);   // 3 - третий элемент      return 0;  } |

### **Размер и количество элементов массива**

Не всегда в программе может быть известен размер массива. В этом случае можно использовать оператор **sizeof**, который возвращает размер массива в байтах в виде значения типа **size\_t**:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int numbers[] = { 5, 6, 7};      size\_t size = sizeof(numbers);      printf("numbers size: %zu \n", size);     // numbers size: 12      return 0;  }  #include <stdio.h>  int main(void)  {      int numbers[] = { 5, 6, 7};      size\_t size = sizeof(numbers);      size\_t count = sizeof(numbers) / sizeof(int);      printf("numbers size: %zu \n", size);       // numbers size: 12      printf("numbers count: %zu \n", count);         // numbers count: 3      return 0;  } |

### **Перебор элементов массива**

Используя циклические конструкции, можно перебрать массив:

|  |  |
| --- | --- |
|  | #include <stdio.h>  int main(void)  {      int numbers[] = { 10, 12, 13, 54, 43 };      size\_t count = sizeof(numbers) / sizeof(numbers[0]);      for(size\_t i =0; i < count; i++)      {          printf("numbers[%zu] = %d \n", i, numbers[i]);      }      return 0;  } |

### **Многомерные массивы**

Массивы могут быть многомерными. Элементы таких массивов сами в свою очередь являются массивами, в которых также элементы могут быть массивами. В большинстве случаев многмерные массивы представляют двухмерные массивы, которые можно представить в виде таблицы. Например, определим двухмерный массив чисел:

|  |  |
| --- | --- |
| 1 | int numbers[3][2] = { {1, 2}, {4, 5}, {7, 8} }; |

Здесь массив numbers имеет три элемента (3 строки), но каждый из этих элементов сам представляет массив из двух элементов (2 столбцов). Такой массив еще можно представить в виде таблицы:

|  |  |
| --- | --- |
| 1 | 2 |
| 4 | 5 |
| 7 | 8 |

И чтобы обратиться к элементам вложенного массива, потребуется два индекса:

|  |  |
| --- | --- |
| 1  2 | int numbers[3][2] = { {1, 2}, {4, 5}, {7, 8} };  printf("numbers[1][0] = %d \n", numbers[1][0]);     // 4 |

другой пример, двухмерный массив с двумя строками и тремя столбцами:

|  |  |
| --- | --- |
| 1 | int numbers[2][3] = { {1, 2, 4}, {5, 7, 8} }; |

### **Строки**

#include <stdio.h>

int main(void)

{

    char message[] = "Hello";

    printf("message: %s \n", message);     // message: Hello

    return 0;

}

char message[] = "Hello";

size\_t length = sizeof(message)/sizeof(char);   // 6 символов

for(size\_t i=0; i<length; i++)

{

    printf("%d ", message[i]);

}

char message[] = {'H', 'e', 'l', 'l', 'o', '\0'};

Для ввода данных в консоли может использоваться функция **scanf()**. Эта функция определена в заголовочном файле **stdio.h** (там же, где и функция printf) и имеет следующее формальное определение:

|  |  |
| --- | --- |
| 1 | scanf(форматная\_строка, аргументы) |

И форматная\_строка, и аргументы для функции scanf обязательны.

Форматная\_строка содержит спецификации преобразования, которые определяют вводимые данные. Общий вид спецификаций преобразования:

|  |  |
| --- | --- |
| 1 | % \* ширина\_поля модификатор спецификатор |

Из этих элементов обязательны только два: знак процента % и спецификатор.

Спецификатор определяет тип вводимых данных:

* **%c**: считывает один символ
* **%d**: считывает десятичное целое число
* **%i**: считывает целое число в любой системе (десятичной, шестнадцатеричной, восьмеричной)
* **%u**: считывает положительное целое число
* **%e**: считывает число с плавающей точкой в экспоненциальной форме
* **%E**: считывает число с плавающей точкой в экспоненциальной форме с заглавным символом экспоненты
* **%f**: считывает число с плавающей точкой
* **%F**: считывает число с плавающей точкой
* **%g**: считывает число с плавающей точкой
* **%G**: считывает число с плавающей точкой
* **%o**: считывает восьмеричное число
* **%x**: считывает шестнадцатеричное число
* **%X**: считывает шестнадцатеричное число
* **%s**: считывает строку
* **%%**: считывает символ процента

Символ звездочки \* в спецификации преобразования позволяет пропустить при вводе водимые символы для типа, указанного через спецификатор.

