

1.Элементы языка Си. Константы, идентификаторы, ключевые слова

Под элементами языка понимаются его базовые конструкции, используемые при написании программ: алфавит; константы; идентификаторы; ключевые слова; комментарии.

Алфавит C++ включает:

- Строчные и прописные буквы латинского алфавита (мы их будем называть буквами).
- Цифры от 0 до 9 (назовём их буквами-цифрами).
- Символ '_' (подчерк - также считается буквой).
- Набор специальных символов:
" { } , | [] + - % / \ ; ' : ? < > = ! & # ~ ^ . *
- Прочие символы.

Идентификаторы — это имена переменных, однозначно определяющих соответствие переменной с ее адресом. К идентификаторам относят: имена переменных, функций; тэги – имена типов структур, объединений, перечислимых типов. Компилятор языка Си не допускает использования идентификаторов, совпадающих по написанию с ключевыми словами.

Два идентификатора для образования которых используются совпадающие строчные и прописные буквы, считаются различными. Например: abc, ABC, A128B, a128b .

Важной особенностью является то, что компилятор допускает любое количество символов в идентификаторе, хотя значимыми являются первые 31 символ. Идентификатор создается на этапе объявления переменной, функции, структуры и т.п. после этого его можно использовать в последующих операторах разрабатываемой программы. Следует отметить важные особенности при выборе идентификатора.

Во первых, идентификатор не должен совпадать с ключевыми словами, с зарезервированными словами и именами функций библиотеки компилятора языка программирования C.

Во вторых, следует обратить особое внимание на использование символа '_' подчеркивание в качестве первого символа идентификатора, поскольку идентификаторы построенные таким образом, что, с одной стороны, могут совпадать с именами системных функций и (или) переменных, а с другой стороны, при использовании таких идентификаторов программы могут оказаться непереносимыми, т.е. их нельзя использовать на компьютерах других типов.

В третьих, на идентификаторы используемые для определения внешних переменных, должны быть наложены ограничения, формируемые используемым редактором связей (отметим, что использование различных версий редактора связей, или различных редакторов накладывает различные требования на имена внешних переменных).

Ключевые слова — это предопределенные идентификаторы, которые имеют специальное значение для компилятора языка Си. Их можно использовать только так, как они определены. Имя элемента

программы не может совпадать по произношению и написанию с ключевым словом.

В языке C имеются следующие ключевые слова:

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

Комментарии — это последовательность символов, которая воспринимается компилятором языка Си как отдельный пробельный символ и игнорируется. (либо // либо /* */) "Символы" в комментарии могут включать в себя любые комбинации символов представительной таблицы, включая символ новой строки, кроме ограничителя "конец комментария" (*). Комментарии могут занимать более одной строки, но не могут быть вложенными.

Константа — это число, символ или строка символов. Константы используются в программе для задания постоянных величин. В языке Си различают четыре типа констант: целые, с плавающей точкой, символьные константы и символьные строки.

Целые константы — это десятичное, восьмеричное или шестнадцатеричное число, которое представляет целое значение. Между цифрами целой константы пробельные символы недопустимы.

Целые константы всегда специфицируют положительные значения. Если требуется отрицательное значение, то необходимо сформировать константное выражение из знака минус и следующей за ним константы. Знак минус рассматривается при этом как арифметическая операция.

Каждая целая константа имеет тип, определяющий ее представление в памяти, например, int.

Целочисленный литерал служит для записи целочисленных значений и является соответствующей последовательностью цифр (возможно со знаком '-'). Целочисленный литерал, начинающийся с 0, воспринимается как восьмеричное целое. В этом случае цифры 8 и 9 не должны встречаться среди составляющих литерал символов.

Целочисленный литерал, начинающийся с 0x или 0X, воспринимается как шестнадцатеричное целое. В этом случае целочисленный литерал может включать символы от A или a, до F или f, которые в шестнадцатеричной системе эквивалентны десятичным значениям от 10 до 15. Непосредственно за литералом может располагаться в произвольном сочетании один или два специальных суффикса: U (или u), означающий unsigned - беззнаковый; и L (или l), означающий long.

Примеры:

017 – восьмеричное представление

16 – десятичное представление

0x10 – шестнадцатеричное представление

26U

0x33L

(Вещественный литерал) "Константа с плавающей точкой" это десятичное число, которое соответствует действительному числу со знаком. Значение действительного числа со знаком состоит из целой части, дробной части и показателя степени. "Цифр" может не быть или их может быть несколько (от 0 до 9), а E (или e) это символ экспоненты. Можно опустить либо цифры до десятичной точки (целая часть числа) либо после десятичной точки (дробная часть числа), но не одновременно. Если используется показатель степени, то только в этом случае можно не вводить десятичную точку. Показатель степени состоит из символа экспоненты (E или e) за которым следует постоянное целое значение. Целое значение может быть отрицательным. Нельзя использовать разделительные символы между цифрами или символами константы.

Константы с плавающей точкой всегда имеют положительные значения. Если нужно использовать отрицательное значение, то поместите знак минус (-) перед константой для формирования выражения с отрицательным значением. В данном случае знак минус интерпретируется как арифметический оператор. Вещественный литерал служит для отображения вещественных значений. Он фиксирует запись соответствующего значения в обычной десятичной или научной нотациях. В научной нотации мантисса отделяется от порядка литерой E или e).

Все константы с плавающей точкой имеют тип double. Непосредственно за литералом могут располагаться один из двух специальных суффиксов: F (или f), означающий float; и L (или l), означающий long.

Примеры:

10.2 10. 1.3E-3 1.3f 1.3l

· **Символьные константы** — это буква, цифра, знак пунктуации или специальный символ, заключенный в апострофы. Значение символьной константы равно коду представляемого ею символа. Символьные константы имеют тип int. Любая литера может быть представлена в нескольких форматах представления: обычном, восьмеричном и шестнадцатеричном. Допустимый диапазон для обозначения символьных литералов в восьмеричном представлении ограничен восьмеричными числами от 0 до 377. Допустимый диапазон для обозначения символьных литералов в шестнадцатеричном представлении ограничен шестнадцатеричными числами от 0x0 до 0xFF.

Примеры:

'c' 'A' '\127' '\x7F' '\\'

Литеры, которые используются в качестве служебных символов при организации формата представления или не имеют графического представления, могут быть представлены с помощью ещё одного специального формата. Ниже приводится список литер, которые представляются в этом формате. К их числу относятся литеры, не имеющие графического представления, а также литеры, которые используются при организации структуры форматов.

Символ	HEX	Имя	Описание
\0	\x00	null	пустая литера
\a	\x07	bel	сигнал

\b	\x08	bs	возврат на шаг
\f	\x0C	ff	перевод страницы
\n	\x0A	lf	перевод строки
\r	\x0D	cr	возврат каретки
\t	\x09	ht	горизонтальная табуляция
\v	\x0B	vt	вертикальная табуляция
\\	\x5C	\	обратная косая черта (обратный слэш)
\'	\x27	'	
\"	\x22	"	
\?	\x3F	?	

· Строковые литералы являются последовательностью (возможно, пустой) литер в одном из возможных форматов представления, заключённых в двойные кавычки. Строковые литералы, расположенные последовательно, соединяются в один литерал, причём литеры соединённых строк остаются различными. Так, например, последовательность строковых литералов "\xF" "F" после объединения будет содержать две литеры, первая из которых является символьным литералом в шестнадцатеричном формате '\F', второй - символьным литералом 'F'. Строковый литерал и объединённая последовательность строковых литералов заканчиваются пустой литерой, которая используется как индикатор конца литерала.

Примеры:

"String"

"Part1""Part2"

"Part1\13\x0APart2"

Символьные строки — это последовательность символов, заключённая в двойные кавычки. Символьная строка рассматривается как массив символов, каждый элемент которого представляет отдельный символ. Тип символьной строки—массив элементов типа char. Число элементов в массиве равно числу символов в символьной строке плюс один, поскольку нулевой символ (признак конца символьной строки) тоже является элементом массива.

2. Типы данных. Целые и вещественные типы. Перечисляемый тип

Тип данных определяет множество значений, набор операций, которые можно применять к таким значениям и способ реализации хранения значений и выполнения операций.

Процесс проверки и накладывания ограничений на типы используемых данных называется контролем типов или **типизацией программных данных**. Различают следующие виды типизации:

- Статическая типизация — контроль типов осуществляется при компиляции.
- Динамическая типизация — контроль типов осуществляется во время выполнения.

Язык Си поддерживает статическую типизацию, и типы всех используемых в программе данных должны быть указаны перед ее компиляцией.

Различают простые, составные и прочие типы данных.

Простые данные

Простые данные можно разделить на

- целочисленные,
- вещественные,
- символьные
- логические.

Составные (сложные) данные

- Массив — индексированный набор элементов одного типа.
- Строковый тип — массив, хранящий строку символов.
- Структура — набор различных элементов (полей записи), хранимый как единое целое и предусматривающий доступ к отдельным полям структуры.

Другие типы данных

- Указатель — хранит адрес в памяти компьютера, указывающий на какую-либо информацию, как правило — указатель на переменную.

Программа, написанная на языке Си, оперирует с данными различных типов. Все данные имеют имя и тип. Обращение к данным в программе осуществляется по их именам (идентификаторам).

Целочисленные данные

Целочисленные данные могут быть представлены в знаковой и беззнаковой форме.

Беззнаковые целые числа представляются в виде последовательности битов в диапазоне от 0 до 2^n-1 , где n – количество занимаемых битов.

Знаковые целые числа представляются в диапазоне $-2^{n-1} \dots +2^{n-1}-1$. При этом старший бит данного отводится под знак числа (0 соответствует положительному числу, 1 – отрицательному).

Основные типы и размеры целочисленных данных:

Количество бит	Беззнаковый тип	Знаковый тип
8	unsigned char 0...255	char -128...127
16	unsigned short 0...65535	short -32768...32767
32	unsigned int	int
64	unsigned long int	long int

Вещественные данные

Вещественный тип предназначен для представления действительных чисел. Вещественные числа представляются в

Различают три основных типа представления вещественных чисел в языке Си:

Тип	Обозначение в Си	Кол-во бит	Биты степени	Мантисса	Сдвиг
-----	------------------	------------	--------------	----------	-------

Тип	Обозначение в Си	Кол-во бит	Биты степени	Мантисса	Сдвиг
простое	float	32	30...23	22...0	127
двойной точности	double	64	62...52	51...0	1023
двойной расширенной точности	long double	80	78...64	62...0	16383

Как видно из таблицы, бит целое у типов float и double отсутствует. При этом диапазон представления вещественного числа состоит из двух диапазонов, расположенных симметрично относительно нуля.

Символьный тип

Символьный тип хранит код символа и используется для отображения символов в различных кодировках. Символьные данные задаются в кодах и по сути представляют собой целочисленные значения. Для хранения кодов символов в языке Си используется тип char.

Логический тип

Логический тип имеет применяется в логических операциях, используется при алгоритмических проверках условий и в циклах и имеет два значения:

- истина — true
- ложь — false

В программе должно быть дано объявление всех используемых данных с указанием их имени и типа. Описание данных должно предшествовать их использованию в программе.

Пример объявления объектов

```
int n; // Переменная n целого типа
double a; // Переменная a вещественного типа двойной точности
```

Перечисляемый тип (сокращённо **перечисление**, [англ.](#) *enumeration, enumerated type*) — в [программировании тип данных](#), чьё множество значений представляет собой ограниченный список идентификаторов.

enum - перечислимый тип.

Перечисления используются для объявления символических имен, которые являются целочисленными константами. То есть типом перечисления, по факту

является **целочисленным типом**, и эти константы можно использоваться везде где можно использовать целочисленные типы.

Всего выделен отдельный тип перечисление (enum), задающий набор всех возможных целочисленных значений переменной этого типа. Синтаксис перечисления

[?](#)

```
enum <имя> {  
    <имя поля 1>,  
    <имя поля 2>,  
    ...  
    <имя поля N>  
}; //здесь стоит ;!
```

Например

[?](#)

```
1  
    #include <conio.h>  
2  
    #include <stdio.h>  
3  
4  
    enum Gender {  
5  
        MALE,  
6  
        FEMALE  
7    };  
8  
9    void main() {  
10        enum Gender a, b;  
11        a = MALE;  
12        b = FEMALE;  
13        printf("a = %d\n", a);  
14        printf("b = %d\n", b);  
15        getch();  
16    }
```

В этой программе объявлено перечисление с именем Gender. Переменная типа enum Gender может принимать теперь только два значения – это MALE И FEMALE.

По умолчанию, первое поле структуры принимает численное значение 0, следующее 1, следующее 2 и т.д. Можно задать нулевое значение явно:

[?](#)


```

1
2  #include <conio.h>
3  #include <stdio.h>
4
5  enum Token {
6      SYMBOL,          //0
7      NUMBER,          //1
8      EXPRESSION = 0, //0
9      OPERATOR,        //1
10     UNDEFINED         //2
11 };
12
13 void main() {
14     enum Token a, b, c, d, e;
15     a = SYMBOL;
16     b = NUMBER;
17     c = EXPRESSION;
18     d = OPERATOR;
19     e = UNDEFINED;
20     printf("a = %d\n", a);
21     printf("b = %d\n", b);
22     printf("c = %d\n", c);
23     printf("d = %d\n", d);
24     printf("e = %d\n", e);
25     getch();
26 }

```

Будут выведены значения 0 1 0 1 2. То есть, значение SYMBOL равно значению EXPRESSION, а NUMBER равно OPERATOR. Если мы изменим программу и напомним

[2](#)

```

4  enum Token {
5      SYMBOL,          //0
6      NUMBER,          //1
7      EXPRESSION = 10, //10

```

```

7      OPERATOR,          //11
8      UNDEFINED          //12
9  };
10

```

То SYMBOL будет равно значению 0, NUMBER равно 1, EXPRESSION равно 10, OPERATOR равно 11, UNDEFINED равно 12.

Принято писать имена полей перечисления, как и константы, заглавными буквами. Так как поля перечисления целого типа, то они могут быть использованы в операторе switch.

Заметьте, что мы не можем присвоить переменной типа Token просто численное значение. Переменная является сущностью типа Token и принимает только значения полей перечисления. Тем не менее, переменной числу можно присвоить значение поля перечисления.

Обычно перечисления используются в качестве набора именованных констант. Часто поступают следующим образом - создают массив строк, ассоциированных с полями перечисления. Например

```

2
1  #include <conio.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  static char *ErrorNames[] = {
6      "Index Out Of Bounds",
7      "Stack Overflow",
8      "Stack Underflow",
9      "Out of Memory"
10 };
11
12 enum Errors {
13     INDEX_OUT_OF_BOUNDS = 1,
14     STACK_OVERFLOW,
15     STACK_UNDERFLOW,
16     OUT_OF_MEMORY
17 };
18
19 void main() {
20     //ошибка случилась
21     printf(ErrorNames[INDEX_OUT_OF_BOUNDS-1]);

```

```
20         exit (INDEX_OUT_OF_BOUNDS);  
21     }  
22  
23
```

Так как поля принимают численные значения, то они могут использоваться в качестве индекса массива строк. Команда `exit(N)` должна получать код ошибки, отличный от нуля, потому что 0 - это плановое завершение без ошибки. Именно поэтому первое поле перечисления равно единице.

Перечисления используются для большей типобезопасности и ограничения возможных значений переменной. Для того, чтобы не писать `enum` каждый раз, можно объявить новый тип. Делается это также, как и в случае структур.

[2](#)

```
typedef enum enumName {  
  
    FIELD1,  
  
    FIELD2  
  
} Name;
```

Например

[2](#)

```
typedef enum Bool {  
  
    FALSE,  
  
    TRUE  
  
} Bool;
```

3.Указатели. Операции разадресации и адреса. Адресная арифметика

Указатели представляют собой объекты, значением которых служат адреса других объектов (переменных, констант, указателей) или функций. Указатели - это неотъемлемый компонент для управления памятью в языке Си.

Для определения указателя надо указать тип объекта, на который указывает указатель, и символ звездочки *. Например, определим указатель на объект типа int:

```
1 int *p;
```

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

```
1 int x = 10;      // определяем переменную
2 int *p;          // определяем указатель
3 p = &x;          // указатель получает адрес переменной
```

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**:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int x = 10;
6     int *p;
7     p = &x;
8     printf("%p \n", p);      // 0060FEA8
9     return 0;
10 }
```

В моем случае машинный адрес переменной `x` - 0060FEA8. Но в каждом отдельном случае адрес может быть иным. Фактически адрес представляет целочисленное значение, выраженное в шестнадцатеричном формате.

То есть в памяти компьютера есть адрес 0x0060FEA8, по которому располагается переменная `x`. Так как переменная `x` представляет тип **int**, то на большинстве архитектур она будет занимать следующие 4 байта (на конкретных архитектурах размер памяти для типа `int` может отличаться). Таким образом, переменная типа `int` последовательно займет ячейки памяти с адресами 0x0060FEA8, 0x0060FEA9, 0x0060FEAA, 0x0060FEAB.

И указатель **p** будет ссылаться на адрес, по которому располагается переменная `x`, то есть на адрес 0x0060FEA8.

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, то есть та операция, которая применяется при определении указателя. Результатом этой операции всегда является объект, на который указывает указатель. Применим данную операцию и получим значение переменной `x`:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int x = 10;
6      int *p;
7      p = &x;
8      printf("Address = %p \n", p);
9      printf("x = %d \n", *p);
10     return 0;
11 }
```

Консольный вывод:

```
Address = 0060FEA8
x = 10
```

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

```

1 int x = 10;
2 int *p = &x;
3 int y = *p;
4 printf("x = %d \n", y); // 10

```

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

```

1 int x = 10;
2 int *p = &x;
3 *p = 45;
4 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:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     char c = 'N';
6     int d = 10;
7     short s = 2;
8
9     char *pc = &c;           // получаем адрес переменной c типа char
10    int *pd = &d;             // получаем адрес переменной d типа int
11    short *ps = &s;           // получаем адрес переменной s типа short
12
13    printf("Variable c: address=%p \t value=%c \n", pc, *pc);
14    printf("Variable d: address=%p \t value=%d \n", pd, *pd);
15    printf("Variable s: address=%p \t value=%hd \n", ps, *ps);
16    return 0;
17 }

```

Указатели в языке Си поддерживают ряд операций: присваивание, получение адреса указателя, получение значения по указателю, некоторые арифметические операции и операции сравнения.

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 **&**:

```
1  int a = 10;
2  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:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a = 10;
6      int b = 2;
7
8      int *pa = &a;
9      int *pb = &b;
10
11     printf("Variable a: address=%p \t value=%d \n", pa, *pa);
12     printf("Variable b: address=%p \t value=%d \n", pb, *pb);
13
14     pa = pb;    // теперь указатель pa хранит адрес переменной b
15     printf("Variable b: address=%p \t value=%d \n", pa, *pa);
16
17     return 0;
18 }
```

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.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = 10;
6
7     int *pa = &a;
8     int *pb = pa;
9
10    *pa = 25;
11
12    printf("Value on pointer pa: %d \n", *pa); // 25
13    printf("Value on pointer pb: %d \n", *pb); // 25
14    printf("Value of variable a: %d \n", a);   // 25
15
16    return 0;
17 }
```

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`.

Адрес указателя

Указатель хранит адрес переменной, и по этому адресу мы можем получить значение этой переменной. Но кроме того, указатель, как и любая переменная, сам имеет адрес, по которому он располагается в памяти. Этот адрес можно получить также через операцию **&**:

```
1 int a = 10;
2 int *pa = &a;
3 printf("address of pointer=%p \n", &pa);          // адрес указателя
4 printf("address stored in pointer=%p \n", pa);    // адрес, который хранится в
5 printf("value on pointer=%d \n", *pa);           // значение по адресу в указателе
   - значение переменной a
```

Операции сравнения

К указателям могут применяться операции сравнения **>**, **>=**, **<**, **<=**, **==**, **!=**. Операции сравнения применяются только к указателям одного типа и константе **NULL**. Для сравнения используются номера адресов:

```
1 int a = 10;
2 int b = 20;
3 int *pa = &a;
4 int *pb = &b;
5 if(pa > pb)
6     printf("pa (%p) is greater than pb (%p) \n", pa, pb);
7 else
8     printf("pa (%p) is less or equal pb (%p) \n", pa, pb);
```

Консольный вывод в моем случае:

```
pa (0060FEA4) is greater than pb (0060FEA0)
```

Приведение типов

Иногда требуется присвоить указателю одного типа значение указателя другого типа. В этом случае следует выполнить операцию приведения типов:

```
1 char c = 'N';
2 char *pc = &c;
3 int *pd = (int *)pc;
```

4	<code>printf("pc=%p \n", pc);</code>
5	<code>printf("pd=%p \n", pd);</code>

Эти операции используются для работы с переменными типа указатель.

Операция разадресации (*) осуществляет косвенный доступ к адресуемой величине через указатель. Операнд должен быть указателем. Результатом операции является величина, на которую указывает операнд. Типом результата является тип величины, адресуемой указателем. Результат не определен, если указатель содержит недопустимый адрес.

Рассмотрим типичные ситуации, когда указатель содержит недопустимый адрес:

- указатель является нулевым;
 - указатель определяет адрес такого объекта, который не является активным в момент ссылки;
 - указатель определяет адрес, который не выровнен до типа объекта, на который он указывает;
 - указатель определяет адрес, не используемый выполняющейся программой.
- Операция адрес (&) дает адрес своего операнда. Операндом может быть любое именованное выражение. Имя функции или массива также может быть операндом операции «адрес», хотя в этом случае знак операции является лишним, так как имена массивов и функций являются адресами. Результатом операции адрес является указатель на операнд. Тип, адресуемый указателем, является типом операнда.

Операция адрес не может применяться к элементам структуры, являющимися полями битов, и к объектам с классом памяти register.

Под адресной арифметикой понимаются действия над указателями, связанные с использованием адресов памяти. Рассмотрим операции, которые можно применять к указателям, попутно заметив, что некоторые из них уже рассматривались, однако здесь мы повторимся, для систематизации изложения материала.

1. Присваивание. Указателю можно присвоить значение адреса. Любое число, присвоенное указателю, трактуется как адрес памяти:

```
int *u, *adr;
int N;
u = &N;           // указателю присвоен адрес переменной N
adr = (int *)0x00FD; // указателю присвоен 16-теричный адрес,
                  // не рекомендуется так делать
```

2. Взятие адреса. Так как указатель является переменной, то для получения адреса памяти, где расположен указатель, можно использовать операцию **взятия адреса &**:

```
int **a, *b;
a = &b; // указателю a присвоен адрес указателя b
```

3. Косвенная адресация. Для того, чтобы получить значение, хранящееся по адресу, на который ссылается указатель, или послать данное по адресу, используется операция **косвенной адресации ***:

```
int *uk;
int n;
int m = 5;
uk = &m; // uk присвоен адрес переменной m
n = *uk; // переменная n примет значение 5
*uk = -13; // переменная m примет значение -13
```

4. Преобразование типа. Указатель на объект одного типа может быть преобразован в указатель на другой тип. При этом следует учитывать, что объект, адресуемый преобразованным указателем, будет

интерпретироваться по-другому. Операция преобразования типа указателя применяется в виде (**<тип>*<указатель>**):

```
int i, *ptr;
i = 0x8e41;
ptr = &i;
printf("%d\n", *ptr); // печатается значение int:-29119 (0x8e41)
printf("%d\n", *((char *)ptr)); // ptr преобразован к типу char,
                                // извлекается 1 байт и его двоичный
код
                                // печатается в виде десятичного
числа,
                                // печатается: 65 */
```

Преобразование типа указателя чаще всего применяется для приведения указателя на неопределенный тип данных void к типу объекта, доступ к которому будет осуществляться через этот указатель.

5. Определение размера. Для определения размера указателя можно использовать операцию размер в виде **sizeof(<указатель>)**. Размер памяти, отводимой компилятором под указатель, зависит от модели памяти. Для близких указателей операция **sizeof** дает значение **2**, для дальних **4**.

6. Сравнение. Сравнение двух указателей любой из операций отношения имеет смысл только в том случае, если оба указателя адресуют общий для них объект, например, строку или массив.

7. Индексация. Указатель может индексироваться применением к нему операции индексации, обозначаемой в Си квадратными скобками **[]**. Индексация указателя имеет вид **<указатель>[<индекс>]**, где **<индекс>** записывается целочисленным выражением.

Возвращаемым значением операции индексации является данное, находящееся по адресу, смещенному в большую или меньшую сторону относительно адреса, содержащегося в указателе в момент применения операции. Этот адрес определяется так: **(адрес в указателе) + (значение <индекс>) * sizeof(<тип>)**, где **<тип>** – это тип указателя.

Из этого адреса извлекается или в этот адрес посылается, в зависимости от контекста применения операции, данное, тип которого интерпретируется в соответствии с типом указателя. Рассмотрим следующий пример:

```
int *uk1;
int b, k;
uk1 = &b; // в uk1 адрес переменной b
k = 3;
b = uk1[k]; // переменной b присваивается значение int,
            // взятое из адреса на 6 большего, чем
            // адрес переменной b; в uk1 адрес не изменился
uk1[k] = -14; // в адрес на 6 больший, чем адрес переменной b,
            // записывается -14
```

Операция индексации не изменяет значение указателя, к которому она применялась.

8. Увеличение/уменьшение. Если к указателю применяется операция увеличения **++** или уменьшения **--**, то значение указателя увеличивается или уменьшается на размер объекта, который он адресует:

```
long b; // b - переменная типа int длиной 4 байта
long *ptr; // ptr - указатель на объект int длиной 4 байта
ptr = &b; // в ptr адрес переменной b
ptr++; // в ptr адрес увеличился на 4
ptr--; // в ptr адрес уменьшился на 4
```

9. Сложение. Одним из операндов операции сложения может быть указатель, а другим операндом обязательно должно быть выражение целого типа. Операция сложения вырабатывает адрес, который определяется следующим образом: **(адрес в указателе) + (значение int_выражения)*sizeof(<тип>)**, где **<тип>** это тип данных, на которые ссылается указатель.

```
double d;
int n;
double *uk;
uk = &d; // в uk адрес переменной d
n = 3;
uk = uk+n; // в результате выполнения операции сложения,
           // а затем операции присваивания, в uk новый
           // адрес на 24 больше, чем предыдущий
uk=uk+n; // в uk адрес увеличился еще на 24
```

10. Вычитание. Левым операндом операции вычитания должен быть указатель, а правым должно быть выражение целого типа. Операция вычитания вырабатывает адрес, который определяется так: **(адрес в указателе) - (значение int_выражения)*sizeof(<тип>)**.

К указателям можно применять только описанные операции и операции, которые выражаются через них, например, разрешается к указателю применить операцию **uk += n;**, так как ее можно выразить через **uk = uk+n;**. Следующие операции недопустимы с указателями:

- сложение двух указателей;
- вычитание двух указателей на различные объекты;
- сложение указателей с числом с плавающей точкой;
- вычитание из указателей числа с плавающей точкой;
- умножение указателей;
- деление указателей;
- поразрядные операции и операции сдвига;

4.Выражения. Операнды и операции (унарные, бинарные, тернарные). Правила преобразования типов.

Комбинация знаков операций и операндов, результатом которой является определенное значение, называется **выражением**. Знаки операций определяют действия, которые должны быть выполнены над операндами. Каждый операнд в выражении может быть выражением. Значение выражения зависит от расположения знаков операций и круглых скобок в выражении, а также от приоритета выполнения операций.

Операнд - это константа, литерал, идентификатор, вызов функции, индексное выражение, выражение выбора элемента или более сложное выражение, сформированное комбинацией операндов, знаков операций и круглых скобок. Любой операнд, который имеет константное значение, называется константным выражением. Каждый операнд имеет тип.

Операции. По количеству операндов, участвующих в операции, операции подразделяются на **унарные, бинарные и тернарные**.

Унарное выражение состоит из операнда и предшествующего ему знаку унарной операции и имеет следующий формат:

ЗнакУнарнойОперации операнд

В языке Си имеются следующие **унарные операции**:

-	арифметическое отрицание (отрицание и дополнение);
~	побитовое логическое отрицание (дополнение);
!	логическое отрицание;
*	разадресация (косвенная адресация);
&	вычисление адреса;
+	унарный плюс;
++	увеличение (инкремент);
--	уменьшение (декремент);
sizeof	Размер

Операции увеличения и уменьшения увеличивают или уменьшают значение операнда на единицу и могут быть записаны как справа так и слева от операнда. Если знак операции записан перед операндом (*префиксная форма*), то изменение операнда происходит до его использования в выражении. Если знак операции записан после операнда (*постфиксная форма*), то операнд вначале используется в выражении, а затем происходит его изменение.

Бинарное выражения состоит из двух операндов, разделенных знаком бинарной операции:

операнд1 ЗнакБинарнойОперации операнд2

Бинарные операции

Знак операции	Операция	Группа операций
*	Умножение	Мультипликативные
/	Деление	
%	Остаток от деления	
+	Сложение	Аддитивные
-	Вычитание	
<<	Сдвиг влево	Операции сдвига
>>	Сдвиг вправо	
<	Меньше	Операции отношения
<=	Меньше или равно	
>=	Больше или равно	
==	Равно	
!=	Не равно	
&	Поразрядное И	Поразрядные операции
	Поразрядное ИЛИ	
^	Поразрядное исключающее ИЛИ	
&&	Логическое И	Логические операции
	Логическое ИЛИ	
,	Последовательное вычисление	Последовательного вычисления
=	Присваивание	Операции присваивания
*=	Умножение с присваиванием	
/=	Деление с присваиванием	
%=	Остаток от деления с присваиванием	
-=	Вычитание с присваиванием	

+=	Сложение с присваиванием	
<<=	Сдвиг влево с присваиванием	
>>=	Сдвиг вправо присваиванием	
&=	Поразрядное И с присваиванием	
=	Поразрядное ИЛИ с присваиванием	
^=	Поразрядное исключающее ИЛИ с присваиванием	

При записи выражений следует помнить, что символы (*), (&), (!), (+) могут обозначать унарную или бинарную операцию.

Тернарное выражение состоит из трех операндов, разделенных знаками тернарной операции (?) и (:), и имеет формат:

операнд1 ? операнд2 : операнд3

Правила преобразования типов:

1. Если операция выполняется над данными различных типов, то *оба данных приводятся к высшему из двух типов*. Это преобразование называется повышением типа. В случае, когда используется данные одного типа, то никаких преобразований не производится.
2. Последовательность типов, упорядоченных от высшего к низшему определена в соответствии с внутренним представлением данных и выглядит так:

double «— float
unsigned int «— unsigned short «— unsigned char
int «— short «— char
3. Для операции присваивания (простой или составной) результат вычисления выражения правой части приводится к типу переменной, которой должно быть присвоено это значение. При этом может произойти повышение типа, либо его понижение. Повышение типа при присваивании обычно происходит нормально, в то время, как понижение может существенно *исказить* результат из-за того, что данное высшего типа не может поместиться в области памяти данного низшего типа.
4. Для сохранения точности вычисления при выполнении арифметических операций все данные типа float преобразуются в тип double, что существенно уменьшает ошибку округления. Конечный результат преобразуется обратно в тип float, если это обусловлено соответствующим оператором описания.

5.Операторы языка Си. Оператор выражение, составной оператор, операторы условного перехода. Понятие false и true

Основу программы на Си составляют выражения, а не операторы. Большинство операторов в программе являются выражениями с ';'. Это позволяет создавать эффективные программы.

Оператор является законченной конструкцией языка Си. Операторы служат основными конструкциями при построении программы. Выражение состоит из операций и операндов (операнд – то, над чем выполняется операция, простейшее выражение может состоять из одного операнда). Оператор служит командой компьютеру. Операторы бывают простые и составные. Простые операторы оканчиваются « ; ».

Простые операторы:

1. Пустой оператор ';' ;
2. Оператор описания
int x, y;
3. Оператор присвоения
count = 0.0;
4. Оператор выражение (управляющий оператор)
sum = sum + count;
var = (var + 10) / 4;
5. Оператор вызова функции
printf(«Привет \n»);

Составные операторы или блоки – это группа операторов, заключенных в фигурные скобки {...}

В результате выполнения в программе **оператора-выражения** вычисляется значение выражения в соответствии с теми операциями, которые в нем определены. Чаще всего в выражении имеется одна или несколько операций присваивания, и тогда оператор-выражение Си имеет тот же смысл, что и оператор присваивания в других языках программирования

```
int x, y, z;  
x = -3 + 4 * 5 - 6; // = 11  
y = 3 + 4 % 5 - 6; // = 1  
z = -3 * 4 % - 6 / 5; // = 0
```

Оператор if

Формат оператора:

*if (выражение) оператор-1;
[else оператор-2;]*

Выполнение оператора if начинается с вычисления выражения.

Далее выполнение осуществляется по следующей схеме:

- если выражение истинно (т.е. отлично от 0), то выполняется оператор-1.
- если выражение ложно (т.е. равно 0), то выполняется оператор-2.
- если выражение ложно и отсутствует оператор-2 (в квадратные скобки заключена необязательная конструкция), то выполняется следующий за if оператор.

После выполнения оператора if значение передается на следующий оператор программы.

Допускается использование вложенных операторов if. Оператор if может быть включен в конструкцию if или в конструкцию else другого оператора if. Чтобы сделать программу более читабельной, рекомендуется группировать операторы и конструкции во вложенных операторах if, используя фигурные скобки. Если же фигурные скобки опущены, то компилятор связывает каждое ключевое слово else с наиболее близким if, для которого нет else.

True & False (пишу от себя)

Как такого типа в котором было бы 2 значения, true и false, в языке си нет. Для этого зачастую используются следующие значения:

- *0 – false*
- *!0 – true* (обычно *!0 = 1*, но в исключительных случаях это может быть любое другое целое число).

6. Организация циклических вычислительных процессов с помощью операторов for, while, do while

Оператор for

Оператор for - это наиболее общий способ организации цикла. Он имеет следующий формат:

for (выражение 1 ; выражение 2 ; выражение 3) тело

Выражение 1 обычно используется для установления начального значения переменных, управляющих циклом. *Выражение 2* - это выражение, определяющее условие, при котором тело цикла будет выполняться. *Выражение 3* определяет изменение переменных, управляющих циклом после каждого выполнения тела цикла.

Схема выполнения оператора for:

- Вычисляется выражение 1.
- Вычисляется выражение 2.
- Если значения выражения 2 отлично от нуля (истина), выполняется тело цикла, вычисляется выражение 3 и осуществляется переход к пункту 2, если выражение 2 равно нулю (ложь), то управление передается на оператор, следующий за оператором for.

Существенно то, что проверка условия всегда выполняется в начале цикла. Это значит, что тело цикла может ни разу не выполниться, если условие выполнения сразу будет ложным.

Пример:

```
int main()
{
    int i,b;

    for (i=1; i<10; i++)

        b=i*i;

    return 0;
}
```

В этом примере вычисляются квадраты чисел от 1 до 9.

Другим вариантом использования оператора for является бесконечный цикл. Для организации такого цикла можно использовать пустое условное выражение, а для выхода из цикла обычно используют дополнительное условие и оператор break.

Пример:

```
for ( ; ; ){  
    ...  
    ... break;  
    ...  
}
```

Так как согласно синтаксису языка Си оператор может быть пустым, тело оператора for также может быть пустым. Такая форма оператора может быть использована для организации поиска.

Пример:

```
for (i=0; t[i]<10 ; i++) ;
```

В данном примере переменная цикла i принимает значение номера первого элемента массива t, значение которого больше 10.

Оператор цикла while

Оператор цикла while называется циклом с предусловием и имеет следующий формат:

while (выражение) тело ;

В качестве выражения допускается использовать любое выражение языка Си, а в качестве тела любой оператор, в том числе пустой или составной. Схема выполнения оператора while следующая:

- Вычисляется выражение.
- Если выражение ложно, то выполнение оператора while заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполняется тело оператора while.
- Процесс повторяется с пункта 1.

Оператор цикла вида

for (выражение-1; выражение-2; выражение-3) тело ;

может быть заменен оператором while следующим образом:

выражение-1;

```
while (выражение-2){  
    тело  
    выражение-3;  
}
```

Так же как и при выполнении оператора for, в операторе while вначале происходит проверка условия. Поэтому оператор while удобно использовать в ситуациях, когда тело оператора не всегда нужно выполнять.

Оператор do while

Оператор цикла do while называется оператором цикла с постусловием и используется в тех случаях, когда необходимо выполнить тело цикла хотя бы один раз. Формат оператора имеет следующий вид:

do {тело} while (выражение);

- Схема выполнения оператора do while :
- Выполняется тело цикла (которое может быть составным оператором).
- Вычисляется выражение.
- Если выражение ложно, то выполнение оператора do while заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполнение оператора продолжается с пункта 1.

Чтобы прервать выполнение цикла до того, как условие станет ложным, можно использовать оператор break.

7. Операторы `go to` и `switch()`

1. Переключатель `Switch`:

Функция: используется для выбора одного из нескольких возможных вариантов. Проверяет совпадает ли значение выражения с одной из заданных констант (целочисленных) или выражений и в случае совпадения выполняет соответствующую ветвь программы.

Строение:

```
switch (выражение) {  
    _____ case константа-выражение: инструкции  
    _____ case константа-выражение: инструкции  
    _____ default: инструкции  
}
```

Примечание: 1. Константы выражения каждого `case` должны быть отличны друг от друга; 2. Если ни одна константа-выражение не совпадает со значением выражения, то выполняется ф-я `default`, если такова имеется, в противном случае ничего не делается; 3. Ветви `case` и `default` могут располагаться в любом порядке друг относительно друга; 4. Если в программе используются несколько групп ветвей `case`, то по завершении каждой группы необходимо выйти из переключателя при помощи инструкции `break`.

Пример для примечания 4:

```
switch(выражение) {  
    _____ case константа-выражение:  
    _____ case константа-выражение:  
    _____ инструкции //если инструкция одинакова для всех case, её можно вынести  
    _____ break  
    _____ case константа-выражение:  
    _____ case константа-выражение:  
    _____ инструкции;  
    _____ break  
    _____ default: инструкции;  
    _____ break // после default также рекомендуется вставлять break, в случае необходимости  
    _____ добавления новых групп case.
```

2. Инструкция `goto` и метки:

Функция: используется преимущественно в сложных вложенных циклах для быстрого прекращения их работы, выхода из них и перехода к другой функции, если последнюю удобнее оформить вне этих вложенных циклов. *goto* как ссылка (на метку).

Структура:

```
for(...){  
    _____for(...){  
        _____  
        _____if(error=1)_____/*Если в цикле ошибка, то  
        _____goto(healing);_____перейти на ф-ю исправления ошибок*/  
    }  
}  
  
healing:_____//это метка; её структура: «имя переменной:»  
_____алгоритм для разрешения проблемы
```

Примечание: как вы поняли из комментария, есть ф-я *goto*, которая позволяет перейти на ф-ю *healing*, причем последняя называется «меткой» и соответствующе оформляется. ВАЖНО: метка видима на протяжении всей **функции** (не программы), то есть *goto* может ссылаться только на те метки, которые находятся с ним в одной ф-и.

8. Оператор `return`

Функция: завершает выполнение функции и возвращает управление вызывающей функции. Выполнение возобновляется в вызывающей функции в точке сразу после вызова. Оператор `return` также может возвращать значение, передавая его вызывающей функции.

Структура:

`return(выражение)`

Примечание:

Параметр "выражение", если он присутствует, вычисляется и преобразуется к типу, возвращаемому функцией. Если функция была объявлена с типом возвращаемого значения `void`, то оператор `return`, содержащий выражение, выводит предупреждение, и выражение не вычисляется.

Если в теле функции оператор `return` не указан, то после выполнения последнего оператора вызванной функции управление автоматически возвращается вызывающей функции. В этом случае возвращаемое значение вызванной функции не определено. Если возвращаемое значение не требуется, функция объявляется с возвращаемым типом `void`; в противном случае возвращаемое значение имеет тип по умолчанию `int`.

9. Перечисляемый тип в языке Си.

Определение:

Перечисляемый тип (англ. ENUMeration type) — тип данных, чьё множество значений представляет собой ограниченный список идентификаторов.

Пример:

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

```
enum months { JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
```

```
enum months summer;
```

Примечание:

Здесь объявлено перечисление с именем *months*. Переменная типа *enum months* может принимать теперь только значения *JAN, FEB... DEC*. То есть, выражения *summer = AUG* и *summer=DEC* будут корректны, а *summer=sun* — нет.

Первое имя в *enum* имеет значение 0, второе — 1 и т. д., однако можно присвоить некое значение, после чего продолжится прогрессия (*JAN=1*, поэтому *FEB=2*). Это можно делать с любым именем, не только с первым. Тогда, например, в случае:

```
enum months { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP=100, OCT, NOV, DEC };
```

имена примут значения:
JAN=0, FEB=1, MAR=2, ..., AUG=7, SEP=100, OCT=101, NOV=102, DEC=103;

Имена в различных перечисляемых типах должны отличаться друг от друга. Значения внутри одного типа могут совпадать.

7. Организация ввода-вывода в языке Си.

Форматный ввод-вывод

Описание:

Функция *printf* преобразует, форматирует и печатает свои аргументы в стандартном выводе под управлением формата. Возвращает она количество напечатанных символов.

Форматная строка содержит два вида объектов: 1) обычные символы, которые напрямую копируются в выходной поток, и 2) спецификации преобразования, каждая из которых вызывает преобразование и печать очередного аргумента *printf*. Любая спецификация преобразования начинается знаком % и заканчивается *символом-спецификатором*. Между % и *символом-спецификатором* могут быть расположены (в указанном ниже порядке) следующие элементы:

1. Знак минус, предписывающий выравнивать преобразованный аргумент по левому краю поля.
2. Число, специфицирующее минимальную ширину поля. Преобразованный аргумент будет занимать поле по крайней мере указанной ширины. При необходимости лишние позиции слева (или справа при левостороннем расположении) будут заполнены пробелами.
3. Точка, отделяющая ширину поля от величины, устанавливающей точность.
4. Число (точность), специфицирующее максимальное количество печатаемых символов в строке, или количество цифр после десятичной точки - для чисел с плавающей запятой, или минимальное количество цифр — для целого.
5. Буква *h*, если печатаемое целое должно рассматриваться как *short*, или *l* (латинская буква *ell*), если целое должно рассматриваться как *long*.

Символы-спецификаторы перечислены ниже. Если за % не помещен символ-спецификатор, поведение функции *printf* будет не определено. Ширину и точность можно специфицировать с помощью *; значение ширины (или точности) в этом случае берется из следующего аргумента (который должен быть типа *int*). Например, чтобы напечатать не более *max* символов из строки *s*, годится следующая запись:

```
printf("%. *s", max, s);
```

Основные форматы

Символ	Тип аргумента	Вид печати
d, i	int	десятичное целое
o	unsigned int	беззнаковое восьмеричное (octal) целое (без нуля слева)
x, X	unsigned int	беззнаковое шестнадцатеричное целое

		(без 0x или 0X слева), для 10...15 используются abcdef или ABCDEF
u	unsigned int	беззнаковое десятичное целое
c	int	одионый символ
s	char *	печатает символы, расположенные до знака \0, или в количестве, заданном точностью
f	double	[.]m.dddddd, где количество цифр d задается точностью (по умолчанию равно 6)
e, E	double	[.]m.ddddde±xx или [.]m.ddddde±xx, где количество цифр d задается точностью (по умолчанию равно 6)
g, G	double	использует %e или %E, если порядок меньше, чем -4, или больше или равен точности; в противном случае использует %f. Завершающие нули и завершающая десятичная точка не печатаются

Ввод

1. Функция scanf ():

Структура:

*int scanf (char *format, ...);*

Описание:

Функция *scanf* читает символы из стандартного входного потока, интерпретирует их согласно спецификациям строки *format* и рассылает результаты в свои остальные аргументы. Аргумент *format* и другие, каждый из которых должен быть указателем, определяют, где будут запоминаться должным образом преобразованные данные.

Функция *scanf* прекращает работу, когда оказывается, что исчерпан формат или вводимая величина не соответствует управляющей спецификации. В качестве результата *scanf* возвращает количество успешно введенных элементов данных. По исчерпанию файла она выдает EOF. Существенно то, что значение EOF не равно нулю, поскольку нуль *scanf* выдает, когда вводимый символ не соответствует первой спецификации форматной

строки. Каждое очередное обращение к `scanf` продолжает ввод символа, следующего сразу за последним обработанным.

2. Функция `sscanf()`:

Примечание: в отличие от `scanf()`, читает из строки, а не из стандартного ввода.

Структура:

*int sscanf(char *string, char *format, arg1, arg2, ...)*

Описание:

Функция `sscanf` просматривает строку `string` согласно формату `format` и рассылает полученные значения в `arg1`, `arg2` и т. д. Последние должны быть указателями.

Формат обычно содержит спецификации, которые используются для управления преобразованиями ввода. В него могут входить следующие элементы:

1. Пробелы или табуляции, которые игнорируются;
2. Обычные символы (исключая %), которые, как ожидается, совпадут с очередными символами, отличными от символов-разделителей входного потока;
3. Спецификации преобразования, каждая из которых начинается со знака % и завершается символом-спецификатором типа преобразования. В промежутке между этими двумя символами в любой спецификации могут располагаться, причем в том порядке, как они здесь указаны: знак * (признак подавления присваивания); число, определяющее ширину поля; буква *h*, *l* или *L*, указывающая на размер получаемого значения; и символ преобразования (*o*, *d*, *x*);

Спецификация преобразования управляет преобразованием следующего вводимого поля. Обычно результат помещается в переменную, на которую указывает соответствующий аргумент. Однако если в спецификации преобразования присутствует *, то поле ввода пропускается и никакое присваивание не выполняется. Поле ввода определяется как строка без символов-разделителей; оно простирается до следующего символа-разделителя или же ограничено шириной поля, если она задана. Поскольку символ новой строки относится к символам-разделителям, то `sscanf` при чтении будет переходить с одной строки на другую. (Символами-разделителями являются символы пробела, табуляции, новой строки, возврата каретки, вертикальной табуляции и перевода строки.)

Символ-спецификатор указывает, каким образом следует интерпретировать очередное поле ввода. Соответствующий аргумент должен быть указателем, как того требует механизм передачи параметров по значению, принятый в Си. Символы-спецификаторы приведены в таблице 7.2.

Перед символами-спецификаторами *d*, *l*, *o*, *u* и *x* может стоять буква *h*, указывающая на то, что соответствующий аргумент должен иметь тип *short ** (а не *int **), или *l* (латинская *ell*), указывающая на тип *long **. Аналогично, перед символами-спецификаторами *e*, *f* и *g* может стоять буква *l*, указывающая, что тип аргумента — *double ** (а не *float **).

Таблица 7.2 Основные форматы

<i>Символ</i>	<i>Вводимые данные</i>	<i>Тип аргумента</i>
d	десятичное целое	int
i	целое (Целое может быть восьмеричным (с 0 слева) или шестнадцатеричным (с 0x или 0X слева))	int
o	восьмеричное целое (с нулем слева или без него)	int *
u	беззнаковое десятичное целое	unsigned int *
x	шестнадцатеричное целое (с 0x или 0X слева или без них)	int *
c	символы (Следующие символы ввода (по умолчанию один) размещаются в указанном месте. Обычный пропуск символов- разделителей подавляется; чтобы прочсть очередной символ, отличный от символа-разделителя, используйте %1s)	char *
s	строка символов (без обрамляющих кавычек), указывающая на массив символов, достаточный для строки и завершающего символа '\0', который будет добавлен	char *
e, f, g	число с плавающей точкой возможно, со знаком; обязательно присутствие либо десятичной точки, либо экспоненциальной части, а возможно, и обеих вместе	float

Пример:

Необходимо прочесть строки ввода, содержащие данные вида «25 дек 1988»;

Решение:

Обращение к *scanf* выглядит следующим образом:

```
int day, year;
```

```
char monthname [20];
```

```
scanf ("%d %s %d", &day, monthname, &year);
```

В своем формате функция *scanf* игнорирует пробелы и табуляции. Кроме того, при поиске следующей порции ввода она пропускает во входном потоке все символы-разделители (пробелы, табуляции, новые строки и т.д.). Воспринимать входной поток, не имеющий фиксированного формата, часто оказывается удобнее, если вводить всю строку целиком и для каждого отдельного случая подбирать подходящий вариант *sscanf*.

Обращения к *scanf* могут перемежаться с вызовами других функций ввода. Любая функция ввода, вызванная после *scanf*, продолжит чтение с первого еще непрочитанного символа.

8.Массивы. Индексные выражения. Хранение в памяти одномерных и многомерных массивов

Массив — структура данных, в которой хранятся однотипные элементы, располагающиеся в памяти последовательно друг за другом в порядке возрастания индексов. Массив имеет вид: тип_переменной название_массива [длина_массива].

Индексы — номера элементов в массиве. Индексное выражение обычно используется для доступа к элементам массива, однако индексацию можно применить к любому указателю. Индексное выражение вычисляют путём сложения целого значения со значением указателя (или с адресом массива) и последующим разыменованем.

Например пусть дан одномерный массив целочисленного типа:

```
int MAS[размер]
```

MAS — адрес, указывающий на начало массива (1 его элемент), т.е. на нулевой индекс, следовательно MAS+1 — адрес, указывающий на 2 элемент массива, т.е. первый индекс массива, при этом, учитывая адресную арифметику, MAS+1 сдвинет адрес 2 элемента на количество байт, занимаемое типом элемента (в данном случае int занимает 4 байта).

Соответственно элемент массива с индексом i будет иметь адрес MAS+i. Для того, чтобы обратиться к элементу массива, необходимо разыменовать его адрес, это делается следующим образом: *(MAS+i) — такой способ обращения к элементу массива полностью эквивалентен: MAS[i]. P.s. считаю полезным приложить ссылку на это видео: одномерные массивы.

Теперь рассмотрим двумерный массив:

```
int MAS[количество строк][количество столбцов]
```

MAS — адрес, указывающий на первый элемент первой строки массива, т.е. указывает на строку с индексом 0. MAS+1 — адрес, который будет указывать на 1 элемент 2 строки массива, т.е. указывает на строку с индексом 1, причём этот адрес MAS+1 сдвинут на длину строки (в данном случае, чтобы узнать сдвиг в битах, необходимо 4 байта (т.к. тип элементов массива int) умножить на количество элементов в строке, причём количество элементов в строке определяется количеством столбцов, что очевидно из матричного представления двумерных массивов). Если хотим взять адрес 1 элемента i строки, то соответственно напишем MAS+i. Поскольку массив двумерный нам необходимо уметь брать адреса не только первого элемента i строки, а любого элемента массива. Возьмём адрес i, j элемента массива, т.е. к элементу массива у которого i индекс строки и j индекс столбца: MAS+i*(длина строки)+j. Для обращения к элементу массива необходимо разыменовать его адрес, при этом следует привести тип адреса начального

элемента массива, т.е. будем иметь: `*((int *)MAS+i*(длина строки)+j)`, такой способ обращения полностью эквивалентен: `MAS[i][j]`.

Многомерные массивы рассматриваются как массивы массивов, к примеру элементами трёхмерного массива являются двумерные массивы, соответственно обращение к их элементам происходит по принципу, изложенному выше. P.s. считаю полезным приложить ссылку на это видео: [двумерные массивы](#).

12. Массивы. Основные алгоритмы их обработки. Ввод-вывод, поиск экстремума, сортировка пузырьком

Существует 3 основных алгоритма сортировки массивов: а) обмен; б) выбор (выборка); в) вставка.

Чтобы понять, как работают эти методы, представьте себе колоду игральных карт. Чтобы отсортировать карты методом обмена, разложите их на столе лицом вверх и меняйте местами карты, расположенные не по порядку, пока вся колода не будет упорядочена. В методе выбора разложите карты на столе, выберите карту наименьшей значимости и положите ее в руку. Затем из оставшихся карт снова выберите карту наименьшей значимости и положите ее на ту, которая уже находится у вас в руке. Процесс повторяется до тех пор, пока в руке не окажутся все карты; по окончании процесса колода будет отсортирована. Чтобы отсортировать колоду методом вставки, возьмите все карты в руку. Выкладывайте их по одной на стол, вставляя каждую следующую карту в соответствующую позицию. Когда все карты окажутся на столе, колода будет отсортирована.

Ввод-вывод массивов. Ввод и вывод массивов происходит с помощью цикла for. Структура: for($i =$ (номер индекса, с которого начнётся перебор, может стоять выражение, вычисляющее этот индекс; $i <$ граница, последний индекс до которого необходимо делать перебор; шаг) {

тело цикла

}

Сортировка методом пузырька (отлично подойдёт для нахождения экстремумов):

Рассмотрим метод сортировки данных, который называется сортировка пузырьком (также его называют метод обмена). Будут приведены алгоритм и его реализация на языке программирования Си.

Упорядоченный массив создается на том же участке памяти, где находится исходная последовательность. Идея метода состоит в том, чтобы попарно сравнивать соседние элементы. Каждый проход начинается с начала последовательности. Сравнивается первый элемент со вторым: если порядок между ними нарушен, то они меняются местами. Затем сравниваются второй с третьим, третий с четвертым и так далее до конца массива; элементы с неправильным порядком в паре меняются местами. В итоге, после первого прохода, максимальный (или минимальный, в зависимости от вида сортировки: по

возрастанию/по убыванию) элемент будет находится на последнем месте в массиве, он как бы “всплывает” вверх. Именно поэтому этот метод называется сортировка пузырьком.

```
// Функция сортировки прямым обменом (метод "пузырька") void
bubbleSort(int *num, int size) { // Для всех элементов for (int i = 0; i < size - 1;
i++) { for (int j = (size - 1); j > i; j--) // для всех элементов после i-ого { if (num[j
- 1] > num[j]) // если текущий элемент меньше предыдущего { int temp =
num[j - 1]; // меняем их местами num[j - 1] = num[j]; num[j] = temp; } } }
```

Вставка.

Рассмотри алгоритм сортировки массива по возрастанию методом вставки. Фиксируем (вставляем) первый элемент массива и сравниваем его со следующим элементом, если он больше, то меняем местами, если меньше оставляем на месте, теперь вставлено 2 элемента, далее переходим к 3, он будет сравниваться со 2, если меньше, вставляем и переходим к следующему, если больше, то меняем местами со 2 элементом и сравниваем с 1 элементом, опять же, если он меньше, то оставляем на месте, если больше, то меняем местами, и так далее со следующими элементами.

```
void insertsort (double *a, int n)
{
double x; int i, j;
for (i=0; i < n; i++)
{
x = a[i];
for (j=i-1; j>=0 && a[j]>x; j--) a[j+1] = a[j];
a[j+1] = x;
}
}
```

Выбор.

На первом проходе цикла выбирается минимальный элемент из текущей последовательности и меняется местами с первым элементом последовательности. На следующей итерации цикла поиск минимального элемента осуществляется со второй

позиции, после меняется местами найденный минимальный элемент со вторым в списке. Такую процедуру выполняем до конца массива, пока он весь не будет отсортирован.

```
for (int i = 0; i < N; i++)  
{  
    minPosition = i;  
    for (int j = i + 1; j < N; j++)  
        if (mass[minPosition] > mass[j])  
            minPosition = j;  
    tmp = mass[minPosition];  
    mass[minPosition] = mass[i];  
    mass[i] = tmp;  
}
```

9.Массивы. Объявление. Передача массива в функции. Возврат массива из функции

Массивы

Массив — это совокупность переменных одного типа, к которым обращаются с помощью общего имени. Доступ к отдельному элементу массива может осуществляться с помощью индекса. В Си все массивы состоят из соприкасающихся (непрерывного) участков памяти. Наименьший адрес соответствует первому элементу. Наибольший адрес соответствует последнему элементу. Массивы могут иметь одну или несколько размерностей.

Массив характеризуется следующими основными понятиями:

1. Элемент массива (значение элемента массива) — значение, хранящееся в определенной ячейке памяти, расположенной в пределах массива, а также адрес этой ячейки памяти.

Каждый элемент массива характеризуется тремя величинами:

- адресом элемента — адресом начальной ячейки памяти, в которой расположен этот элемент;
- индексом элемента (порядковым номером элемента в массиве);
- значением элемента.

2. Адрес массива — адрес начального элемента массива.

3. Имя массива — идентификатор, используемый для обращения к элементам массива.

4. Размер массива — количество элементов массива

5. Размер элемента — количество байт, занимаемых одним элементом массива.

Графически расположение массива в памяти компьютера можно представить в виде непрерывной ленты адресов.

Расположение массива в памяти

Адрес	n	n+k	n+2k		n+k(q-1)
Значение	a[0]	a[1]	a[2]	...	a[q-1]
Индекс	0	1	2		q-1

Представленный на рисунке массив содержит Q элементов с индексами от 0 до $q-1$. Каждый элемент занимает в памяти компьютера k байт, причем расположение элементов в памяти последовательное.

Адреса i -го элемента массива имеет значение $n+k \cdot i$

Адрес массива представляет собой адрес начального (нулевого) элемента массива. Для обращения к элементам массива используется порядковый номер (индекс) элемента, начальное

значение которого равно 0. Так, если массив содержит Q элементов, то индексы элементов массива меняются в пределах от 0 до $Q-1$.

6. Длина массива — количество байт, отводимое в памяти для хранения всех элементов массива.

$$\text{ДлинаМассива} = \text{РазмерЭлемента} * \text{КоличествоЭлементов}$$

Для определения размера элемента массива может использоваться функция

```
int sizeof(тип);
```

Например,

```
sizeof(char) = 1;
sizeof(int) = 4;
sizeof(float) = 4;
sizeof(double) = 8;
```

Объявление и инициализация

Для объявления массива в языке Си используется следующий синтаксис:

тип имя[размерность]={инициализация}

Инициализация представляет собой набор начальных значений элементов массива, указанных в фигурных скобках, и разделенных запятыми.

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};    // массив a из 10 целых чисел
```

Если количество инициализирующих значений, указанных в фигурных скобках, меньше, чем количество элементов массива, указанное в квадратных скобках, то все оставшиеся элементы в массиве (для которых не хватило инициализирующих значений) будут равны нулю. Это свойство удобно использовать для задания нулевых значений всем элементам массива.

```
int b[10] = {0};    // массив b из 10 элементов, инициализированных 0
```

Если массив проинициализирован при объявлении, то константные начальные значения его элементов указываются через запятую в фигурных скобках. В этом случае количество элементов в квадратных скобках может быть опущено.

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

При обращении к элементам массива индекс требуемого элемента указывается в квадратных скобках []

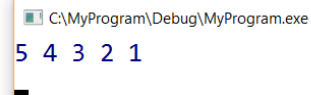
```
#include <stdio.h>
int main()
{
```

```

int a[] = { 5, 4, 3, 2, 1 }; // массив a содержит 5 элементов
printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
getchar();
return 0;
}

```

Результат выполнения программы:



```

C:\MyProgram\Debug\MyProgram.exe
5 4 3 2 1

```

Однако часто требуется задавать значения элементов массива в процессе выполнения программы. При этом используется объявление массива без инициализации. В таком случае указание количества элементов в квадратных скобках обязательно.

```
int a[10];
```

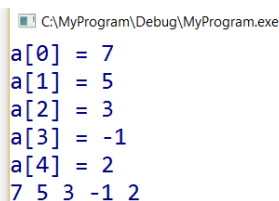
Для задания начальных значений элементов массива очень часто используется параметрический цикл:

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    int a[5]; // объявлен массив a из 5 элементов
    int i;
    // Ввод элементов массива
    for (i = 0; i < 5; i++)
    {
        printf("a[%d] = ", i);
        scanf("%d", &a[i]); // &a[i] — адрес i-го элемента массива
    }
    // Вывод элементов массива
    for (i = 0; i < 5; i++)
        printf("%d ", a[i]); // пробел в формате печати обязателен
    getchar(); getchar();
    return 0;
}

```

Результат выполнения программы



```

C:\MyProgram\Debug\MyProgram.exe
a[0] = 7
a[1] = 5
a[2] = 3
a[3] = -1
a[4] = 2
7 5 3 -1 2

```

Многомерные массивы

В языке Си могут быть также объявлены многомерные массивы. Отличие многомерного массива от одномерного состоит в том, что в одномерном массиве положение элемента определяется одним индексом, а в многомерном — несколькими. Примером многомерного массива является матрица.

Общая форма объявления многомерного массива

```
тип имя[размерность1][размерность2]...[размерностьт];
```

Элементы многомерного массива располагаются в последовательных ячейках оперативной памяти по возрастанию адресов. В памяти компьютера элементы многомерного массива располагаются подряд, например, массив, имеющий 2 строки и 3 столбца,

```
int a[2][3];
```

будет расположен в памяти следующим образом

Адрес	n	n+4	n+8	n+12	n+16	n+20
Значение	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
Индекс	0 0	0 1	0 2	1 0	1 1	1 2

Общее количество элементов в приведенном двумерном массиве определится как

$$\text{КоличествоСтрок} * \text{КоличествоСтолбцов} = 2 * 3 = 6.$$

Количество байт памяти, требуемых для размещения массива, определится как

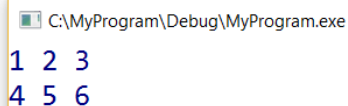
$$\text{КоличествоЭлементов} * \text{РазмерЭлемента} = 6 * 4 = 24 \text{ байта.}$$

Инициализация многомерных массивов

Значения элементов многомерного массива, как и в одномерном случае, могут быть заданы константными значениями при объявлении, заключенными в фигурные скобки {}. Однако в этом случае указание количества элементов в строках и столбцах должно быть обязательно указано в квадратных скобках [].

```
#include <stdio.h>
int main()
{
    int a[2][3] = { 1, 2, 3, 4, 5, 6 };
    printf("%d %d %d\n", a[0][0], a[0][1], a[0][2]);
    printf("%d %d %d\n", a[1][0], a[1][1], a[1][2]);
    getchar();
    return 0;
}
```

Результат выполнения

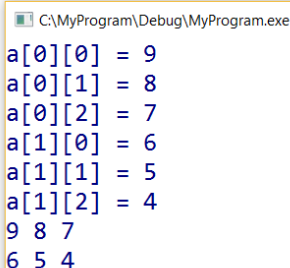


```
C:\MyProgram\Debug\MyProgram.exe
1 2 3
4 5 6
```

Однако чаще требуется вводить значения элементов многомерного массива в процессе выполнения программы. С этой целью удобно использовать вложенный **параметрический цикл**.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    int a[2][3]; // массив из 2 строк и 3 столбцов
    int i, j;
```

// Ввод элементов массива



```
C:\MyProgram\Debug\MyProgram.exe
a[0][0] = 9
a[0][1] = 8
a[0][2] = 7
a[1][0] = 6
a[1][1] = 5
a[1][2] = 4
9 8 7
6 5 4
```

```

for (i = 0; i<2; i++) // цикл по строкам
{
    for (j = 0; j<3; j++) // цикл по столбцам
    {
        printf("a[%d][%d] = ", i, j);
        scanf("%d", &a[i][j]);
    }
}
// Вывод элементов массива
for (i = 0; i<2; i++) // цикл по строкам
{
    for (j = 0; j<3; j++) // цикл по столбцам
    {
        printf("%d ", a[i][j]);
    }
    printf("\n"); // перевод на новую строку
}
getchar(); getchar();
return 0;
}

```

Результат выполнения

Передача массива в функцию

Обработку массивов удобно организовывать с помощью специальных функций. Для обработки массива в качестве аргументов функции необходимо передать

- адрес массива,
- размер массива.

Исключение составляют функции обработки строк, в которые достаточно передать только адрес.

При передаче переменные в качестве аргументов функции данные передаются как копии. Это означает, что если внутри функции произойдет изменение значения параметра, то это никак не повлияет на его значение внутри вызывающей функции.

Если в функцию передается адрес переменной (или адрес массива), то все операции, выполняемые в функции с данными, находящимися в пределах видимости указанного адреса, производятся над оригиналом данных, поэтому исходный массив (или значение переменной) может быть изменено вызываемой функцией.

Возврат массива из функции

Доступ к элементам массива можно осуществить с помощью указателей. При этом обращение к полям структуры через указатель будет выглядеть как:

указатель -> поле

или

(*указатель).поле

Указатель — указатель на структуру или объединение;
поле — поле структуры или объединения;

10. Структуры и объединения. Передача их в функции и возврат из функций

Язык Си предоставляет пять способов создания своих типов данных:

1. Структура — это совокупность переменных, объединенных одним именем. Она называется составным типом данных. (Также часто используется термин «смешанный тип данных».)
2. Битовое поле — это разновидность структуры, предоставляющая легкий доступ к отдельным битам.
3. Объединение позволяет одному участку памяти содержать два или более различных типов данных.
4. Перечисление — это список символов.
5. Ключевое слово `typedef` создает новое имя существующему типу.

Структуры

Структура — это объединение нескольких объектов, возможно, различного типа под одним именем, которое является типом структуры. В качестве объектов могут выступать переменные, массивы, указатели и другие структуры.