Ширина\_поля представляет целое положительное число, которое позволяет определить, какое количество байтов будет учитываться при вводе.

Модификаторы позволяют конкретизировать тип данных. В частности, есть следующие модификаторы:

* **h**: для ввода значений типа **short int** (%hd)
* **l**: для ввода значений типа **long int** (%ld) или **double** (%lf, %le)
* **L**: для ввода значений типа **long double** (%Lf, %Le)

В качестве аргументов в функцию **scanf()** передаются адреса переменной, которая будет получать введенное значение. Для получения адреса переменной перед ее именем ставится знак амперсанда **&**. Например, если переменная называется age, то ее адрес мы можем получить с помощью выражения &age.

Например, введем с консоли числовое значение:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int age;      printf("Input your age:");      scanf("%d", &age);        printf("age = %d", age);      return 0;  } |

### Ввод строк

Функция **scanf()** также позволяет вводить строки. Например:

|  |  |
| --- | --- |
|  | #include <stdio.h>   int main(void)  {      char name[10];      // вводим имя      printf("Enter your name: ");      scanf("%10s", name);        printf("Name: %s \n",name);      return 0;  } |

Здесь для имени выделяется 10 символов. Теоретически мы можем ввести и большее количество символов, но чтобы только 10 из них учитывались, в строку форматирования передается ширина поля, которая представляет 10 символов - %10s. Когда функция считает достаточное количетсво символов, она прекратит считывание.

Обратите внимание, что для ввода строки перед названием переменной не указывается символ адреса.

# **Указатели**

### Определение указателя

Для определения указателя надо указать тип объекта, на который указывает указатель, и символ звездочки \*.

|  |  |
| --- | --- |
|  | тип\_данных\* название\_указателя; |

Сначала идет тип данных, на который указывает указатель, и символ звездочки \*. Затем имя указателя.

Например, определим указатель на объект типа int:

|  |  |
| --- | --- |
|  | int \*p; |

Пока указатель не ссылается ни на какой объект. Теперь присвоим ему адрес переменной:

|  |  |
| --- | --- |
|  | int main(void)  {      int x = 10;     // определяем переменную      int \*p;         // определяем указатель      p = &x;         // указатель получает адрес переменной      return 0;  } |

### Получение адреса данных

Указатель хранит адрес объекта в памяти компьютера. И для получения адреса к переменной применяется операция **&**. Эта операция применяется только к таким объектам, которые хранятся в памяти компьютера, то есть к переменным и элементам массива.

Что важно, переменная x имеет тип int, и указатель, который указывает на ее адрес тоже имеет тип int. То есть должно быть соответствие по типу.

Какой именно адрес имеет переменная x? Для вывода значения указателя можно использовать специальный спецификатор **%p**:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int x = 10;      int \*p;      p = &x;      printf("%p \n", p);     // 0060FEA8      return 0;  } |

### **Получение значения по адресу**

Но так как указатель хранит адрес, то мы можем по этому адресу получить хранящееся там значение, то есть значение переменной x. Для этого применяется операция **\*** или операция разыменования (dereference operator). Результатом этой операции всегда является объект, на который указывает указатель. Применим данную операцию и получим значение переменной x:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int x = 10;      int \*p;      p = &x;      printf("Address = %p \n", (void\*) p);      printf("x = %d \n", \*p);      return 0;  } |

Используя полученное значение в результате операции разыменования мы можем присвоить его другой переменной:

|  |  |
| --- | --- |
|  | int x = 10;  int \*p  = &x;  int y = \*p;     // присваиваем переменной y значение по адресу из указателя p  printf("x = %d \n", y); // 10 |

И также используя указатель, мы можем менять значение по адресу, который хранится в указателе:

|  |  |
| --- | --- |
|  | int x = 10;  int \*p = &x;  \*p = 45;  printf("x = %d \n", x);  // 45 |

Так как по адресу, на который указывает указатель, располагается переменная x, то соответственно ее значение изменится.