Структуры позволяют трактовать группу связанных между собой объектов не как множество отдельных элементов, а как единое целое. Структура представляет собой сложный тип данных, составленный из простых типов.

Общая форма объявления структуры:

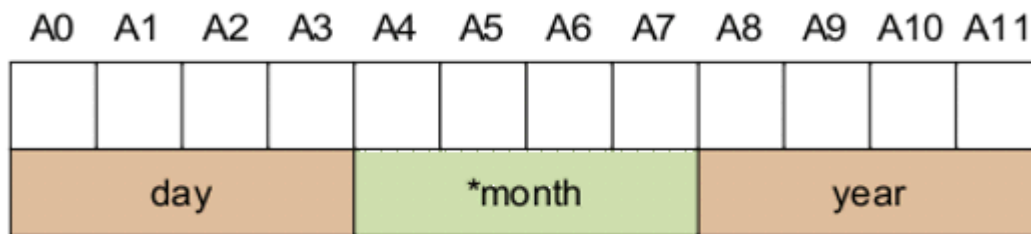
```
struct тип_структуры
{
    тип ИмяЭлемента1;
    тип ИмяЭлемента2;
    . . .
    тип ИмяЭлементаn;
};
```

После закрывающей фигурной скобки `}` в объявлении структуры обязательно ставится точка с запятой.

Пример объявления структуры

```
struct date
{
    int day;           // 4 байта
    char *month;       // 4 байта
    int year;          // 4 байта
};
```

Поля структуры располагаются в памяти в том порядке, в котором они объявлены:



В указанном примере структура `date` занимает в памяти 12 байт. Кроме того, указатель `*month` при инициализации будет началом текстовой строки с названием месяца, размещенной в памяти.

При объявлении структур, их разрешается вкладывать одну в другую.

```
struct persone
{
    char lastname[20];    // фамилия
    char firstname[20];   // имя
    struct date bd;       // дата рождения
};
```

Инициализация полей структуры

Инициализация полей структуры может осуществляться двумя способами:

- присвоение значений элементам структуры в процессе объявления переменной, относящейся к типу структуры;
- присвоение начальных значений элементам структуры с использованием функций ввода-вывода (например, `printf()` и `scanf()`).

В первом способе инициализация осуществляется по следующей форме:

```
struct ИмяСтруктуры ИмяПеременной={ЗначениеЭлемента1, ЗначениеЭлемента_2, . . . , ЗначениеЭлементаn};
```

```
struct date bd={8,"июня", 1978};
```

Имя элемента структуры является составным. Для обращения к элементу структуры нужно указать имя структуры и имя самого элемента. Они разделяются точкой:

ИмяПеременной.ИмяЭлементаСтруктуры

```
printf("%d %s %d",bd.day, bd.month, bd.year);
```

Второй способ инициализации объектов языка Си с использованием функций ввода-вывода.

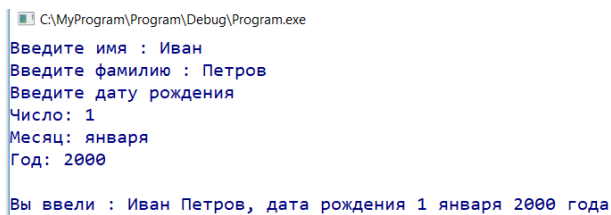
```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
struct date {
```

```

    int day;
    char month[20];
    int year;
};
struct persone {
    char firstname[20];
    char lastname[20];
    struct date bd;
};
int main() {
    system("chcp 1251");
    system("cls");
    struct persone p;
    printf("Введите имя : ");
    scanf("%s", p.firstname);
    printf("Введите фамилию : ");
    scanf("%s", p.lastname);
    printf("Введите дату рождения\nЧисло: ");
    scanf("%d", &p.bd.day);
    printf("Месяц: ");
    scanf("%s", p.bd.month);
    printf("Год: ");
    scanf("%d", &p.bd.year);
    printf("\nВы ввели : %s %s, дата рождения %d %s %d года",
        p.firstname, p.lastname, p.bd.day, p.bd.month, p.bd.year);
    getchar(); getchar();
    return 0;
}

```

Результат работы:



```

C:\MyProgram\Program\Debug\Program.exe
Введите имя : Иван
Введите фамилию : Петров
Введите дату рождения
Число: 1
Месяц: января
Год: 2000

Вы ввели : Иван Петров, дата рождения 1 января 2000 года

```

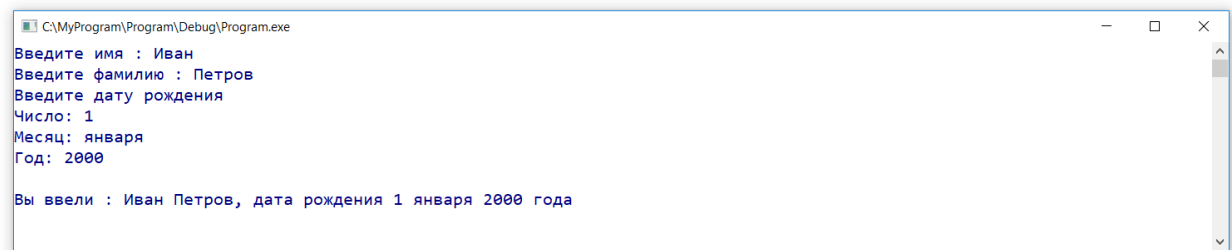
Имя структурной переменной может быть указано при объявлении структуры. В этом случае оно размещается после закрывающей фигурной скобки `}`. Область видимости такой структурной переменной будет определяться местом описания структуры.

```

struct complex_type    // имя структуры
{
    double real;
    double imag;
} number;              // имя структурной переменной

```

Поля приведенной структурной переменной: `number.real`, `number.imag` .



Структуры предоставляют возможность хранения большого количества различных значений, объединенных одним общим названием. Это делает программу более модульной, что в свою очередь позволяет легко изменять код, потому что он становится более компактным. Структуры, как правило, используют тогда, когда в программе есть много данных и их нужно сгруппировать вместе — например, такие данные могут использоваться для хранения записей из базы данных.

Битовые поля

Используя структуры, можно упаковать целочисленные компоненты еще более плотно, чем это было сделано с использованием массива.

Набор разрядов целого числа можно разбить на битовые поля, каждое из которых выделяется для определенной переменной. При работе с битовыми полями количество битов, выделяемое для хранения каждого поля отделяется от имени двоеточием

тип имя: КоличествоБит

При работе с битовыми полями нужно внимательно следить за тем, чтобы значение переменной не потребовало памяти больше, чем под неё выделено.

Пример. Разработать программу, осуществляющую упаковку даты в формат

15	9 8	5 4	0
Год (0 = 1980)	Месяц (1...12)	День (1...31)	

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#define YEAR0 1980
struct date
{
    unsigned short day : 5;
    unsigned short month : 4;
    unsigned short year : 7;
};
int main() {
    struct date today;
    system("chcp 1251");
    system("cls");
    today.day = 16;
    today.month = 12;
    today.year = 2013 - YEAR0; //today.year = 33
    printf("\n Сегодня %u.%u.%u \n", today.day, today.month, today.year + YEAR0);
}

```

```

printf("\n Размер структуры today : %d байт", sizeof(today));
printf("\n Значение элемента today = %hu = %hx шестн.", today
, today);
getchar();
return 0;
}

```

Результат выполнения

Массивы структур

Работа с массивами структур аналогична со статическими массивами других типов
Пример. Библиотека из 3 книг

```

#include <stdio.h>
#include <stdlib.h>
struct book
{
    char title[15];
    char author[15];
    int value;
}
int main()
{
    struct book libry[3];
    system("chcp 1251");
    system("cls");
    for (i = 0; i<3; i++)

```

```

        printf("Введите название %d книги : ", i + 1);
        gets_s(libry[i].title);
        printf("Введите автора %d книги : ", i + 1);
        gets_s(libry[i].author);
        printf("Введите цену %d книги : ", i + 1);
        scanf_s("%d", &libry[i].value);
        getchar();
    }
    for (i = 0; i<3; i++)
    {
        printf("\n %d. %s ", i + 1, libry[i].author);
        printf("%s %d", libry[i].title, libry[i].value);
    }
    getchar();
    return 0;
}

```

C:\MyProgram\Program\Debug\Program.exe

Сегодня 16.12.2013

Размер структуры today : 2 байт

Значение элемента today = 17296 = 4390 шестн.

работе
данных.

C:\MyProgram\Project\Debug\Project.exe

```

Введите название 1 книги : Сказки
Введите автора 1 книги : Пушкин
Введите цену 1 книги : 230
Введите название 2 книги : Басни
Введите автора 2 книги : Крылов
Введите цену 2 книги : 450
Введите название 3 книги : Стихи
Введите автора 3 книги : Барто
Введите цену 3 книги : 36

```

```

1. Пушкин Сказки 230
2. Крылов Басни 450
3. Барто Стихи 36

```

Указатели на структуры

Доступ к элементам структуры или объединения можно осуществить с помощью указателей. Для этого необходимо инициализировать указатель адресом структуры или объединения.

Для организации работы с массивом можно использовать указатель. При этом обращение к полям структуры через указатель будет выглядеть как:

указатель -> поле

или

(*указатель).поле

Указатель — указатель на структуру или объединение;

поле — поле структуры или объединения;

Обратите внимание на удаление массива структур: при удалении экземпляра структуры он не удаляет своих полей самостоятельно, поэтому необходимо сначала удалять поля, после этого удалять сам массив.

Динамическое выделение памяти для структур

Динамически выделять память под массив структур необходимо в том случае, если заранее неизвестен размер массива. Для определения размера структуры в байтах используется операция `sizeof(ИмяСтруктуры)`.

Пример. Библиотека из 3 книг

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
struct book
{
    char title[15];
    char author[15];
    int value;
};
int main()
{
    struct book *lib;
    int i;
```

```

system("chcp 1251");
system("cls");
lib = (struct book*)malloc(3 * sizeof(struct book));
for (i = 0; i<3; i++)
{
    printf("Введите название %d книги : ", i + 1);
    gets_s((lib + i)->title);
    printf("Введите автора %d книги : ", i + 1);
    gets_s((lib + i)->author);
    printf("Введите цену %d книги : ", i + 1);
    scanf_s("%d", &(lib + i)->value);
    getchar();
}
for (i = 0; i<3; i++)
{
    printf("\n %d. %s ", i + 1, (lib + i)->author);
    printf("%s %d", (lib + i)->title, (lib + i)->value);
}
getchar();
return 0;
}

```

Результат выполнения аналогичен предыдущему решению.

Определение нового типа

Когда мы определяем новую структуру с помощью служебного слова **struct**, в пространстве имён структур создаётся новый идентификатор. Для доступа к нему необходимо использовать служебное слово **struct**. Можно определить новый тип с помощью служебного слова **typedef**. Тогда будет создан псевдоним для нашей структуры, видимый в глобальном контексте.

Объединения

Объединениями называют сложный тип данных, позволяющий размещать в одном и том же месте оперативной памяти данные различных типов.

Размер оперативной памяти, требуемый для хранения объединений, определяется размером памяти, необходимым для размещения данных того типа, который требует максимального количества байт.

Когда используется элемент меньшей длины, чем наиболее длинный элемент объединения, то этот элемент использует только часть отведенной памяти. Все элементы объединения хранятся в одной и той же области памяти, начиная с одного адреса.

Общая форма объявления объединения


```
union ИмяОбъединения
{
    тип ИмяОбъекта1;
    тип ИмяОбъекта2;
    . . .
    тип ИмяОбъектаn;
};
```

имя 1			
имя 2			
имя 3			

Объединения применяются для следующих целей:

- для инициализации объекта, если в каждый момент времени только один из многих объектов является активным;
- для интерпретации представления одного типа данных в виде другого типа.

Например, удобно использовать объединения, когда необходимо вещественное число типа **float** представить в виде совокупности байтов

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
union types
{
    float f;
    unsigned char b[4];
};
int main()
{
    types value;
    printf("N = ");
    scanf("%f", &value.f);
    printf("%f = %x %x %x %x", value.f, value.b[0], value.b[1], value.b[2], value.b[3]);
    getchar();
    getchar();
    return 0;
}
```

Результат выполнения:

```
C:\MyProgram\Program\Debug\Program.exe
N = 15.3
15.300000 = cd cc 74 41
```

Пример. Поменять местами два младших байта во введенном числе

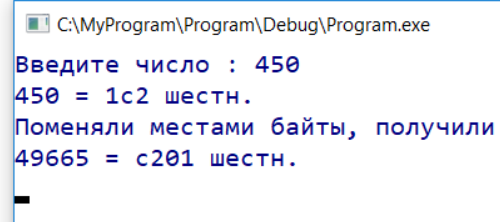
```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
int main() {
    char temp;
    system("chcp 1251");
    system("cls");
    union
```

```

{
    unsigned char p[2];
    unsigned int t;
} type;
printf("Введите число : ");
scanf("%d", &type.t);
printf("%d = %x шестн.\n", type.t, type.t);
// Замена байтов
temp = type.p[0];
type.p[0] = type.p[1];
type.p[1] = temp;
printf("Поменяли местами байты, получили\n");
printf("%d = %x шестн.\n", type.t, type.t);
getchar(); getchar();
return 0;
}

```

Результат выполнения



```

C:\MyProgram\Program\Debug\Program.exe
Введите число : 450
450 = 1c2 шестн.
Поменяли местами байты, получили
49665 = c201 шестн.

```

11.Правила определения переменных и типов. Инициализация данных

Данные, значения которых во время выполнения программы можно изменять, называются переменными, неизменяемые данные называются константами. В программе все данные перед их использованием должны быть объявлены или определены. В операторах определения данных указывается тип данных и перечисляется через запятую имена переменных, имеющих данный тип.

Определение переменных имеет следующий формат:

[спецификатор класса памяти] спецификатор типа_ идентификатор

[static] int peremennaya

В случае, если при определении сразу присваивается значение, то говорят, что она инициализирована.

int peremennaya=12

Идентификатор может быть записан с квадратными скобками, круглыми скобками или перед ним может быть один или несколько знаков *. В Си переменная может начинаться с подчерка или буквы, но не с числа. Переменная может включать в себя символы английского алфавита, цифры и знак подчёркивания. Переменная не должна совпадать с ключевыми словами (это специальные слова, которые используются в качестве управляющих конструкций, для определения типов и т.п.) Нередко компиляторы имеют ограничение на длину названий переменных, например, для некоторых компиляторов длина имени переменной не должна превышать 31 символа, для других компиляторов ограничение может быть более жестким. Также стоит учитывать, что C - регистрозависимый язык, а это значит, что регистр символов имеет большое значение.

Спецификатор типа – это одно или несколько ключевых слов, определяющих тип переменной. Язык Си определяет стандартный набор основных типов данных (int (целочисленные), char (символы), double (действительные числа двойной точности), float (действительные числа)). При определении переменных им можно присвоить начальное значение.

Базовые типы: целые: спецификация типов:

- signed char – знаковый символьный;
- signed int – знаковый целый;
- signed short int – знаковый короткий целый;
- signed long int – знаковый длинный целый;
- unsigned char- беззнаковый символьный;
- unsigned int- беззнаковый целый;
- unsigned short int – беззнаковый короткий целый;
- unsigned long int – беззнаковый длинный целый.

Базовые типы: плавающие: спецификация типов:

- float – плавающий одинарной точности;
- double – плавающий двойной точности;
- long float – длинный плавающий одинарной точности;
- long double – длинный плавающий двойной точности.

Базовые типы: прочие: спецификация типов:

- void– пустой:
- enum– перечислимый.

Есть слова, определяющие класс памяти, например, static(переменная будет сохранять в памяти свое место и значение и будет «видна» только в этом блоке, а также в блоках вложенных в него).

Если ключевое слово, определяющее класс памяти, опущено, то класс памяти определяется по контексту. Статическая переменная может иметь только константную инициализацию.

Например, она не может быть инициализирована вызовом функции.

Во время работы с числами можно использовать шестнадцатеричный и восьмеричный формат представления. Числа в шестнадцатеричной системе счисления начинаются с 0x, в восьмеричной системе с нуля. Соответственно, если число начинается с нуля, то в нём не должно быть цифр выше 7

```
int x = 0xFF;
```

```
int y = 077
```

Очень важно запомнить, что переменные в си не инициализируются по умолчанию нулями, как во многих других языках программирования. После объявления переменной в ней хранится "мусор" - случайное значение, которое осталось в той области памяти, которая была выделена под переменную.

Объявление и присваивание значения:

```
int A;
```

```
A=7;
```

Инициализация:

```
int a = 7
```

Переменная может быть инициализирована только после операции объявления.

Правило: Предоставляя исходное значение переменной, используйте инициализацию, вместо операции присваивания.

14.Определение и вызов функций.

Фактические и формальные параметры

После имени функции в круглых скобках перечислены формальные параметры с указанием их типов. Формальные параметры разделены запятыми. В теле функции ими пользуются так же, как обычными переменными.

Функция может быть и без параметров, тогда их список будет пустым. Такой пустой список можно указать в явном виде, поместив для этого внутри скобок ключевое слово `void` или просто ничего не указывая в круглых скобках.

```
void prnErr()
```

Аргументы, передаваемые функции при ее вызове, называются фактическими параметрами. Фактические параметры — это то, что стоит на самом деле при вызове функции. А при вызове функции в качестве фактических параметров могут стоять: имена переменных (такие же или совершенно другие), выражения или просто константы.

Значения фактических параметров заносятся в соответствующие формальные параметры, т.е. фактические параметры как бы замещают формальные параметры при вызове функции.

Обратить внимание! Тип должен указываться для каждого формального параметра в отдельности:

```
Пример: // функция, которая находит модуль вещественного числа
float MyAbs (float x) // x — формальный параметр
{
    if (x<0) return (float) (-x);
    else return x;
}
float res, a;

a = -18.25f;
res = MyAbs (a); // res = 18.25f; переменная a — фактический параметр
res = MyAbs (-23); // res = 23; константа 23 — фактический параметр
```

При вызове функции фактические параметры копируются в специальные ячейки памяти в стеке (стек — часть памяти). Эти ячейки памяти отведены для формальных параметров. Таким образом, формальные параметры (через использование стека) получают значение фактических параметров.

Поскольку, фактические параметры копируются в стек, то изменение значений формальных параметров в теле функции не изменит значений фактических параметров (так как это есть копия фактических параметров). Если требуется изменить переменную из функции, то нужно передавать указатель на эту переменную.

Область видимости формальных параметров функции определяется границами тела функции, в которой они описаны.

13 (№9). Определение и вызов функций.

Передача массивов. Возврат массивов

Когда массив используется в качестве аргумента функции, передается только адрес массива, а не копия всего массива. При вызове функции с именем массива в функцию передается указатель на первый элемент массива. (Надо помнить, что в С имена массивов без индекса - это указатели на первый элемент массива.) Параметр должен иметь тип, совместимый с указателем. Имеется три способа объявления параметра, предназначенного для получения указателя на массив. Во-первых, он может быть объявлен как массив

```
void display(int num[10]);
```

Следующий способ состоит в объявлении параметра для указания на безразмерный массив

```
void display(int num[]);
```

Последний способ, которым может быть объявлен `num`, - это наиболее типичный способ, применяемый при написании профессиональных программ, - через указатель:

```
void display(int *num)
```

В общем случае для передачи одномерного массива в функцию передается указатель на первый элемент массива и переменную размера массива.

Можно передавать и статические, и массивы переменного значения, и динамические массивы.

В случае, если все используемые массивы имеют один размер, то можно задавать его через глобальную константу и в функцию не передавать.

С другой стороны, элемент массива используется как аргумент, трактуемый как и другие простые переменные. Например, программа может быть написана без передачи всего массива:

```
#include <stdio.h>
void display(int num);
int main(void) /* вывод чисел */
{
    int t[10], i;
    for (i=0; i<10; ++i) t[i] = i;
    for (i=0; i<10; i++) display(t[i]);
    return 0;
}
```

```
void display(int num)
{
    printf ("%d ", num);
}
```

Как можно видеть, в `display()` передается параметр типа `int`. Не имеет значения, что `display()` вызывается с элементом массива в качестве параметра, поскольку передается только одно значение.

Важно понять, что при использовании массива в качестве аргумента функции происходит передача в функцию его адреса. Это означает, что код внутри функции действует и может изменять настоящее значение массива, используемого при вызове.

14.Определение и вызов функций.

Передача параметров функции main

В отличие от других языков программирования высокого уровня в языке СИ нет деления на процедуры, подпрограммы и функции, здесь вся программа строится только из функций.

Функция — это совокупность объявлений и операторов, обычно предназначенная для решения определенной задачи. Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова. В любой программе на СИ должна быть функция с именем `main` (главная функция), именно с этой функции, в каком бы месте программы она не находилась, начинается выполнение программы.

Функция — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

При вызове функции ей при помощи аргументов (формальных параметров) могут быть переданы некоторые значения (фактические параметры), используемые во время выполнения функции. Функция может возвращать некоторое (одно !) значение. Это возвращаемое значение и есть результат выполнения функции, который при выполнении программы подставляется в точку вызова функции, где бы этот вызов ни встретился. Допускается также использовать функции не имеющие аргументов и функции не возвращающие никаких значений. Действие таких функций может состоять, например, в изменении значений некоторых переменных, выводе на печать некоторых текстов и т.п..

С использованием функций в языке СИ связаны три понятия

1. определение функции (описание действий, выполняемых функцией),
2. объявление функции (задание формы обращения к функции),
3. вызов функции.

Определение функции

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

- тип возвращаемого значения;
- имя функции;
- информация о формальных аргументах;
- тело функции.

Определение функции имеет следующий синтаксис:

```
ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)
{
    ТелоФункции;
    ...
    return(ВозвращаемоеЗначение);
}
```

Пример: Функция сложения двух вещественных чисел


```
float function(float x, float z)
{
    float y;
    y=x+z;
    return(y);
}
```

В указанном примере возвращаемое значение имеет тип **float**. В качестве возвращаемого значения в вызывающую функцию передается значение переменной **y**. Формальными аргументами являются значения переменных **X** и **Z**.

Если функция не возвращает значения, то тип возвращаемого значения для нее указывается как **void**. При этом операция **return** может быть опущена. Если функция не принимает аргументов, в круглых скобках также указывается **void**.

Различают **системные** (в составе систем программирования) и **собственные** функции.

Системные функции хранятся в стандартных библиотеках, и пользователю не нужно вдаваться в подробности их реализации. Достаточно знать лишь их сигнатуру. Примером системных функций, используемых ранее, являются функции **printf()** и **scanf()**.

Собственные функции — это функции, написанные пользователем для решения конкретной подзадачи.

Разбиение программ на функции дает следующие преимущества:

- Функцию можно вызвать из различных мест программы, что позволяет избежать повторения программного кода.
- Одну и ту же функцию можно использовать в разных программах.
- Функции повышают уровень модульности программы и облегчают ее проектирование.
- Использование функций облегчает чтение и понимание программы и ускоряет поиск и исправление ошибок.

С точки зрения вызывающей программы функцию можно представить как некий «черный ящик», у которого есть несколько входов и один выход. С точки зрения вызывающей программы неважно, каким образом производится обработка информации внутри функции. Для корректного использования функции достаточно знать лишь ее сигнатуру.

Вызов функции

Общий вид вызова функции

Переменная = ИмяФункции(СписокФактическихАргументов);

Фактический аргумент — это величина, которая присваивается формальному аргументу при вызове функции. Таким образом, **формальный аргумент** — это переменная в вызываемой функции, а **фактический аргумент** — это конкретное значение, присвоенное этой переменной вызывающей функцией. Фактический аргумент может быть константой, переменной или выражением. Если фактический аргумент представлен в виде выражения, то его значение сначала вычисляется, а затем передается в вызываемую функцию. Если в функцию требуется передать несколько значений, то они записываются через запятую. При этом формальные параметры заменяются значениями фактических параметров в порядке их следования в сигнатуре функции.

Возврат в вызывающую функцию

По окончании выполнения вызываемой функции осуществляется возврат значения в точку ее вызова. Это значение присваивается переменной, тип которой должен соответствовать типу возвращаемого значения функции. Функция может передать в вызывающую программу только одно значение. Для передачи возвращаемого значения в вызывающую функцию используется оператор `return` в одной из форм:

```
return(ВозвращаемоеЗначение);
```

```
return ВозвращаемоеЗначение;
```

Действие оператора следующее: значение выражения, заключенного в скобки, вычисляется и передается в вызывающую функцию. Возвращаемое значение может использоваться в вызывающей программе как часть некоторого выражения.

Оператор `return` также завершает выполнение функции и передает управление следующему оператору в вызывающей функции. Оператор `return` не обязательно должен находиться в конце тела функции.

Функции могут и не возвращать значения, а просто выполнять некоторые вычисления. В этом случае указывается пустой тип возвращаемого значения `void`, а оператор `return` может либо отсутствовать, либо не возвращать никакого значения:

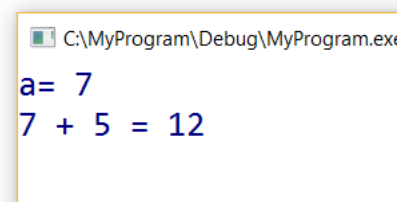
```
return;
```

Пример: Посчитать сумму двух чисел.

```
#define _CRT_SECURE_NO_WARNINGS // для возможности использования scanf
#include <stdio.h>
// Функция вычисления суммы двух чисел

// в функцию передаются два
{
вычисляем сумму чисел и

возвращаем значение k
int main()
{
    int a, r; // описание двух целых переменных
    printf("a= ");
    scanf("%d", &a); // вводим a
    r = sum(a, 5); // вызов функции: x=a, y=5
    printf("%d + 5 = %d", a, r); // вывод: a + 5 = r
    getchar(); getchar(); // мы использовали scanf(),
    return 0; // поэтому getchar() вызываем дважды
}
```



```
int sum(int x, int y)
целых числа
    int k = x + y; //
сохраняем в k
    return k;      //
}
```

Результат выполнения

В языке Си нельзя определять одну функцию внутри другой.

В языке Си нет требования, чтобы семантика функции обязательно предшествовало её вызову. Функции могут определяться как до вызывающей функции, так и после нее. Однако если семантика вызываемой функции описывается ниже ее вызова, необходимо до вызова функции определить прототип этой функции, содержащий:

- тип возвращаемого значения;
- имя функции;
- типы формальных аргументов в порядке их следования.

Прототип необходим для того, чтобы компилятор мог осуществить проверку соответствия типов передаваемых фактических аргументов типам формальных аргументов. Имена формальных аргументов в прототипе функции могут отсутствовать.

Если в примере выше тело функции сложения чисел разместить после тела функции `main`, то код будет выглядеть следующим образом:

```
#define _CRT_SECURE_NO_WARNINGS // для возможности использования scanf
#include <stdio.h>
int sum(int, int); // сигнатура
int main()
{
    int a, r;
    printf("a= ");
    scanf("%d", &a);
    r = sum(a, 5); // вызов функции: x=a, y=5
    printf("%d + 5 = %d", a, r);
    getchar(); getchar();
    return 0;
}
int sum(int x, int y) //семантика
{
    int k;
    k = x + y;
    return(k);
}
```

Сигнатура функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

Рекурсивные функции

Функция, которая вызывает сама себя, называется **рекурсивной функцией**.

Рекурсия — вызов функции из самой функции.

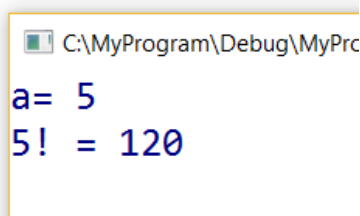
Пример рекурсивной функции — функция вычисления факториала.

```

#define _CRT_SECURE_NO_WARNINGS // для возможности использования scanf
#include <stdio.h>
int fact(int num) // вычисление факториала числа num
{
    if (num <= 1) return 1; // если число не больше 1, возвращаем 1
    else return num*fact(num - 1); // рекурсивный вызов для числа на 1
меньше
}
// Главная функция
int main()
{
    int a, r;
    printf("a= ");
    scanf("%d", &a);
    r = fact(a); // вызов функции: num=a
    printf("%d! = %d", a, r);
    getchar(); getchar();
    return 0;
}

```

Результат выполнения



Математические функции

Математические функции хранятся в стандартной библиотеке `math.h`. Аргументы большинства математических функций имеют тип `double`. Возвращаемое значение также имеет тип `double`.

Углы в тригонометрических функциях задаются в радианах.

Передача параметров в функцию main

При создании консольного приложения в языке программирования Си, автоматически создается строка очень похожая на эту:

```
int main(int argc, char* argv[]) // параметры функции main()
```

Эта строка — заголовок главной функции `main()`, в скобках объявлены параметры `argc` и `argv`. Так вот, если программу запускать через командную строку, то существует возможность передать какую-либо информацию этой программе, для этого и существуют параметры `argc` и `argv[]`. Параметр `argc` имеет тип данных `int`, и содержит количество параметров, передаваемых в функцию `main`. Причем `argc` всегда не меньше 1, даже когда мы не передаем никакой информации, так как первым параметром считается имя функции. Параметр `argv[]` это массив указателей на строки. Через командную строку можно передать только данные строкового типа. Указатели и строки — это две большие темы, под которые созданы отдельные разделы. Так вот именно через параметр `argv[]` и передается какая-либо информация.

Разработаем программу, которую будем запускать через командную строку, и передавать ей некоторую информацию.

15.Время жизни и область видимости программных объектов

Время жизни переменной может быть *глобальным* и *локальным*. Переменная с глобальным временем жизни характеризуется тем, что в течение всего времени выполнения программы с ней ассоциирована ячейка памяти и значение. Переменной с локальным временем жизни выделяется новая ячейка памяти при каждом входе в блок, в котором она определена или объявлена. **Время жизни функции** всегда глобально.

Область видимости объекта (переменной или функции) определяет, в каких участках программы допустимо использование имени этого объекта.

Область видимости имени начинается в точке объявления, точнее, сразу после объявителя, но перед инициализатором. Поэтому допускается использование имени в качестве инициализирующего значения для себя самого.

```
int x = x;           // Странно!  
Это допустимо, но не разумно.
```

Прежде чем имя может быть использовано в программе на Си, оно должно быть объявлено (иметь *объявление*), т.е. должен быть указан тип имени, чтобы компилятор знал, на сущность какого вида ссылается имя. *Определение* не только связывает тип с именем, но и определяет некоторую сущность, которая соответствует имени. В программе на Си для каждого имени должно быть ровно одно определение. Объявлений же может быть несколько. Все объявления некой сущности должны согласовываться по типу этой сущности.

```
int count;  
int count;           // Ошибка – повторное определение
```

```
extern int error_number;  
extern short error_number; // Ошибка – несоответствие типов объявлений
```

Объявления и определения, записанные внутри какого-либо блока, называются *внутренними* или *локальными*. Объявления и определения, записанные за пределами всех блоков, называются *внешними* или *глобальными*.

Переменные

Объявление переменной задает имя и атрибуты переменной. **Определение переменной**, помимо этого, приводит к *выделению для нее памяти*. Кроме того, определение переменной задает её *начальное значение* (явно или неявно). Таким образом, не каждое объявление переменной является определением переменной. К объявлениям, которые не являются определением относятся объявления формальных параметров функций, а также объявления со спецификацией класса памяти **extern**, которые являются ссылкой на переменную, определённую в другом месте программы.

Глобальные переменные

Переменная, объявленная на внешнем уровне, имеет глобальное время жизни. При отсутствии инициализатора такая переменная инициализируется нулевым значением. Область видимости переменной, определенной на внешнем уровне, распространяется от точки, где она определена, до конца исходного файла. Переменная недоступна выше своего определения в том же самом исходном файле. На другие исходные файлы программы область видимости

переменной может быть распространена только в том случае, если ее определение не содержит спецификации класса памяти *static*.

Если в объявлении переменной задана спецификация класса памяти *static*, то в других исходных файлах могут быть определены другие переменные с тем же именем и любым классом памяти. Эти переменные никак не будут связаны между собой.

Спецификация класса памяти *extern* используется для объявления переменной, определенной где-то в другом месте программы. Такие объявления используются в случае, когда нужно распространить на данный исходный файл область видимости переменной, определенной в другом исходном файле на **внешнем** уровне. Область видимости переменной распространяется от места объявления до конца исходного файла. В объявлениях, которые используют спецификацию класса памяти *extern*, инициализация не допускается, так как они ссылаются на переменные, значения которых определены в другом месте.

Локальные переменные

Переменная, объявленная на внутреннем уровне, доступна только в том, блоке в котором она объявлена, *независимо от класса памяти*. По умолчанию она имеет класс памяти *auto*. Переменные этого класса размещаются в стеке. Переменные класса памяти *auto* автоматически не инициализируются, поэтому в случае отсутствия инициализации в объявлении значение переменной класса памяти *auto* считается неопределенным.

Спецификация класса памяти *register* требует, чтобы переменной была выделена память в регистре, если это возможно. Т.к. работа с регистрами происходит быстрее, спецификация класса памяти *register* обычно используется для переменных, к которым предполагается обращаться очень часто.

Для каждого рекурсивного входа в блок порождается новый набор переменных класса памяти *auto* и *register*. При этом каждый раз производится инициализация переменных, в объявлении которых заданы инициализаторы.

Если переменная, объявленная на внутреннем уровне, имеет спецификацию памяти *static*, то область видимости остается прежней, а время жизни становится глобальным. В отличие от переменных класса памяти *auto*, переменные, объявленные со спецификацией класса памяти *static*, сохраняют свое значение при выходе из блока. Переменные класса памяти *static* могут быть инициализированы константным выражением. Если явной инициализации нет, то переменная класса памяти *static* автоматически инициализируется нулевым значением. Инициализация выполняется один раз во время компиляции и не повторяется при каждом входе в блок. Все рекурсивные вызовы данного блока будут разделять единственный экземпляр переменной класса памяти *static*.

Переменная, объявленная со спецификацией класса памяти *extern*, является ссылкой на переменную с тем же самым именем, определенную на **внешнем** уровне в любом исходном файле программы. Цель внутреннего объявления *extern* состоит в том, чтобы сделать определение переменной доступным именно внутри данного блока.

Соккрытие имён

Объявление имени в блоке может скрыть объявление этого имени в охватывающем блоке или глобальное имя. То есть имя может быть замещено внутри блока и будет ссылаться там на другую сущность. После выхода из блока имя восстанавливает свой прежний смысл. К скрытому глобальному имени можно обратиться с помощью операции разрешения области видимости **::**. Скрытое имя члена класса можно использовать, квалифицировав его именем класса. Скрытое

глобальное имя можно использовать, если квалифицировать его унарной операцией разрешения области видимости.

```
int x;

void f()
{ double x = 0;           // Глобальная переменная x скрыта

  ::x = 2;                // Присваивание глобальной переменной x
  x = 2.5;                // Присваивание локальной переменной x
}

int f(int x) { ... }

class X
{ public:
  static int f() { ...
}
};

int ff()
{ return X::f(); }        // Вызов функции f класса X, а не глобальной функции f
```

Не существует способа обращений к скрытой локальной переменной.

Объявления в условиях и цикле `for`

Во избежание случайного неправильного использования переменных их лучше вводить в наименьшей возможной области видимости. В частности, локальную переменную лучше объявлять в тот момент, когда ей надо присвоить значение. В этом случае исключаются попытки использования переменной до момента её инициализации.

Одним из самых элегантных применений этих идей является объявление переменной в условии. Рассмотрим пример.

```
if (double d = f(x))
  y /= d;

  Область видимости переменной d простирается от точки её объявления до конца
оператора, контролируемого условием. Если бы в инструкции if была ветвь else, то областью
видимости переменной d были бы обе ветви.
```

Очевидной и традиционной альтернативой является объявление переменной до условия, но в этом случае область видимости началась бы до места использования переменной и продолжалась бы после завершения её «сознательной» жизни.

```
double d;
...
d2 = d;
...
```



```
if (d = f(x))          // Внимание!!!  
    y /= d;
```

```
...
```

```
d = 2.0;                // Два несвязанных использования переменной d
```

Объявление переменных в условиях, кроме того, что даёт логические преимущества, приводит также к более компактному исходному коду.

Объявление в условии должно объявлять и инициализировать единственную переменную или константу.

Переменную можно также объявить в инициализирующей части оператора *for*. В этом случае область видимости переменной (или переменных) простирается до конца оператора.

```
void f(int x[], int n)  
{ for (int i = 0; i < n; i++) x[i] = i * i; }
```

Если требуется узнать значение индекса после выхода из цикла, переменную надо объявить вне его.

Функции

Объявление функции (прототип) задает её имя, тип возвращаемого значения и атрибуты её формальных параметров. Объявления функций в языке Си имеют следующий синтаксис:

```
[<спецификация класса памяти>] <тип> <имя> (<список формальных  
параметров>);
```

Определение функции специфицирует тело функции, которое представляет собой составной оператор, содержащий объявления и операторы. Определение функции также задает её имя, тип возвращаемого значения и атрибуты её формальных параметров. Определение функции имеет следующий синтаксис:

```
[<спецификация класса памяти>] <тип> <имя> (<список формальных  
параметров>)  
  
{ <тело функции> }
```

Функции имеют глобальное время жизни.

Определение функции может быть задано **только на внешнем уровне**. Область видимости функции распространяется от определения до конца файла. Чтобы использовать вызов функции выше её определения, нужно написать объявление функции (прототип). Прототип некоторой функции может быть расположен на внешнем уровне (тогда вызов этой функции будет возможен из любой функции исходного файла) или же на внутреннем уровне (тогда вызов данной функции будет возможен только из блока, в котором находится прототип).

Для использования функции, расположенной в другом исходном файле, также необходимо использовать прототип.

Если функция определена со спецификацией класса памяти **static**, её использование в других исходных файлах программы невозможно.

Встраиваемые функции компоуются внутренним образом (т.е. не могут быть использованы в других файлах программы), если только явным образом не указана внешняя компоновка с помощью ключевого слова **extern**.

16.Инициализация глобальных и локальных переменных

При инициализации необходимо придерживаться следующих правил:

- а) Объявления содержащие спецификатор класса памяти `extern` не могут содержать инициаторов.
- б) Глобальные переменные всегда инициализируются, и если это не сделано явно, то они инициализируются нулевым значением.
- в) Переменная с классом памяти `static` может быть инициализирована константным выражением. Инициализация для них выполняется один раз перед началом программы. Если явная инициализация отсутствует, то переменная инициализируется нулевым значением.
- г) Инициализация переменных с классом памяти `auto` или `register` выполняется всякий раз при входе в блок, в котором они объявлены. Если инициализация переменных в объявлении отсутствует, то их начальное значение не определено.
- е) Начальными значениями для глобальных переменных и для переменных с классом памяти `static` должны быть константные выражения. Адреса таких переменных являются константами и эти константы можно использовать для инициализации объявленных глобально указателей. Адреса переменных с классом памяти `auto` или `register` не являются константами и их нельзя использовать в инициаторах.

Пример:

```
int global_var;

int func(void) {
    int local_var; /* по умолчанию auto */
    static int *local_ptr=&local_var; /* так неправильно */
    static int *global_ptr=&global_var; /* а так правильно */
    register int *reg_ptr=&local_var; /* и так правильно */
}
```

В приведенном примере глобальная переменная `global_var` имеет глобальное время жизни и постоянный адрес в памяти, и этот адрес можно использовать для инициализации статического указателя `global_ptr`. Локальная переменная `local_var`, имеющая класс памяти `auto` размещается в памяти только на время работы функции `func`, адрес этой переменной не является константой и не

может быть использован для инициализации статической переменной `local_ptr`. Для инициализации локальной регистровой переменной `reg_ptr` можно использовать неконстантные выражения, и, в частности, адрес переменной `local_ptr`.

17.Динамические объекты. Способы

выделения и освобождения памяти

Глобальные, а также статические локальные объекты помещаются в статической памяти, а локальные автоматические объекты размещаются в стеке. Объекты в статической памяти и стеке создаются и удаляются компилятором.

Статическая память очищается при завершении программы, а объекты из стека существуют, пока выполняется блок, в котором они определены.

В дополнение к этим типам в С можно создавать динамические объекты.

Продолжительность их жизни не зависит от того, где они созданы.

Динамические объекты существуют, пока не будут удалены явным образом.

Для управления динамическими объектами в С существует 2 основных функции из стандартной библиотеки: `malloc()` — сокращение от `memory allocate` (выделить память) и `free()` — освободить.

a) `malloc()`. Имеет прототип

```
void *malloc(unsigned s);
```

Выделяет память длиной в `s` байт и возвращает указатель на начало выделенной памяти. В случае неудачного выполнения возвращает `NULL`

b) `free()`. Имеет прототип

```
void *free(void *bl);
```

Освобождает ранее выделенный блок памяти, на начало которого указывает указатель `bl`.

Если мы не используем эту функцию, то динамическая память все равно освободится автоматически при завершении работы программы. Однако все же хорошей практикой является вызов функции `free()`, который позволяет как можно раньше освободить память.

Также используются функции:

`calloc()`. Имеет прототип

```
void *calloc(unsigned n, unsigned m);
```

Выделяет память для `n` элементов по `m` байт каждый и возвращает указатель на начало выделенной памяти. В случае неудачного выполнения возвращает `NULL`

`realloc()`. Имеет прототип

```
void *realloc(void *bl, unsigned ns);
```

Изменяет размер ранее выделенного блока памяти, на начало которого указывает указатель `bl`, до размера в `ps` байт. Если указатель `bl` имеет значение `NULL`, то есть память не выделялась, то действие функции аналогично действию `malloc`

Рациональное использование запрашиваемой памяти:

Пользователь может запрашивать у операционной системы сколько угодно памяти (в рамках технических характеристик компьютера разумеется), но чтобы определить сколько же конкретно необходимо памяти используют функцию из стандартной библиотеки `sizeof()`, выглядит это следующим образом:

Вначале пользователь вводит количество элементов, которое попадает в переменную `n`. После этого необходимо выделить память для данного количества элементов. Для выделения памяти здесь мы могли бы воспользоваться любой из трех вышеописанных функций: `malloc`, `calloc`, `realloc`. Но конкретно в данной ситуации воспользуемся функцией `malloc`:

```
block = malloc(n * sizeof(int));
```

Прежде всего надо отметить, что все три выше упомянутые функции для универсальности возвращаемого значения в качестве результата возвращают указатель типа `void *`. Но в нашем случае создается массив типа `int`, для управления которым используется указатель типа `int *`, поэтому выполняется неявное приведение результата функции `malloc` к типу `int *`.

В саму функцию `malloc` передается количество байтов для выделяемого блока. Это количество подсчитать довольно просто: достаточно умножить количество элементов на размер одного элемента `n * sizeof(int)`.

После выполнения всех действий память освобождается с помощью функции `free()`:

18.Динамические массивы. Особенности выделения и освобождения памяти для многомерных массивов

Определение:

Динамическим называется массив, размер которого может изменяться во время исполнения программы

Выделение памяти в Си (функция *malloc ()*)

Описание:

Функция *malloc ()* определена в заголовочном файле *stdlib.h*. Она используется для инициализации указателей необходимым объемом памяти. Память выделяется из сектора оперативной памяти доступного для любых программ, выполняемых на данной машине. Аргументом ф-и *malloc ()* является количество байт памяти, которую необходимо выделить. Ф-я возвращает указатель на выделенный блок в памяти.

Так как различные типы данных имеют разные требования к памяти, мы как-то должны научиться получить размер в байтах для данных разного типа. Например, нам нужен участок памяти под массив значений типа *int* — это один размер памяти, а если нам нужно выделить память под массив того же размера, но уже типа *char* — это другой размер. Поэтому нужно как-то вычислять размер памяти. Это может быть сделано с помощью операции *sizeof ()*, которая принимает выражение и возвращает его размер. Например, *sizeof (int)* вернет количество байтов, необходимых для хранения значения типа *int*. Рассмотрим пример:

```
#include <stdlib.h>;
```

```
int *ptrVar = malloc ( sizeof (int) );
```

Здесь указателю *ptrVar* присваивается адрес на участок памяти, размер которого соответствует типу данных *int*. Автоматически, этот участок памяти становится недоступным для других программ. А это значит, что после того, как выделенная память станет ненужной, её нужно явно высвободить. Если же память не будет явно высвобождена, то по завершению работы программы, память так и не освободится для операционной системы, это называется утечкой памяти. Также можно определять размер выделяемой памяти, которую нужно выделить, передавая пустой указатель, вот пример:

```
int *ptrVar = malloc (sizeof (*ptrVar) );
```

Что здесь происходит? Операция *sizeof (*ptrVar)* оценит размер участка памяти, на который ссылается указатель. Так как *ptrVar* является указателем на участок памяти типа *int*, то *sizeof()* вернет размер целого числа. То есть, по сути, по первой части определения указателя, вычисляется размер для второй части. Так зачем же это нам

надо? Это может понадобиться, если вдруг необходимо поменять определение указателя *int*, например, на *float* и тогда нам не нужно менять тип данных в двух частях определения указателя. Достаточно будет того, что мы поменяем первую часть:

```
float *ptrVar = malloc (sizeof (*ptrVar));
```

Высвобождение выделенной памяти

Описание:

Высвобождение памяти выполняется с помощью функции *free ()*. Вот пример:

```
free (ptrVar);
```

После освобождения памяти, хорошей практикой является сброс указателя в нуль, то есть присвоить **ptrVar = 0*. Если указателю присвоить *0*, указатель становится нулевым, другими словами, он уже никуда не указывает. Всегда после высвобождения памяти, присваивайте указателю *0*, в противном случае, даже после высвобождения памяти, указатель все равно на неё указывает, а значит вы случайно можете нанести вред другим программам, которые, возможно будут использовать эту память, но вы даже ничего об этом не узнаете и будете думать, что программа работает корректно.

19. Директивы препроцессора.

Макроопределения

Определение:

Препроцессор — это специальная программа, являющаяся частью компилятора языка Си. Она предназначена для предварительной обработки текста программы. Препроцессор позволяет включать в текст программы файлы и вводить макроопределения.

Описание:

Работа препроцессора осуществляется с помощью специальных директив (указаний). Они отмечаются знаком решетки #. По окончании строк, обозначающих директивы в языке Си, точку с запятой можно не ставить.

Основные директивы препроцессора

1. **#include** — вставляет текст из указанного файла;
2. **#define** — задаёт макроопределение (макрос) или символическую константу;
3. **#undef** — отменяет предыдущее определение;
4. **#if** — осуществляет условную компиляцию при истинности константного выражения;
5. **#ifdef** — осуществляет условную компиляцию при определённости символической константы;
6. **#ifndef** — осуществляет условную компиляцию при неопределённости символической константы;
7. **#else** — ветка условной компиляции при ложности выражения;
8. **#elif** — ветка условной компиляции, образуемая слиянием *else* и *if*;
9. **#endif** — конец ветки условной компиляции;
10. **#line** — препроцессор изменяет номер текущей строки и имя компилируемого файла;
11. **#error** — выдача диагностического сообщения;
12. **#pragma** — действие, зависящее от конкретной реализации компилятора.

Директива #include

Описание:

Директива *#include* позволяет включать в текст программы указанный файл. Если файл является стандартной библиотекой и находится в папке компилятора, он заключается в угловые скобки <>.

Если файл находится в текущем каталоге проекта, он указывается в кавычках "". Для файла, находящегося в другом каталоге необходимо в кавычках указать полный путь.

Директива #define

Описание:

Директива *#define* позволяет вводить в текст программы константы и макроопределения.

Общая форма записи:

#define Идентификатор Замена;

Пример:

#define A 3;

Директива *#define* указывает компилятору, что нужно подставить строку, определенную аргументом *Замена*, вместо каждого аргумента *Идентификатор* в исходном файле. Идентификатор не заменяется, если он находится в комментарии, в строке или как часть более длинного идентификатора.

Удобно использовать директиву *#define* для определения размеров массивов. Тогда, если мы захотим глобально поменять размер массива, то достаточно изменить значение в директиве *#define*.

В процессе работы мы можем многократно определять новое значение для одного идентификатора. Но некоторые компиляторы, в частности, *gcc*, могут выдавать предупреждения при повторном определении идентификатора, и чтобы выйти из этой ситуации, мы можем использовать директиву *#undef* для отмены действия макроса. Эта директива имеет следующее определение:

#undef идентификатор

Пример:

#define STRING "Good morning \n";

```
int main(void) {  
    printf (STRING);  
    #undef STRING;  
    #define STRING "Good afternoon \n";  
}
```

Определение:

Все идентификаторы, определяемые с помощью директив *#define*, которые предполагают замену на определенную последовательность символов, называют макроопределениями или макросами.

Описание:

Макросы позволяют определять замену не только для отдельных символов, но и для целых выражений, например:

```
#include <stdio.h>;

#define HELLO printf ("Hello World! \n");

#define FOR for (int i=0; i<4; i++);

int main (void) {

    FOR HELLO;

    return 0;

}
```

Макрос *HELLO* определяет вывод на консоль строки *"Hello World! \n"*. А макрос *FOR* определяет цикл, который отработывает 4 раза.

20. Ввод и вывод символов и строк.

Функции для работы со строками

1. Ввод-вывод символов и строк:

1.1. Ввод-вывод символов:

1.1.1. Ввод символов:

Осуществляется при помощи ф-и *getchar()*.

Её структура:

```
int getchar (void);
```

За одно обращение к ней считывается один символ ввода из текстового потока, где текстовый поток — это последовательность символов, разбитая на строки, каждая из которых содержит 0 или более символов и завершается символом новой строки.

Так, после выполнения:

```
c = getchar ();
```

переменной «с» присваивается символ ввода.

1.1.2. Вывод символов:

Осуществляется при помощи ф-и *putchar()*.

Её структура:

```
int putchar (int c);
```

За одно обращение к ней выводится один символ.

Так, после выполнения:

```
putchar (c);
```

будет напечатано содержимое переменной «с» в виде символа.

1.2. Ввод-вывод строк:

1.2.1. Ввод строк:

Осуществляется при помощи ф-и *gets()*.

Её структура:

```
char *gets (char *s);
```

Вводимые данные помещаются в массив, на который указывает указатель **s*, передаваемый в функцию в качестве параметра. При успешном прочтении строки функция возвращает адрес этого массива *s*, а в случае ошибки возвращается значение *NULL*.

При вводе символов функция *gets()* завершает свою работу при вводе символа «\n», то есть при нажатии на клавишу *Enter*. Но вместо этого символа в строку записывается нулевой символ «\0», который и будет указывать на завершение строки.

1.2.2. Вывод строк:

Осуществляется при помощи ф-и *puts()*

Её структура:

```
int puts (char *s);
```

В качестве параметра передается указатель на строку, а возвращаемым результатом ф-и является последний выведенный символ.

При этом ф-я *puts()* будет выводить символы переданной строки, пока не дойдет до нулевого символа «\0». Если же выводимый массив символов не содержит этого символа, то результат программы не определен.

2. Ф-и для работы со строками:

Здесь «*s*» и «*t*» имеют тип *char*, а «*c*» и «*n*» — тип *int*

1. <i>strcat</i> (<i>s</i> , <i>t</i>)	—	приписывает <i>t</i> в конце <i>s</i> ;
2. <i>strncat</i> (<i>s</i> , <i>t</i> , <i>n</i>)	—	приписывает <i>n</i> символов из <i>t</i> в конец <i>s</i> ;
3. <i>strcmp</i> (<i>s</i> , <i>t</i>)	—	возвращает 1) отрицательное число, если <i>s</i> < <i>t</i> ; 2) нуль, если <i>s</i> == <i>t</i> ; 3) положительное число, если <i>s</i> > <i>t</i> ;
4. <i>strncmp</i> (<i>s</i> , <i>t</i> , <i>n</i>)	—	делает то же, что и <i>strcmp</i> , но кол-во сравниваемых символов <= <i>n</i> ;
5. <i>strcpy</i> (<i>s</i> , <i>t</i>)	—	копирует <i>t</i> в <i>s</i>
6. <i>strncpy</i> (<i>s</i> , <i>t</i> , <i>n</i>)	—	копирует <= <i>n</i> символов из <i>t</i> в <i>s</i>
7. <i>strlen</i> (<i>s</i>)	—	возвращает длину <i>s</i>
8. <i>strchr</i> (<i>s</i> , <i>c</i>)	—	возвращает указатель на первое появление символа « <i>c</i> » в <i>s</i> или, если « <i>c</i> » нет в <i>s</i> , NULL
9. <i>strrchr</i> (<i>s</i> , <i>c</i>)	—	возвращает указатель на последнее появление символа « <i>c</i> » в <i>s</i> или, если « <i>c</i> » нет в <i>s</i> , NULL

21.Текстовые файлы и функции работы с ними

Для работы с файлами существует структура *FILE*, определенная в заголовочном файле *stdio.h*. В этой структуре содержится информация о файле (адрес буфера, положение текущего символа в буфере, открыт файл на чтение или на запись, были ли ошибки при работе с файлом и не встретился ли конец файла). *FILE* аналогичен имени типа (к примеру *int*).

Также для работы с файлами есть несколько ф-й:

1. Открытие/закрытие файла:

1.1. Открытие файла:

fopen ():

Описание: для работы с файлом (чтения и записи) он должен быть открыт с помощью ф-и *fopen ()*. Она получает на вход *имя файла* и возвращает *указатель на файл*, который ссылается на структуру *FILE*. Если обнаружена ошибка, то возвращено будет значение *NULL*.

Структура:

`FILE *fp;` //т.е. fp — это указатель на FILE

`FILE *fopen(char *name, char *mode);` //т.е. fopen () возвращает указатель на файл (fp)

Пример обращения в программе:

`FILE *fp;`

`fp = fopen (имя файла, "режим открытия");`

`... //остальная программа`

Режим открытия поясняет, каким образом пользователь намерен работать с файлом. Есть несколько основных видов режимов:

1. "r" — (read), открытие файла на чтение;
2. "w" — (write), открытие файла на запись, причем, если файл отсутствует, то он будет создан, а если файл имеет содержимое, то оно будет удалено;
3. "a" — (append), открытие файла на добавление, т.е. запись информации в конец уже существующего файла;

1.2. Закрытие файла:

fclose ():

Описание: служит для закрытия ранее открытого файла. В случае успеха возвращает 0, а в случае неудачи — *EOF*.

Структура:

`int fclose (указатель на файл);`

Пример:

`FILE *fp;`

`... //условия, циклы и т.д.`

`fclose (fp);` //программа закрывает открытый ранее файл

return 0; //отчитывается об успешном закрытии файла

2. Запись файла:

2.1. fputc ():

Описание: посимвольная запись в файл. Возвращает код считанного символа или *EOF* при ошибке.

Структура:

int fputc (*символ, поток*);

Пример:

FILE *fp;

...

fputc (*символ, поток*);

...

2.2. fprintf ():

Описание: посимвольная запись в файл. Возвращает кол-во записанных символов или отрицательное число при ошибке.

Структура:

int fprintf (*файл, в который записываем, формат вывода, то, что записываем*);

Пример:

FILE *fp;

...

fprintf (fp, "%с", ...); // «...» — переменная, содержащая то, что нужно ввести в файл

...

3. Считывание из файла:

3.1. fgetc ():

Описание: посимвольное считывание из файла. Возвращает код считанного символа, а при ошибке или конце файла — *EOF*.

Структура:

char fgetc (*поток*) //поток — то, откуда считываем

Пример:

FILE *fp;

...

//условия, циклы и т.д.

fgetc (fp);

...

3.2. fscanf ():

Описание: посимвольное чтение из файла. Возвращает кол-во аргументов, которым были присвоены значения или *EOF* в случае ошибки.

Структура:

int fscanf (*файл, из которого считываем, формат считывания, то, что считываем*);

Пример:

```
FILE *fp;
```

```
...
```

```
fscanf (fp, "%c", ...) // «...» — то, что нужно считать
```

```
...
```


22. Бинарные файлы и функции работы с ними

Текстовые файлы хранят данные в виде текста. Это значит, что если, например, мы записываем целое число 12345678 в файл, то записывается 8 символов, а это 8 байт данных, несмотря на то, что число помещается в целый тип. Кроме того, вывод и ввод данных является форматированным, то есть каждый раз, когда мы считываем число из файла или записываем в файл происходит трансформация числа в строку или обратно. Это затратные операции, которых можно избежать.

Текстовые файлы позволяют хранить информацию в виде, понятном для человека. Можно, однако, хранить данные непосредственно в бинарном виде. Для этих целей используются бинарные файлы.

fwrite

Запись в файл осуществляется с помощью функции

```
size_t fwrite ( const void * ptr, size_t size, size_t count, FILE * stream );
```

Функция возвращает число удачно записанных элементов. В качестве аргументов принимает указатель на массив, размер одного элемента, число элементов и указатель на файловый поток. Вместо массив, конечно, может быть передан любой объект.

```
FILE *output = NULL;  
int number;  
output = fopen("D:/c/output.bin", "wb");  
scanf("%d", &number);  
fwrite(&number, sizeof(int), 1, output);
```

Эта часть кода записывает в бинарный файл введенное пользователем число.

fread

Запись в бинарный файл объекта похожа на его отображение: берутся данные из оперативной памяти и пишутся как есть. Для считывания используется функция fread.

```
size_t fread ( void * ptr, size_t size, size_t count, FILE * stream );
```

Функция возвращает число удачно прочитанных элементов, которые помещаются по адресу ptr. Всего считывается count элементов по size байт.

fseek

Одной из важных функций для работы с бинарными файлами является функция fseek

```
int fseek ( FILE * stream, long int offset, int origin );
```

Эта функция устанавливает указатель позиции, ассоциированный с потоком, на новое положение. Индикатор позиции указывает, на каком месте в файле мы остановились. Когда мы открываем файл, позиция равна 0. Каждый раз, записывая байт данных, указатель позиции сдвигается на

единицу вперёд.

fseek принимает в качестве аргументов указатель на поток и сдвиг в offset байт относительно origin. origin может принимать три значения

SEEK_SET - начало файла

SEEK_CUR - текущее положение файла

SEEK_END - конец файла. К сожалению, стандартом не определено, что такое конец файла, поэтому полагаться на эту функцию нельзя.

В случае удачной работы функция возвращает 0.

В си определён специальный тип fpos_t, который используется для хранения позиции индикатора позиции в файле.

Функция

*int **fgetpos** (FILE * stream, fpos_t * pos);*

используется для того, чтобы назначить переменной pos текущее положение.

Функция

*int **fsetpos** (FILE * stream, const fpos_t * pos);*

используется для перевода указателя в позицию, которая хранится в переменной pos. Обе функции в случае удачного завершения возвращают ноль.

*long int **ftell** (FILE * stream);*

возвращает текущее положение индикатора относительно начала файла. Для бинарных файлов - это число байт, для текстовых не определено (если текстовый файл состоит из однобайтовых символов, то также число байт).

23. Системы счисления. Представления числовых констант в различных системах счисления в языке Си

Целая константа: это десятичное, восьмеричное или шестнадцатеричное число, которое представляет целую величину в одной из следующих форм: десятичной, восьмеричной или шестнадцатеричной.

Десятичная константа состоит из одной или нескольких десятичных цифр, причем первая цифра не должна быть нулем (в противном случае число будет воспринято как восьмеричное).

Восьмеричная константа состоит из обязательного нуля и одной или нескольких восьмеричных цифр (среди цифр должны отсутствовать восьмерка и девятка, так как эти цифры не входят в восьмеричную систему счисления).

Шестнадцатеричная константа начинается с обязательной последовательности 0x или 0X и содержит одну или несколько шестнадцатеричных цифр (цифры представляющие собой набор цифр шестнадцатеричной системы счисления: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)

Примеры целых констант:

<i>Десятичная константа</i>	<i>Восьмеричная константа</i>	<i>Шестнадцатеричная константа</i>
16	020	0x10
127	0117	0x2B
240	0360	0xF0

Если требуется сформировать отрицательную целую константу, то используют знак "-" перед записью константы (который будет называться унарным минусом). Например: -0x2A, -088, -16 .

Каждой целой константе присваивается тип, определяющий преобразования, которые должны быть выполнены, если константа используется в выражениях. Тип константы определяется следующим образом:

- десятичные константы рассматриваются как величины со знаком, и им присваивается тип int (целая) или long (длинная целая) в соответствии со значением константы. Если константа меньше 32768, то ей присваивается тип int в противном случае long.

- восьмеричным и шестнадцатеричным константам присваивается тип int, unsigned int (беззнаковая целая), long или unsigned long в зависимости от значения константы согласно таблице.

<i>Диапазон шестнадцатеричных констант</i>	<i>Диапазон восьмеричных констант</i>	<i>Тип</i>
0x0 - 0x7FFF	0 - 077777	int
0x8000 – 0xFFFF	0100000 - 0177777	unsigned int
0x10000 – 0x7FFFFFFF	0200000 - 01777777777	long

0x80000000 – 0xFFFFFFFF	0200000000000 - 037777777777	unsigned long
-------------------------	------------------------------	---------------

Для того чтобы любую целую константу определить типом long, достаточно в конце константы поставить букву "l" или "L". Пример:

5l, 6l, 128L, 0105L, 0X2A11L.

28. Перевод числе из одной системы счисления в другую

Перевод в десятичную:

~~Для перевода двоичного числа в десятичное необходимо его записать в виде многочлена, состоящего из произведений цифр числа и соответствующей степени числа 2, и вычислить по правилам десятичной арифметики:~~

$$X_2 = A_n \cdot 2^{n-1} + A_{n-1} \cdot 2^{n-2} + A_{n-2} \cdot 2^{n-3} + \dots + A_2 \cdot 2^1 + A_1 \cdot 2^0$$

Пример:

$$11101000_2 = 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 232_{10}$$

~~Для перевода восьмеричного числа в десятичное необходимо его записать в виде многочлена, состоящего из произведений цифр числа и соответствующей степени числа 8, и вычислить по правилам десятичной арифметики:~~

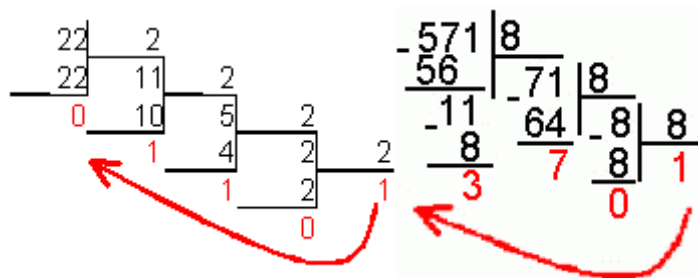
$$75013_8 = 7 \cdot 8^4 + 5 \cdot 8^3 + 0 \cdot 8^2 + 1 \cdot 8^1 + 3 \cdot 8^0 = 31243_{10}$$

Перевод в десятичную из других систем отсчета всегда осуществляется аналогичное, то есть за основание берется число, которое обозначает систему отсчета, а за степень второго множителя его позиция

Перевод из десятичной:

Для перевода десятичного числа в двоичную систему его необходимо последовательно делить на 2 до тех пор, пока не останется остаток, меньший или равный 1. Число в двоичной системе записывается как последовательность последнего результата деления и остатков от деления в обратном порядке.

Пример перевода числа 22(=10110 в двоичной) из десятичной в двоичную и числа 571(=1073) из десятичной в восьмеричную



То есть делим на число, обозначающее новую систему и собираем число с остатка. Если нужно, перед числом в любой системе отсчета можно поставить нули (1011=00001011). Чтобы перевести, например, из двоичной в восьмеричную, можно перевести из двоичной в десятичную, а потом в двоичную.

24.Битовые операции. {Представление чисел в ЭВМ. LSB/MSB, отрицательные числа}

Побитовые И, ИЛИ, НЕ, исключающее ИЛИ

ЗАМЕЧАНИЕ: здесь и далее в примерах используются 8-битные числа для упрощения записи. Всё это верно и для любых других чисел.

Напомню для начала, что логические операции И, ИЛИ, исключающее ИЛИ и НЕ могут быть описаны с помощью таблиц истинности

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

Логический оператор И

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

Логический оператор ИЛИ

X	Y	X XOR Y
---	---	---------

0	0	0
0	1	1
1	0	1
1	1	0

Логический оператор исключающее или

X	NOT X
0	1
1	0

Логический оператор НЕ

В побитовых (bit-wise) операциях значение бита, равное 1, рассматривается как логическая истина, а 0 как ложь. Побитовое И (оператор &) берёт два числа и логически умножает соответствующие биты. Например, если логически умножить 3 на 8, то получим 0

```
char a = 3;
char b = 8;
char c = a & b;
printf("%d", c);
```

Так как в двоичном виде 3 в виде однобайтного целого представляет собой 00000011

а 8
00001000

Первый бит переменной с равен логическому произведению первого бита числа а и первого бита числа b. И так для каждого бита.

```
00000011
00001000
↓↓↓↓↓↓↓↓
00000000
```

Соответственно, побитовое произведение чисел 31 и 17 даст 17, так как 31 это 00011111 , а 17 это 00010001

```
00011111
00010001
↓↓↓↓↓↓↓↓
00010001
```

Побитовое произведение чисел 35 и 15 равно 3.