Создадим еще несколько указателей:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      char c = 'N';      int d = 10;      short s = 2;        char \*pc = &c;          // получаем адрес переменной с типа char      int \*pd = &d;           // получаем адрес переменной d типа int      short \*ps = &s;         // получаем адрес переменной s типа short        printf("Variable c: address=%p \t value=%c \n", (void\*) pc, \*pc);      printf("Variable d: address=%p \t value=%d \n", (void\*) pd, \*pd);      printf("Variable s: address=%p \t value=%hd \n", (void\*) ps, \*ps);      return 0; |

**Операции с указателями**

### Присваивание

Указателю можно присвоить либо адрес объекта того же типа, либо значение другого указателя или константу **NULL**.

Присвоение указателю адреса уже рассматривалось в прошлой теме. Для получения адреса объекта используется операция **&**:

|  |  |
| --- | --- |
|  | int a = 10;  int \*pa = &a;   // указатель pa хранит адрес переменной a |

Причем указатель и переменная должны иметь тот же тип, в данном случае int.

Присвоение указателю другого указателя:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int a = 10;      int b = 2;        int \*pa = &a;      int \*pb = &b;        printf("Variable a: address=%p \t value=%d \n", (void\*)pa, \*pa);      printf("Variable b: address=%p \t value=%d \n", (void\*)pb, \*pb);        pa = pb;    // теперь указатель pa хранит адрес переменной b      printf("Variable b: address=%p \t value=%d \n", (void\*)pa, \*pa);        return 0;  } |

Когда указателю присваивается другой указатель, то фактически первый указатель начинает также указывать на тот же адрес, на который указывает второй указатель.

Если мы не хотим, чтобы указатель указывал на какой-то конкретный адрес, то можно присвоить ему условное нулевое значение с помощью константы **NULL**, которая определена в заголовочном файле stdio.h:

|  |  |
| --- | --- |
| 1 | int \*pa = NULL; |

### **Разыменование указателя**

Операция разыменования указателя в виде \*имя\_указателя, позволяет получить объект по адресу, который хранится в указателе.

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int a = 10;        int \*pa = &a;      int \*pb = pa;        \*pa = 25;        printf("Value on pointer pa: %d \n", \*pa);  // 25      printf("Value on pointer pb: %d \n", \*pb);  // 25      printf("Value of variable a: %d \n", a);    // 25        return 0;  } |

Через выражение \*pa мы можем получить значение по адресу, который хранится в указателе pa, а через выражение типа \*pa = значение вложить по этому адресу новое значение.

И так как в данном случае указатель pa указывает на переменную a, то при изменении значения по адресу, на который указывает указатель, также изменится и значение переменной a.

### **Указатель на void**

Указатели указывают на данные определенных типов. Например, указатель типа int\* указывает на значение типа int, но не может указывать на данные других типов, скажем, на объект типа float. Однако можно также определять указатели типа **void\***, которые могут указывать на данные любого типа. И неявно указатели любых можно преобразовать в указатель типа **void\***:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int x = 123;      int \*ip = &x;       // указатель хранит адрес объекта int      void \*vp;      vp = ip;            // void-указатель получает адрес из указателя ip      printf("Value: %d\n", \*((int \*)vp));    // Value: 123        return 0;  } |

Следует учитывать, что к void-указателю мы НЕ можем применить операцию разыменования и тем самым получить значение под адресу, который хранится в этом указателе. Поэтому для получения значения надо приводить к указателю соответствующего типа:

|  |  |
| --- | --- |
| 1 | printf("Value: %d\n", \*((int \*)vp)); |

Одно из распространенных применений void-указателя - это вывод адреса на консоль:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int x = 123;      int \*ip = &x;                   // указатель хранит адрес объекта int      void \*vp = ip;                  // void-указатель получает адрес из указателя ip      printf("vp: %p\n", vp);         // получаем адрес, который хранится в указателе vp      printf("ip: %p\n", (void\*)ip);  // преобразование к типу void\* - получаем адрес из указателя ip        return 0; |