```
00100011
00001111
```

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
00000011

Аналогично работает операция побитового ИЛИ (оператор |), за исключением того, что она логически суммирует соответствующие биты чисел без переноса.

Например,

```
char a = 15;

char b = 11;

char c = a | b;

printf("%d", c);
```

выведет 15, так как 15 это 00001111, а 11 это 00001011

00001111
00001011
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
00001111

Побитовое ИЛИ для чисел 33 и 11 вернёт 43, так как 33 это 00100001, а 11 это 00001011

00100001
00001011
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
00101011

Побитовое отрицание (оператор ~) работает не для отдельного бита, а для всего числа целиком. Оператор инверсии меняет ложь на истину, а истину на ложь, для каждого бита.

Например,

```
char a = 65;

char b = ~a;

printf("%d", b);
```

Выведет -66, так как 65 это 01000001, а инверсия даст 10111110

что равно -66. Кстати, вот алгоритм для того, чтобы сделать число отрицательным: для нахождения дополнительного кода числа его надо инвертировать и прибавить к нему единицу.

```
char a = 107;

char b = ~a + 1;

printf("a = %d, -a = %d", a, b);
```

Исключающее ИЛИ (оператор ^) применяет побитовую операцию XOR. Например, для чисел

```
char a = 12;

char b = 85;

char c = a ^ b;

printf("%d", c);
```

будет выведено 89, так как a равно 00001100, а b равно 01010101. В итоге получим 01011001

Иногда логические операторы && и || путают с операторами & и |. Такие ошибки могут существовать в коде достаточно долго, потому что такой код в ряде случаев будет работать. Например, для чисел 1 и 0. Но так как в си истиной является любое ненулевое значение, то

побитовое умножение чисел 3 и 4 вернёт 0, хотя логическое умножение должно вернуть истину.

```
int a = 3;

int b = 4;

printf("a & b = %d\n", a & b); //выведет 0

printf("a && b = %d\n", a && b); //выведет не 0 (конкретнее, 1)
```

Операции побитового сдвига

Операций сдвига две – битовый сдвиг влево (оператор `<<`) и битовый сдвиг вправо (оператор `>>`). Битовый сдвиг вправо сдвигает биты числа вправо, дописывая слева нули. Битовый сдвиг влево делает противоположное: сдвигает биты влево, дописывая справа нули. Вышедшие за пределы числа биты отбрасываются.

Например, сдвиг числа 5 влево на 2 позиции
`00000101 << 2 == 00010100`

Сдвиг числа 19 вправо на 3 позиции
`00010011 >> 3 == 00000010`

Независимо от архитектуры (big-endian, или little-endian, или middle-endian) числа в двоичном виде представляются слева направо, от более значащего бита к менее значащему. Побитовый сдвиг принимает два операнда – число, над которым необходимо произвести сдвиг, и число бит, на которое необходимо произвести сдвиг.

```
int a = 12;

printf("%d << 1 == %d\n", a, a << 1);

printf("%d << 2 == %d\n", a, a << 2);

printf("%d >> 1 == %d\n", a, a >> 1);

printf("%d >> 2 == %d\n", a, a >> 2);
```

Так как сдвиг вправо (`>>`) дописывает слева нули, то для целых чисел операция равносильна целочисленному делению пополам, а сдвиг влево умножению на 2. Произвести битовый сдвиг для числа с плавающей точкой без явного приведения типа нельзя. Это вызвано тем, что для си не определено представление числа с плавающей точкой. Однако можно переместить число типа `float` в `int`, затем сдвинуть и вернуть обратно

```
float b = 10.0f;

float c = (float) (*((unsigned int*)&b) >> 2);

printf("%.3f >> 2 = %.3f", b, c);
```

Но мы, конечно же, получим не `5.0f`, а совершенно другое число.

Особенностью операторов сдвига является то, что они могут по-разному вести себя с числами со знаком и без знака, в зависимости от компилятора. Действительно, отрицательное число обычно содержит один бит знака. Когда мы будем производить сдвиг влево, он может пропасть, число станет положительным. Однако, компилятор может сделать так, что сдвиг останется знакопостоянным и будет проходить по другим правилам. То же самое и для сдвига вправо.

```
unsigned int ua = 12;
```

```
signed int sa    = -11;

printf("ua = %d, ua >> 2 = %d\n", ua, ua >> 2);

printf("sa = %d, sa >> 2 = %d\n", sa, sa >> 2);

printf("(unsigned) sa = %u, sa >> 2 = %u\n", sa, sa >> 2);

printf("sa = %d, ((unsigned) sa) >> 2 = %d", sa, ((unsigned) sa) >> 2);
```

В данном случае при первом сдвиге всё работает, как и задумано, потому что число без знака. Во втором случае компилятор VSE2013 оставляет знак. Однако если посмотреть на представление этого числа, как беззнакового, сдвиг происходит по другим правилам, с сохранением самого левого бита. В последней строчке, если привести число со знаком к числу без знака, то произойдёт обычный сдвиг, и мы получим в результате положительное число.

Побитовые операторы и операторы сдвига не изменяют значения числа, возвращая новое. Они также как и арифметические операторы, могут входить в состав сложного присваивания

```
int a = 10;

int b = 1;

a >>= 3;

a ^= (b << 3);
```

и т.д.

(26) Представление чисел в ЭВМ

1. Информация и данные

Информация (от лат. *information* — разъяснение, изложение) — содержание (смысл) сообщения или сигнала, сведения, рассматриваемые в процессе их передачи или восприятия. При помощи компьютеров (ЭВМ) информация может передаваться в различных формах: число, текст, аудио, видео и другие.

Информация воспринимается человеком и может вызвать у него ту или иную реакцию. Разные люди из одного и того же сообщения могут извлечь разное количество информации. Профессиональный музыкант, например, способен по нотной записи музыкального произведения получить больше сведений о его исполнении, чем программист, а программист по тексту программы может рассказать о ее поведении больше, чем музыкант.

Компьютеры имеют дело не с информацией, а с данными. Получив исходные данные, они «механически» перерабатывают их по определенным алгоритмам в выходные данные (результаты), из которых человеку обычно легче извлекать информацию, чем из исходных данных. Получив на входе арифметическое выражение « $(13+27) \cdot 2$ », компьютер на выход выдает эквивалентное выражение «80», значение (смысл) которого человеку воспринять легче (не надо совершать в уме или на бумаге арифметические действия).

2. Представление данных в компьютере

Обычно входные и выходные данные представляются в форме, удобной для человека. Числа люди привыкли изображать в десятичной системе счисления. Для компьютера удобнее двоичная система. Это объясняется тем, что технически гораздо проще реализовать устройства (например, запоминающий элемент) с двумя, а не с десятью устойчивыми состояниями (есть электрический ток — нет тока, намагничен — не

намагничен и т.п.). Можно считать, что одно из двух состояний означает единицу, другое — ноль.

Любые данные (числа, символы, графические и звуковые образы) в компьютере представляются в виде последовательностей из нулей и единиц. Эти последовательности можно считать словами в алфавите $\{0, 1\}$, так что обработку данных внутри компьютера можно воспринимать как преобразование слов из нулей и единиц по правилам, зафиксированным в микросхемах процессора. Такой взгляд роднит вычислительные машины с абстрактными вычислителями. Вспомните машины Тьюринга или нормальные алгоритмы Маркова.

Элемент последовательности из нулей и единиц (член такой последовательности) называют *битом*. Именительный падеж — бит (сокр. от англ. *bit*, *Binary uniT* — двоичный разряд).

Отображение внешней информации во внутреннее представление называется кодированием. *Кодом* (франц. *code*, от лат. *codex* — свод законов) называют как сам способ отображения, так и множество слов (кодовых комбинаций), используемых при кодировании.

3. Представление целых чисел

Для представления чисел в ЭВМ обычно используют битовые наборы — последовательности нулей и единиц фиксированной длины. Организовать обработку наборов фиксированной длины технически легче, чем наборов переменной длины. Позиция в битовом наборе называется *разрядом*. В ЭВМ разрядом называют также часть регистра (или ячейки памяти), хранящую один бит.

3.1. Целые числа без знака

Как определить, какое целое число представляет тот или иной битовый набор? Возможны разные способы. Например, можно считать, что представляемое число равно количеству единиц в битовом наборе («единичная» система счисления). Такой способ позволяет представить всего k различных целых чисел от 0 до $k - 1$, где k — длина набора. Очевидно, что этот способ неэкономный — одному и тому же числу могут соответствовать несколько различных наборов. Количество всевозможных битовых наборов длины k равно 2^k , поэтому выгоднее различным наборам поставить в соответствие различные числа. Это позволит представить 2^k различных чисел. Обычно рассматривают диапазон целых чисел $[N, N + 2^k)$. При $N = 0$ имеем представление беззнаковых (неотрицательных) чисел от 0 до $2^k - 1$.

Существует всего $(2^k)!$ (количество перестановок из 2^k элементов) способов закодировать беззнаковые числа битовыми наборами. Среди всех этих теоретически возможных способов представления чисел наиболее удобен такой: битовый набор, соответствующий числу, является k -разрядной записью этого числа в двоичной системе счисления. Таким образом, можно реализовать арифметические операции над числами, используя известные школьные алгоритмы поразрядной обработки для битовых наборов.

3.2. Целые числа со знаком

Для представления знаковых целых чисел используются три способа:

- 1) *прямой код*;
- 2) *обратный код*;
- 3) *дополнительный код*.

Все три способа используют самый левый (старший) разряд битового набора длины k для кодирования знака числа: знак «плюс» кодируется нулем, а «минус» —

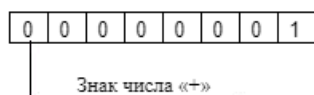
единицей. Остальные $k - 1$ разрядов (называемые *мантиссой* или цифровой частью) используются для представления абсолютной величины числа.

3.2.1. Положительные целые числа (и число 0)

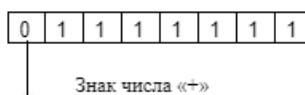
Положительные числа в прямом, обратном и дополнительном кодах изображаются одинаково — цифровая часть содержит двоичную запись числа, в знаковом разряде содержится 0.

Например, для $k = 8$:

Число $1_{10} = 1_2$



Число $127_{10} = 1111111_2$



Диапазон представимых чисел: $0 \dots 2^{k-1} - 1$

3.2.2. Отрицательные целые числа

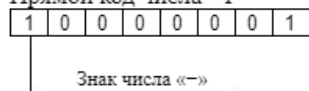
Отрицательные числа в прямом, обратном и дополнительном кодах имеют разное изображение.

3.2.2.1. Прямой код отрицательных чисел

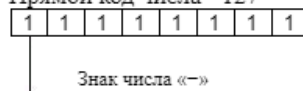
В знаковый разряд помещается цифра 1, а в разряды цифровой части числа — двоичный код его абсолютной величины.

Пример (при $k = 8$):

Прямой код числа -1



Прямой код числа -127



Диапазон представимых чисел: $-(2^{k-1} - 1) \dots 0$

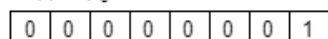
3.2.2.2. Обратный код отрицательных чисел

Получается инвертированием всех цифр двоичного кода абсолютной величины числа, включая разряд знака: нули заменяются единицами, а единицы — нулями.

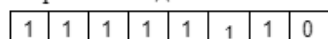
Пример ($k = 8$):

Число -1

Код модуля числа:



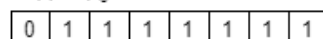
Обратный код числа:



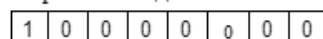
Знак числа «-»

Число -127

Код модуля числа:



Обратный код числа:

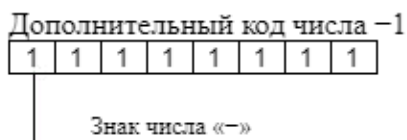


Знак числа «-»

Диапазон представимых чисел: $-(2^{k-1} - 1) \dots 0$

3.2.2.3. Дополнительный код отрицательных чисел

Получается образованием обратного кода с последующим прибавлением единицы к его младшему разряду. Например:



Диапазон представимых чисел: $-2^{k-1} \dots -1$

Заметим, что ноль имеет два представления в прямом и обратном коде, а в дополнительном коде представление нуля единственно.

3.2.3. Вычисление обратного и дополнительного кодов

Один и тот же битовый набор длины k можно интерпретировать по-разному:

- 1) как представление некоторого числа без знака;
- 2) как представление некоторого числа со знаком (в прямом, обратном или дополнительном коде).

Компьютер не знает, что именно представляет тот или иной битовый набор — для него это просто слово в алфавите $\{0, 1\}$, а смысл этого слова известен программисту. Числовым значением такого слова будем называть неотрицательное целое, двоичная (k -разрядная) запись которого совпадает с данным словом.

Пусть x — число со знаком. Тогда числовое значение его обратного и дополнительного кодов можно определить с помощью функций $obr(x)$ и $дон(x)$:

$$obr(x) = \begin{cases} x, & \text{если } x \geq 0 \\ -|x|, & \text{если } x < 0 \end{cases}$$

$2^k - 1$

$$дон(x) = \begin{cases} x, & \text{если } x \geq 0 \\ 2^k - |x|, & \text{если } x < 0 \end{cases}$$

Например, при $k = 8$:

$$obr(+1) = дон(+1) = 1 = 0000$$

$$0001_2; obr(-127) = 255 - 127 = 128 = 1000$$

$$0000_2; дон(-127) = 256 - 127 = 129 = 1000$$

$$0001_2; obr(-1) = 255 - 1 = 254 = 1111$$

$$1110_2; дон(-1) = 256 - 1 = 255 = 1111 1111_2;$$

$$дон(-128) = 256 - 128 = 128 = 1\ 000\ 0000_2. \text{ В обратном коде число } -128 \text{ не}$$

представимо.

Ниже приведена таблица, демонстрирующая различные интерпретации битовых наборов длины 3.

Битовый набор ($k = 3$)	Что представляет Беззнаковое целое	Знаковое целое в прямом коде	Знаковое целое в обратном коде	Знаковое целое в дополнительном коде
0 0 0	0	+ 0	+ 0	+ 0
0 0 1	1	+ 1	+ 1	+ 1
0 1 0	2	+ 2	+ 2	+ 2

0 1 1	3	+ 3	+ 3	+ 3
1 0 0	4	-0	-3	-4
1 0 1	5	-1	-2	-3
1 1 0	6	-2	-1	-2
1 1 1	7	-3	-0	-1

3.2.4. Диапазоны значений целых чисел

Целые числа обычно занимают в памяти компьютера один, два или четыре байта. В суперкомпьютерах могут быть и более «длинные» целые.

Формат числа в байтах	Диапазон			
	Запись с порядком		Обычная запись	
	Со знаком		Без знака	
1	-2 ⁷ .. 2 ⁷ - 1	0..2 ⁸ - 1	-128.. 127	0..255
2	-2 ¹⁵ .. 2 ¹⁵ - 1	0..2 ¹⁶ - 1	-32768.. 32 767	0..65 535
4	-2 ³¹ .. 2 ³¹ - 1	0..2 ³² - 1	-2 147 483 648 .. 2 147 483 647	0..4 294 967 295

3.3. Арифметические действия над целыми числами

Обратный и дополнительный коды применяются особенно широко, так как позволяют упростить конструкцию арифметико-логического устройства (АЛУ) компьютера путем замены некоторых арифметических операций сложением.

Обычно десятичные числа при вводе в машину автоматически преобразуются в двоичный код (целые без знака), обратный или дополнительный код (целые со знаком) и в таком виде хранятся, перемещаются и участвуют в операциях. При выводе результатов из машины происходит обратное преобразование в десятичные числа.

3.3.1. Сложение и вычитание

3.3.1.1. Сложение и вычитание чисел без знака

Сложение и вычитание беззнаковых чисел происходит по обычным для позиционных систем счисления алгоритмам.

Примеры (для $k = 3$):

$$001_2 + 100_2 = 101_2;$$

$$101_2 - 010_2 = 011_2$$

Ситуации, когда уменьшаемое меньше вычитаемого или когда результат суммы не уместится в k разрядов, считаются ошибочными и должны отслеживаться устройством компьютера. Реакция на такие ошибки может быть различной в разных типах компьютеров.

3.3.1.2. Сложение и вычитание чисел со знаком в обратном коде

Сложение в обратном коде происходит следующим образом: по обычному алгоритму складываются все разряды, включая знаковый. Результат такого сложения для k -разрядных наборов имеет длину $k + 1$ (самый левый разряд результата равен единице, если был перенос при сложении старших разрядов операндов, иначе — нулю). Значение левого $k + 1$ -го разряда добавляется к младшему разряду результата. Получаем k -разрядный набор, который и будет суммой двух чисел в обратном коде.

Пример ($k = 3$):

$$+3_{10} + (-1_{10}) = 011_2 + 110_2 = \mathbf{1001_2} \cdot \quad 001_2 + 1 = 010_2 = +2_{10}.$$

Вычитание чисел в обратном коде $x - y$ сводится к сложению $x + (-y)$.

3.3.1.3. Сложение и вычитание чисел со знаком в дополнительном коде

В дополнительном коде сложение происходит так: по обычному алгоритму складываются все разряды, включая знаковый; единица переноса в $k + 1$ -й разряд отбрасывается (т.е. сложение по модулю 2^k).

Пример ($k = 3$):

$$+3_{10} + (-1_{10}) = 011_2 + 111_2 = \mathbf{1010_2} \cdot \quad 010_2 = +2_{10}.$$

При вычитании тоже действует обычный алгоритм, причем если уменьшаемое меньше вычитаемого, к двоичному коду уменьшаемого слева приписывается единица (т.е. добавляется 2^k) и только после этого производится вычитание (такой способ называется вычитание по модулю 2^k).

Пример ($k = 3$):

$$1_{10} - 3_{10} = 001_2 - 011_2 \cdot \quad \mathbf{1001_2} - 011_2 = 110_2 = -2_{10}.$$

Если x и y — числовые значения дополнительного кода знаковых чисел, то числовые значения дополнительных кодов суммы и разности определяются по следующим формулам:

$$(x + y) \bmod 2^k = \begin{cases} x + y, & \text{если } x + y < 2^k, \\ x + y - 2^k, & \text{если } x + y \geq 2^k \end{cases}$$

$$(x - y) \bmod 2^k = \begin{cases} x - y, & \text{если } x \geq y, \\ (2^k + x) - y, & \text{если } x < y \end{cases}$$

3.3.2. Умножение и деление

Во многих компьютерах умножение производится как последовательность сложений и сдвигов. Для этого в АЛУ имеется регистр, называемый накапливающим сумматором, который до начала выполнения операции содержит число ноль. В процессе выполнения операции в нем поочередно размещаются множимое и результаты промежуточных сложений, а по завершении операции — окончательный результат. Другой регистр АЛУ, участвующий в выполнении этой операции, вначале содержит множитель. Затем по мере выполнения сложений содержащееся в нем число уменьшается, пока не достигнет нулевого значения.

Деление для компьютера является трудной операцией. Обычно оно реализуется путем многократного прибавления к делимому дополнительного кода делителя.

3.3.3. Ошибки при выполнении арифметических операций

При выполнении арифметических операций могут возникать ситуации, когда старшие разряды результата операции не помещаются в отведенной для него области памяти. Ниже приводятся примеры ошибочных вычислений ($k = 3$).

Сложение знаковых чисел в обратном коде:

$$-3_{10} + (-2_{10}) = 100_2 + 101_2 = 1001_2 \cdot 2^3 \cdot \quad 001_2 + 1 = 010 = +2_{10}$$

Вычитание знаковых чисел в обратном коде:

$$+2_{10} - (-3_{10}) = 010_2 - 101_2 \cdot 2^3 \cdot \quad 1010_2 - 101_2 = 101_2 = -3_{10}.$$

Такая ситуация называется переполнением цифровой части (мантиссы) формата числа. Для обнаружения переполнения и оповещения о возникшей ошибке в компьютере используются специальные средства. Реакция на разные ошибки может быть разной. Так, в некоторых ЭВМ при делении на ноль вычисления прекращаются (фатальная ошибка), а при переполнении мантиссы устанавливается признак переполнения в так называемом регистре флагов и вычисления продолжают.

4. Представление вещественных чисел

Вещественными числами (в отличие от целых) в компьютерной технике называются числа, имеющие дробную часть. При их изображении во многих языках программирования вместо запятой принято ставить точку. Так, например, число 5 — целое, а числа 5,1 и 5,0 — вещественные. Для удобства отображения чисел, принимающих значения из достаточно широкого диапазона (то есть, как очень маленьких, так и очень больших), используется форма записи чисел с порядком основания системы счисления. Например, десятичное число 1,75 можно в этой форме представить так:

$$1,75 \cdot 10_0 = 0,175 \cdot 10_1 = 0,0175 \cdot 10_2 = \dots,$$

или так:

$$17,5 \cdot 10_{-1} = 175,0 \cdot 10_{-2} = 1750,0 \cdot 10_{-3} = \dots$$

Любое число N в системе счисления с основанием q можно записать в виде $N = M \cdot q^p$, где M называется *мантиссой* числа, а p — *порядком*. Такой способ записи чисел называется представлением с плавающей точкой. Если «плавающая» точка расположена в мантиссе перед первой значащей цифрой, то при фиксированном количестве разрядов, отведённых под мантиссу, обеспечивается запись максимального количества значащих цифр числа, то есть максимальная точность представления числа в машине. Из этого следует, что мантисса должна быть правильной дробью, первая цифра которой отлична от нуля: $M \in [0,1; 1)$. Такое, наиболее выгодное для компьютера, представление вещественных чисел называется *нормализованным*. Мантиссу и порядок q -ичного числа принято записывать в системе с основанием q , а само основание — в десятичной системе.

4.1. Примеры нормализованного представления:

Десятичная система	Двоичная система
$752,15 = 0,75215 \cdot 10_3;$	$-101,01 = -0,10101 \cdot 2_{11}$ (порядок $11_2 = 3_{10}$)
$-0,000039 = -0,39 \cdot 10_{-4};$	$-0,000011 = 0,11 \cdot 2_{-100}$ (порядок $-100_2 = -4_{10}$)

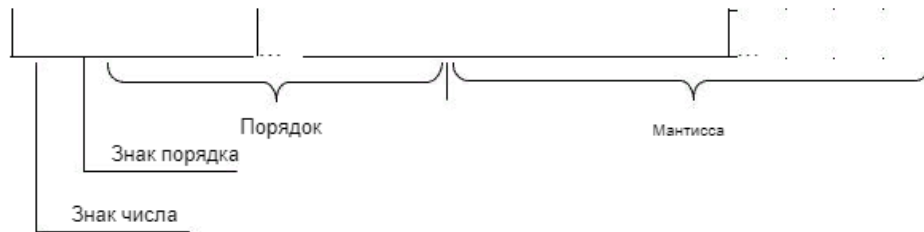
Вещественные числа в компьютерах различных типов записываются поразному. При этом компьютер обычно предоставляет программисту возможность выбора из нескольких числовых форматов наиболее подходящего для конкретной задачи — с использованием четырех, шести, восьми или десяти байтов.

В качестве примера приведем характеристики форматов вещественных чисел, используемых IBM PC-совместимыми персональными компьютерами:

Форматы вещественных чисел	Размер в байтах	Примерный диапазон абсолютных значений	Количество значащих десятичных цифр
Одинарный	4	$10^{-45} \dots 10^{38}$	7 или 8
Вещественный	6	$10^{-39} \dots 10^{38}$	11 или 12
Двойной	8	$10^{-324} \dots 10^{308}$	15 или 16
Расширенный	10	$10^{-4932} \dots 10^{4932}$	19 или 20

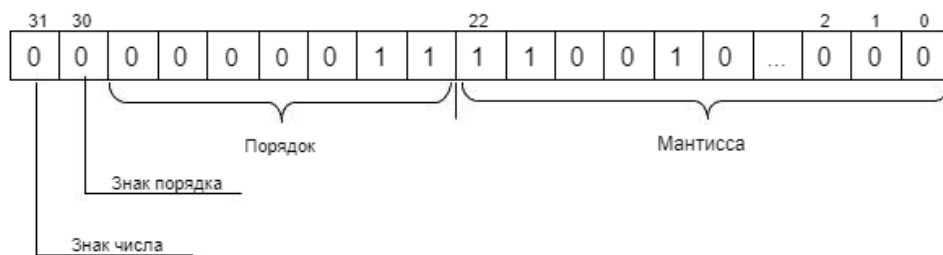
4.2. Представление в виде набора битов

Числа с плавающей точкой представляются в виде битовых наборов, в которых отводятся разряды для мантиисы, порядка, знака числа и знака порядка:

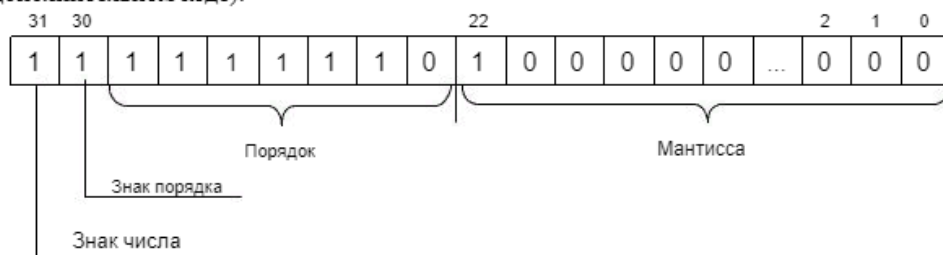


Чем больше разрядов отводится под запись мантиисы, тем выше точность представления числа. Чем больше разрядов занимает порядок, тем шире диапазон от наименьшего отличного от нуля числа до наибольшего числа, представимого в машине при заданном формате.

Покажем на примерах, как записываются некоторые числа в нормализованном виде в четырехбайтовом формате с семью разрядами для записи порядка. Число $6,25_{10} = 110,01_2 = 0,11001_2 \cdot 2^{11}_2$:



Число $-0,125_{10} = -0,001_2 = -0,1_2 \cdot 2^{-10}_2$ (отрицательный порядок записан в дополнительном коде):



4.3. Арифметические действия над нормализованными числами

К началу выполнения арифметического действия операнды операции помещаются в соответствующие регистры АЛУ.

При сложении и вычитании сначала производится подготовительная операция, называемая выравниванием порядков. В процессе выравнивания порядков мантисса числа с меньшим порядком сдвигается в своем регистре вправо на количество разрядов, равное разности порядков операндов. После каждого сдвига порядок увеличивается на единицу.

В результате выравнивания порядков одноименные разряды чисел оказываются расположенными в соответствующих разрядах обоих регистров, после чего мантиссы складываются или вычитаются.

В случае необходимости полученный результат нормализуется путем сдвига мантиссы результата влево. После каждого сдвига влево порядок результата уменьшается на единицу.

Пример 1. Сложить двоичные нормализованные числа $0,10111_2 \cdot 2^{-1}$ и $0,11011_2 \cdot 2^{10_2}$. Разность порядков слагаемых здесь равна трем, поэтому перед сложением мантисса первого числа сдвигается на три разряда вправо:

$$\begin{array}{r} 0,00010111_2 \cdot 2^{10_2} \\ + 0,11011_2 \cdot 2^{10_2} \\ \hline \end{array}$$

$$0,11101111_2 \cdot 2^{10_2}$$

Пример 2. Выполнить вычитание двоичных нормализованных чисел $0,10101_2 \cdot 2^{10_2}$ и $0,11101_2 \cdot 2^1$. Разность порядков уменьшаемого и вычитаемого здесь равна единице, поэтому перед вычитанием мантисса второго числа сдвигается на один разряд вправо:

$$\begin{array}{r} 0,10101_2 \cdot 2^{10_2} \\ - \quad 0,011101_2 \cdot 2^{10_2} \\ \hline 0,001101_2 \cdot 2^{10_2} \end{array}$$

Результат получился не нормализованным, поэтому его мантисса сдвигается влево на два разряда с соответствующим уменьшением порядка на две единицы: $0,1101_2 \cdot 2^0$.

При умножении двух нормализованных чисел их порядки складываются, а мантиссы перемножаются.

Пример 3. Выполнить умножение двоичных нормализованных чисел: $(0,11101_2 \cdot 2^{101_2}) \cdot (0,1001_2 \cdot 2^{11_2}) = (0,11101_2 \cdot 0,1001_2) \cdot 2^{(101_2 + 11_2)} = 0,100000101_2 \cdot 2^{1000_2}$

При делении двух нормализованных чисел из порядка делимого вычитается порядок делителя, а мантисса делимого делится на мантиссу делителя. Затем в случае необходимости полученный результат нормализуется.

Пример 4. Выполнить деление двоичных нормализованных чисел: $0,1111_2 \cdot 2^{100_2} \div 0,101_2 \cdot 2^{11_2} = (0,1111_2 \div 0,101_2) \cdot 2^{(100_2 - 11_2)} = 1,1_2 \cdot 2^1 = 0,11_2 \cdot 2^{10_2}$

Использование представления чисел с плавающей точкой существенно усложняет схему арифметико-логического устройства.

Отрицательные числа

Дополнительный код ([англ. two's complement](#), иногда *twos-complement*) — наиболее распространённый способ представления [отрицательных целых чисел](#) в [компьютерах](#). Он позволяет заменить операцию вычитания на операцию сложения и сделать операции сложения и вычитания одинаковыми для знаковых и [беззнаковых](#) чисел, чем упрощает архитектуру [ЭВМ](#). В англоязычной литературе [обратный код](#) называют *первым дополнением*, а [дополнительный код](#) называют *вторым дополнением*.

Дополнительный код для отрицательного числа можно получить инвертированием его двоичного модуля (первое дополнение) и прибавлением к инверсии единицы (второе дополнение), либо вычитанием числа из нуля.

Дополнительный код (второе дополнение) двоичного числа получается добавлением 1 к младшему значащему разряду его [первого дополнения](#).^[1]

Второе дополнение (англ. [Two's complement](#)) двоичного числа определяется как величина, полученная вычитанием числа из наибольшей степени двух (из 2^N для N-битного второго дополнения).

Представление отрицательного числа в дополнительном коде

При записи числа в дополнительном коде старший разряд является знаковым. Если его значение равно 0, то в остальных разрядах записано положительное [двоичное число](#), совпадающее с [прямым кодом](#).

Двоичное 8-разрядное число *со знаком* в дополнительном коде может представлять любое целое в диапазоне от -128 до +127. Если старший разряд равен нулю, то наибольшее целое число, которое может быть записано в оставшихся 7 разрядах, равно .

Примеры:

Десятичное представление	Двоичное представление (8 бит)		
	прямой	обратный	дополнительный
127	0111 1111	0111 1111	0111 1111
1	0000 0001	0000 0001	0000 0001
0	0000 0000	0000 0000	0000 0000
-0	1000 0000	1111 1111	---
-1	1000 0001	1111 1110	1111 1111
-2	1000 0010	1111 1101	1111 1110
-3	1000 0011	1111 1100	1111 1101
-4	1000 0100	1111 1011	1111 1100
-5	1000 0101	1111 1010	1111 1011

-6	1000 0110	1111 1001	1111 1010
-7	1000 0111	1111 1000	1111 1001
-8	1000 1000	1111 0111	1111 1000
-9	1000 1001	1111 0110	1111 0111
-10	1000 1010	1111 0101	1111 0110
-11	1000 1011	1111 0100	1111 0101
-127	1111 1111	1000 0000	1000 0001
-128	---	---	1000 0000

Дополнительный код для десятичных чисел

Тот же принцип можно использовать и в компьютерном представлении десятичных чисел: для каждого разряда цифра X заменяется на 9-X, и к получившемуся числу добавляется 1. Например, при использовании четырёхзначных чисел -0081 заменяется на 9919 ($9919+0081=0000$, пятый разряд выбрасывается).

При применении той же идеи к привычной 10-ичной системе счисления получится (например, для гипотетического процессора, использующего 10-ичную систему счисления):

10-ичная система счисления ("обычная" запись)	10-ичная система счисления, дополнительный код
...	...
13	0013
12	0012
11	0011
10	0010

10-ичная система счисления ("обычная" запись)	10-ичная система счисления, дополнительный код
9	0009
8	0008
...	...
2	0002
1	0001
0	0000
-1	9999
-2	9998
-3	9997
-4	9996
...	...
-9	9991
-10	9990
-11	9989
-12	9988
...	...

Преобразование в дополнительный код

Преобразование числа из прямого кода в дополнительный осуществляется по следующему алгоритму.

1. Если старший (знаковый) разряд числа, записанного в прямом коде, равен 0, то число положительное и никаких преобразований не делается;
2. Если старший (знаковый) разряд числа, записанного в прямом коде, равен 1, то число отрицательное, все разряды числа, кроме знакового, [инвертируются](#), а к результату прибавляется 1.

Пример. Преобразуем отрицательное число -5 , записанное в прямом коде, в дополнительный код. Прямой код отрицательного числа -5 :

1101

Инвертируем все разряды числа, кроме знакового, получая таким образом [обратный код](#) (первое дополнение) отрицательного числа -5 :

1010

Добавим к результату 1, получая таким образом дополнительный код (второе дополнение) отрицательного числа -5 :

1011

Для преобразования отрицательного числа -5 , записанного в дополнительном коде, в положительное число 5, записанное в прямом коде, используется похожий алгоритм. А именно:

1011

Инвертируем все разряды отрицательного числа -5 , получая таким образом положительное число 4 в прямом коде:

0100

Добавив к результату 1 получим положительное число 5 в прямом коде:

0101

И проверим, сложив с дополнительным кодом

$0101 + 1011 = 10000$, пятый разряд выбрасывается.

p-адические числа

В системе [p-адических чисел](#) изменение знака числа осуществляется преобразованием числа в его дополнительный код. Например, если используется 5-ичная система счисления, то число, противоположное 0001_5 (1_{10}), равно 4444_5 (-1_{10}).

Реализация алгоритма преобразования в дополнительный код (для 8-битных чисел)

C/C++