## Арифметика указателей

int main(void)

{

    double d = 10.6;

    double \*pd = &d;

    printf("Pointer pd: address=%p \n", (void\*)pd);

    pd = pd + 2;    // перемещаем указатель вперед на 2 значения double (2 \* 8 = 16 байт)

    printf("Pointer pd: address=%p \n", (void\*)pd);

    char c = 'N';

    char \*pc = &c;

    printf("Pointer pc: address=%p \n", (void\*)pc);

    pc = pc - 3;        // перемещаем указатель назад на 3 значения char (3 \* 1 = 3 байта)

    printf("Pointer pc: address=%p \n", (void\*)pc);

    return 0;

}

 int main(void)

// {

// int a = 10;

// int \*pn = &a;

// \*pn++; // перемещаем указатель вперед на значения int (1 \* 4 = 4 байт)

// printf("ukaz %p %d", (void\*)pn, \*pn);

// return 0;

// }

Расстояние между указателями представляет тип **ptrdif\_t** - на 64-разрядной архитектуре этот тип является псевдонимом для базового типа **long long** и занимает 8 байт.

Или другой пример:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int arr[64];      int\* ptr1 = &arr[10];      int\* ptr2 = &arr[40];      ptrdiff\_t dist = ptr2 - ptr1; // 30        printf("dist=%lld \n", dist);        return 0;  } |

В данном случае находим разницу между указателями на 10-й и 40-й элементы массива. В итоге расстояние будет равно 30 - 30 элементам int. Чтобы получить расстояние в байтах, соответственно надо умножить расстояние на размер типа указателя.

Но в то же время есть особенности, в частности, с операциями инкремента и декремента. Дело в том, что операции **\***, **++** и **--** имеют одинаковый приоритет и при размещении рядом выполняются справа налево.

Например, выполним постфиксный инкремент:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int a = 10;  int \*pa = &a;  printf("pa: address=%p \t value=%d \n", (void\*)pa, \*pa);  int b = \*pa++;      // инкремент адреса указателя    printf("b: value=%d \n", b);  printf("pa: address=%p \t value=%d \n", (void\*)pa, \*pa); |

В выражении b = \*pa++; сначала к указателю присваивается единица (то есть к адресу добавляется 4, так как указатель типа **int**). Затем так как инкремент постфиксный, с помощью операции разыменования возвращается значение, которое было до инкремента - то есть число 10. И это число 10 присваивается переменной b. И в моем случае результат работы будет следующий:

pa: address=0060FEA4 value=10

b: value=10

pa: address=0060FEA8 value=6356648

Изменим выражение:

|  |  |
| --- | --- |
| 1 | b = (\*pa)++; |

Скобки изменяют порядок операций. Здесь сначала выполняется операция разыменования и получение значения, затем это значение увеличивается на 1. Теперь по адресу в указателе находится число 11. И затем так как инкремент постфиксный, переменная b получает значение, которое было до инкремента, то есть опять число 10. Таким образом, в отличие от предыдущего случая все операции производятся над значением по адресу, который хранит указатель, но не над самим указателем. И, следовательно, изменится результат работы:

pa: address=0060FEA4 value=10

b: value=10

pa: address=0060FEA4 value=11

Аналогично будет с префиксным инкрементом:

|  |  |
| --- | --- |
| 1 | b = ++\*pa; |

В данном случае сначала с помощью операции разыменования получаем значение по адресу из указателя pa, к этому значению прибавляется единица. То есть теперь значение по адресу, который хранится в указателе, равно 11. Затем результат операции присваивается переменной b:

pa: address=0060FEA4 value=10

b: value=11

pa: address=0060FEA4 value=11

Изменим выражение:

|  |  |
| --- | --- |
| 1 | b = \*++pa; |

Теперь сначала изменяет адрес в указателе, затем мы получаем по этому адресу значение и присваиваем его переменной b. Полученное значение в этом случае может быть неопределенным:

pa: address=0060FEA4 value=10

b: value=6356648

pa: address=0060FEA8 value=6356648

### Указатели на константы

Кроме переменных в программе на Си для хранения данных могут использоваться константы, которые предваряются ключевым словом **const**, и указатели также могут указывать на константы, но в этом случае перед определением указателя также ставится слово const:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      const int a = 10;      const int \*pa = &a;      printf("address=%p \t value=%d \n", pa, \*pa);        return 0;  } |

Возможна также ситуация, когда указатель на константу на самом деле указывает на переменную:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int a = 10;      const int \*pa = &a;      printf("value=%d \n", \*pa);     // 10      a = 22;      printf("value=%d \n", \*pa);     // 22      //\*pa = 34;     // так делать нельзя        return 0;  } |

В этом случае переменную отдельно мы сможем изменять, однако по прежнему изменить ее значение через указатель мы не сможем.

Через указатель на константу мы не можем изменять значение переменной/константы. Но мы можем присвоить указателю адрес любой другой переменной или константы:

|  |  |
| --- | --- |
|  | int a = 10;  const int \*pa = &a;     // указатель указывает на переменную a  const int b = 45;  pa = &b;                // указатель указывает на константу |

### Константные указатели

Но также мы можем определять **константные указатели**. Они не могут изменять адрес, который в них хранится, но могут изменять значение по этому адресу.

|  |  |
| --- | --- |
|  | int a = 10;  int \*const pa = &a;  printf("value=%d \n", \*pa);     // 10  \*pa = 22;                       // меняем значение  printf("value=%d \n", \*pa);     // 22    int b = 45;  // pa = &b;         так нельзя сделать |

### Константный указатель на константу

И объединение обоих предыдущих случаев - **константный указатель на константу**, который не позволяет менять ни хранимый в нем адрес, ни значение по этому адресу:

|  |  |
| --- | --- |
|  | int a = 10;  const int \*const pa = &a;    //\*pa = 22;  так сделать нельзя    int b = 45;  // pa = &b;  так сделать нельзя |

## Указатели, массивы и строки

Имя массива без индексов в Си является адресом его первого элемента. Соответственно через операцию разыменования мы можем получить значение по этому адресу:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int array[] = {1, 2, 3, 4, 5};      printf("array[0] = %d", \*array);    // array[0] = 1        return 0;  } |

Прибавляя определенное число к имени массива, мы можем получить указатель на соответствующий элемент массива:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int array[] = {1, 2, 3, 4, 5};      int second = \*(array + 1); // получим второй элемент      printf("array[1] = %d", second);    // array[1] = 2        return 0;  } |

Более того, когда мы в обращаемся к определенному элементу массива, используя квадратные скобки, например:

|  |  |
| --- | --- |
| 1 | array[2] |

компилятор рассмотривает эту запись как прибавление индекса к указателю на начальный элемент:

|  |  |
| --- | --- |
| 1 | array+2 |

Поэтому мы даже можем написать 2[array], что также будет валидным обращением к элементу массива:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | #include <stdio.h>    int main(void)  {      int array[] = {1, 2, 3, 4, 5};      int third = 2[array];      printf("array[2] = %d", third); // array[2] = 3        return 0;  } Использование указателя для работы с массивом Имя массива всегда хранит адрес самого первого элемента, соответственно его можно присвоить другому указателю и затем через указатель обращаться к элеиментам массива:   |  |  | | --- | --- | |  | #include <stdio.h>  int main(void)  {      int array[5] = {1, 2, 3, 4, 5};      int \*ptr = array;   // указатель ptr хранит адрес первого элемента массива array      printf("value: %d \n", \*ptr);   // 1      return 0;  } |   Прибавляя (или вычитая) определенное число от адреса указателя, можно переходить по элементам массива. Например, перейдем к третьему элементу:   |  |  | | --- | --- | |  | #include <stdio.h>  int main(void)  {      int array[5] = {1, 2, 3, 4, 5};      int \*ptr = array;   // указатель ptr хранит адрес первого элемента массива array      ptr = ptr + 2;      // перемезаем указатель на 2 элемента вперед      printf("value: %d \n", \*ptr);   // value: 3      return 0;  } |   Здесь указатель ptr изначально указывает на первый элемент массива. Увеличив указатель на 2, мы пропустим 2 элемента в массиве и перейдем к элементу array[2].  И как и другие данные, можно по указателю изменить значение элемента массива:   |  |  | | --- | --- | |  | #include <stdio.h>    int main(void)  {      int array[5] = {1, 2, 3, 4, 5};      int \*ptr = array;       // указатель ptr хранит адрес первого элемента массива array      ptr = ptr + 2;          // переходим к третьему элементу      \*ptr = 8;               // меняем значение элемента, на который указывает указатель        printf("array[2]: %d \n", array[2]);    // array[2] : 8      return 0;  } |   Стоит отметить, что указатель также может использовать индексы, как и массивы:   |  |  | | --- | --- | |  | #include <stdio.h>    int main(void)  {      int array[5] = {1, 2, 3, 4, 5};      int \*ptr = array;       // указатель ptr хранит адрес первого элемента массива array        int value = ptr[2];     // используем индексы - получаем 3-й элемент (элемент с индексом 2)        printf("value: %d \n", value);  // value: 3      return 0;  } |  Строки и указатели Ранее мы рассмотрели, что строка по сути является набором символов, окончанием которого служит нулевой символ '\0'. И фактически строку можно представить в виде массива:   |  |  | | --- | --- | | 1 | char hello[] = "Hello METANIT.COM!"; |   Но в языке Си также для представления строк можно использовать указатели на тип **char**:   |  |  | | --- | --- | |  | #include <stdio.h>    int main(void)  {      char \*hello = "Hello METANIT.COM!"; // указатель на char - фактически строка      printf("%s", hello);      return 0;  } |   В данном случае оба определения строки - с помощью массива и указателя будут в равнозначны. |

### **Перебор массива с помощью указателей**

С помощью указателей легко перебрать массив:

|  |  |
| --- | --- |
|  | int array[5] = {1, 2, 3, 4, 5};    for(int \*ptr=array; ptr<=&array[4]; ptr++)  {      printf("address=%p \t value=%d \n", (void\*)ptr, \*ptr);  } |

Так как указатель хранит адрес, то мы можем продолжать цикл, пока адрес в указателе не станет равным адресу последнего элемента (ptr<=&array[4]).

Аналогичным образом можно перебрать и многомерный массив:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int array[3][4] = { {1, 2, 3, 4} , {5, 6, 7, 8}, {9, 10, 11, 12}};      int n = sizeof(array)/sizeof(array[0]);         // число строк      int m = sizeof(array[0])/sizeof(array[0][0]);   // число столбцов        int \*final = array[0] + n \* m - 1;  // указатель на самый последний элемент      for(int \*ptr=array[0], i = 1; ptr <= final; ptr++, i++)      {          printf("%d \t", \*ptr);          // если остаток от целочисленного деления равен 0,          // переходим на новую строку          if(i%m==0)          {              printf("\n");          }      }      return 0; |

Мы также могли бы обойтись и без указателя на последний элемент, проверяя значение счетчика, пока оно не станет равно общему количеству элементов (m \* n):

|  |  |
| --- | --- |
|  | for(int \*ptr = array[0], i = 0; i < m\*n;)  {      printf("%d \t", \*ptr++);      if(++i%m==0)      {          printf("\n");      }  } |

### **String interning**

Стоит отметить, что в языке Си для работы со строками применяется такой механизм как **string interning** или **интернирование строк**. В этом случае строки в виде строковых литералов сохраняются в приложении в секции **.rodata** (read-only data), которые предназначены для данных только для чтения, а строковые литералы рассматриваются как неизменяемые данные. Например, возьмем следующую программу:

|  |  |
| --- | --- |
|  | #include <stdio.h>    char\* str1 = "Hello";  char\* str2 = "World";    int main(void)  {      printf("str1 = %p \n", &str1[0]);      printf("str2 = %p \n", &str2[0]);        return 0;  } |

Здесь определены две строки - str1 и str2, в функции main выводим адрес первого символа каждой из этих строк. Так, в моем случае я получу вывод:

str1 = 00007FF75E674000

str2 = 00007FF75E674006

По выводу видно, что первый символ второй строки расположен через 6 байтов начала первой строки. То есть строки расположены в памяти рядом друг с другом.

Но теперь сделаем строки одинаковыми:

|  |  |
| --- | --- |
|  | #include <stdio.h>    char\* str1 = "Hello World";  char\* str2 = "Hello World";    int main(void)  {      printf("str1 = %p \n", &str1[0]);      printf("str2 = %p \n", &str2[0]);        return 0;  } |

Теперь вывод будет другим: Т.е. у них будет один и тот же адрес

str1 = 00007FF75F674000

str2 = 00007FF75F674000

### **Массивы указателей**

Указатели сами по себе представляют значения, которые можно хранить в массивах. То есть в итоге получится массив указателей.

Массив указателей определяется одним из трех способов:

|  |  |
| --- | --- |
|  | тип \*имя\_массива [размер];  тип \*имя\_массива [] = инициализатор;  тип \*имя\_массива [размер] = инициализатор; |

Используем все эти способы:

|  |  |
| --- | --- |
|  | int array[] = {1, 2, 3, 4};  int \*p1[3];  int \*p2[] = { &array[1], &array[2], &array[0] };  int \*p3[3] = { &array[3], &array[1], &array[2] }; |

Массив указателей p1 состоит из трех элементов, но он не инициализирован и является пустым.

Массивы p2 и p3 в качестве элементов хранят адреса на элементы массива a.

Выведем на конслоль значения, на которые ссылаются указатели:

|  |  |  |  |
| --- | --- | --- | --- |
|  | #include <stdio.h>    int main(void)  {      int array[] = {1, 2, 3, 4};      int \*p[] = { &array[1], &array[2], &array[0] };        for(int i = 0; i < 3; i++)      {          printf("%d", \*p[i]);      }      return 0;  }  Вместо \*p[i] мы могли бы написать \*\*(p+i):   * p+i - к адресу в указателе p прибавляем число i и таким образом перемещаемся * по указателям в массиве p.   \*(p+i) - разыменовываем i-тый указатель в массиве и в результате получаем адрес  одного из элементов из массива array.  \*\*(p+i) - получаем значение по полученному на предыдущем шаге адресу элемента   * из массива array.  |  |  | | --- | --- | |  | #include <stdio.h>    int main(void)  {      int array[] = {1, 2, 3, 4};      int \*p[] = { &array[1], &array[2], &array[0] };        for(int i = 0; i < 3; i++)      {          printf("%d", \*\*(p+i));      }      return 0;  } | |

### **Указатель и массив строк**

Соответственно если указатель типа **char** можно представить в виде строки, то массив указателей типа **char** представляет собой массив строк:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      char \*fruit[] = {"apricot", "apple", "banana", "lemon", "orange"};      for(int i=0; i < 5; i++)      {          printf("%s \n", fruit[i]);      }      return 0;  } |

Также мы могли бы написать:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      char \*fruit[] = {"apricot", "apple", "banana", "lemon", "orange"};      for(int i=0; i < 5; i++)      {          printf("%s \n", \*(fruit + i));      }      return 0; |

### **Указатели на указатели**

Кроме обычных указателей в языке Си мы можем создавать указатели на другие указатели. Если указатель хранит адрес переменной, то указатель на указатель хранит адрес указателя, на который он указывает. Такие ситуации еще называются многоуровневой адресацией.

Например:

|  |  |
| --- | --- |
| 1 | int \*\*ptr; |

Переменная ptr представляет указатель на указатель на объект типа **int**. Две звездочки в определении указателя говорят о том, что мы имеем дело с двухуровневой адресацией. Например:

|  |  |
| --- | --- |
|  | #include <stdio.h>    int main(void)  {      int  x = 22;      int \*px = &x;       // указатель px хранит адрес переменной x      int \*\*ppx = &px;    // указатель ppx хранит адрес указателя px        printf("Address of px: %p \n", (void \*)ppx);      printf("Address of x: %p \n", (void \*)\*ppx);      printf("Value of x: %d \n", \*\*ppx);      return 0;  } |