```
1 int convert(int a) {
2     if (a < 0)
3         a = ~(-a) + 1;
4     return a;
5 }
```

Преимущества и недостатки

Преимущества

- Общие инструкции (процессора) для сложения, вычитания и левого сдвига для знаковых и беззнаковых чисел (различия только в арифметических флагах, которые нужно проверять для контроля переполнения в результате).
- Отсутствие числа «[минус ноль](#)».

Недостатки

- Представление отрицательного числа не читается по обычным правилам, для его восприятия нужен особый навык или вычисления
- В некоторых представлениях (например, [двоично-десятичный код](#)) или их составных частях (например, мантисса числа с [плавающей запятой](#)) дополнительное кодирование неудобно
- Модуль наибольшего числа не равен модулю наименьшего числа. Например, для восьмибитного целого со знаком, максимальное число: $127_{10} = 01111111_2$, минимальное число: $-128_{10} = 10000000_2$. Соответственно, не для любого числа существует противоположное. Операция изменения знака может потребовать дополнительной проверки.

Пример программного преобразования

Если происходит чтение данных из файла или области памяти, где они хранятся в двоичном дополнительном коде (например, файл WAVE), может оказаться необходимым преобразовать байты. Если данные хранятся в 8 битах, необходимо, чтобы значения 128-255 были отрицательными.

C# .NET / C style

```
byte b1 = 254; //11111110 (бинарное)
byte b2 = 121; //01111001 (бинарное)
byte c = 1<< (sizeof(byte)*8-1); //2 возводится в степень 7. Результат:
10000000 (бинарное)
byte b1Conversion=(c ^ b1) + c; //Результат: -2. А фактически, двоичный
дополнительный код.
byte b2Conversion=(c ^ b2) + c; //Результат остаётся 121, потому что
знаковый разряд - ноль.
```

Расширение знака

Расширение знака (англ. [Sign extension](#)) — операция над двоичным числом, которая позволяет увеличить разрядность числа с сохранением знака и значения. Выполняется

добавлением цифр со стороны старшего значащего разряда. Если число положительное (старший разряд равен 0), то добавляются нули, если отрицательное (старший разряд равен 1) — единицы.

Пример

Десятичное число	Двоичное число (8 разрядов)	Двоичное число (16 разрядов)
10	0000 1010	0000 0000 0000 1010
-15	1111 0001	1111 1111 1111 0001

LSB/MSB (САМЫЙ ДУРАЦКИЙ ПУНКТ, ПОТОМУ ЧТО ПО НЕМУ ХЕР НАЙДЕШЬ НОРМАЛЬНУЮ ИНФУ БОЛЬШЕ ОДНОГО АБЗАЦА, Я ДАЖЕ ПОНЯТИЯ НЕ ИМЕЮ, ЧТО ИМЕННО ЗДЕСЬ ТРЕБУЕТСЯ, поэтому еще допишу позже)

Биты в байте принято нумеровать от 0 до 7 и отображать их при табличной записи справа налево. Т.е. *младший бит* (*наименее значимый бит*, англ. *LSB* от *least significant bit*) – бит номер 0 изображают справа, *старший бит* (*наиболее значимый бит*, англ. *MSB* от *most significant bit*) – бит номер 7 – слева (рис. 4.1).

Номер бита	MSB 7	6	5	4	3	2	1	LSB 0
Значение бита	0	1	0	0	0	1	0	1

Рис. 4.1 Графическое изображение байта данных, содержащего число 69. Биты нумеруются справа налево.

25.Препроцессор.Компиляция.Линковка

Препроцессор — это специальная программа, являющаяся частью компилятора языка Си. Она предназначена для предварительной обработки текста программы. Препроцессор позволяет включать в текст программы файлы и вводить макроопределения.

Работа препроцессора осуществляется с помощью специальных директив (указаний). Они отмечаются знаком решетка #. По окончании строк, обозначающих директивы в языке Си, точку с запятой можно не ставить.

Основные директивы препроцессора

#include — вставляет текст из указанного файла
#define — задаёт макроопределение (макрос) или
символическую константу
#undef — отменяет предыдущее определение
#if — осуществляет условную компиляцию при истинности
константного выражения
#ifdef — осуществляет условную компиляцию при
определённости символической константы
#ifndef — осуществляет условную компиляцию при
неопределённости символической константы
#else — ветка условной компиляции при ложности выражения
#elif — ветка условной компиляции, образуемая слиянием else и
if
#endif — конец ветки условной компиляции
#line — препроцессор изменяет номер текущей строки и имя
компилируемого файла
#error — выдача диагностического сообщения
#pragma — действие, зависящее от конкретной реализации
компилятора.

Директива #include

Директива #include позволяет включать в текст программы указанный файл. Если файл является стандартной библиотекой и находится в папке компилятора, он заключается в угловые скобки <>.

Если файл находится в текущем каталоге проекта, он указывается в кавычках ". Для файла, находящегося в другом

каталоге необходимо в кавычках указать полный путь.

```
#include <stdio.h>
#include "func.c"
```

Директива #define

Директива #define позволяет вводить в текст программы константы и макроопределения.
Общая форма записи

```
#define Идентификатор Замена
```

Поля **Идентификатор** и **Замена** разделяются одним или несколькими пробелами.

Директива #define указывает компилятору, что нужно подставить строку, определенную аргументом **Замена**, вместо каждого аргумента **Идентификатор** в исходном файле. Идентификатор не заменяется, если он находится в комментарии, в строке или как часть более длинного идентификатора.

Основные функции препроцессора

[Препроцессором](#) выполняются следующие действия:

- замена соответствующих [диграфов](#) и [триграфов](#) на эквивалентные символы «#» и «\»;
- удаление экранированных [символов перевода строки](#);
- замена строчных и блочных [комментариев](#) пустыми строками (с удалением окружающих пробелов и символов табуляции);
- вставка (включение) содержимого произвольного файла (`#include`);
- [макроподстановки](#) (`#define`);
- условная [компиляция](#) (`#if`, `#ifdef`, `#elif`, `#else`, `#endif`);
- вывод сообщений (`#warning`, `#error`).

Условная компиляция позволяет выбрать код для компиляции в зависимости от:

- модели [процессора](#) (платформы);
- разрядности адресов;
- размерности типов;
- наличия/отсутствия поддержки расширений языка;
- наличия/отсутствия библиотек и/или функций;
- особенностей поведения конкретных функций;
- и другого.

Этапы работы препроцессора:

- [лексический анализ](#) кода C/C++ ([синтаксический анализ](#) не выполняется);

- обработка директив;
- выполнение подстановок:
 - диграфов и триграфов;
 - комментариев;
 - директив;
 - [лексем](#), заданных директивами.

Синтаксис директив

Директивой (командной строкой) препроцессора называется строка в исходном коде, имеющая следующий формат: `#ключевое_слово параметры`:

- ноль или более символов пробелов и/или табуляции;
- символ `#`;
- одно из предопределённых ключевых слов;
- параметры, зависящие от ключевого слова.

Список ключевых слов:

- `define` — создание константы или макроса;
- `undef` — удаление константы или макроса;
- `include` — вставка содержимого указанного файла;
- `if` — проверка истинности выражения;
- `ifdef` — проверка существования константы или макроса;
- `ifndef` — проверка не существования константы или макроса;
- `else` — ветка условной компиляции при ложности выражения `if`;
- `elif` — проверка истинности другого выражения; краткая форма записи для комбинации `else` и `if`;
- `endif` — конец ветки условной компиляции;
- `line` — указание имени файла и номера текущей строки для компилятора;
- `error` — вывод сообщения и остановка компиляции;
- `warning` — вывод сообщения без остановки компиляции;
- `pragma` — указание действия, зависящего от реализации, для препроцессора или компилятора;
- если ключевое слово не указано, директива игнорируется;
- если указано несуществующее ключевое слово, выводится сообщение об ошибке и компиляция прерывается. (В некоторых компиляторах, таких как g++, компиляция продолжается, просто показывая предупреждение)

Описание директив

Вставка файлов (`#include`)[\[править\]](#) | [править код](#)

При обнаружении директив `#include "..."` и `#include <...>`, где «...» — имя файла, препроцессор читает содержимое указанного файла, выполняет директивы и замены (подстановки), заменяет директиву `#include` на директиву `#line` и обработанное содержимое файла.

Для `#include "..."` поиск файла выполняется в текущей папке и папках, указанных в командной строке компилятора. Для `#include <...>` поиск файла выполняется в папках, содержащих файлы стандартной библиотеки (пути к этим папкам зависят от реализации компилятора).

При обнаружении директивы `#include последовательность-лексем` не совпадающей ни с одной из предыдущих форм, рассматривает последовательность лексем как текст, который в результате всех макроподстановок должен дать `#include <...>` или `#include "..."`.

Сгенерированная таким образом директива далее будет интерпретироваться в соответствии с полученной формой.

Включаемые файлы обычно содержат:

- объявления функций;
- объявления глобальных переменных;
- определения [интерфейсов](#);
- определения типов данных;
- и другое.

Директива `#include` обычно указывается в начале файла (в заголовке), поэтому включаемые файлы называются [заголовочными](#).

Пример включения файлов из стандартной библиотеки языка C.

```
#include <math.h> // включение объявлений математических функций
#include <stdio.h> // включение объявлений функций ввода-вывода
```

Использование препроцессора считается неэффективным по следующим причинам:

- каждый раз при включении файлов выполняются директивы и замены (подстановки); компилятор мог бы сохранять результаты препроцессирования для использования в будущем;
- множественные включения одного файла приходится предотвращать вручную с помощью директив условной компиляции; компилятор мог бы выполнять эту задачу самостоятельно.

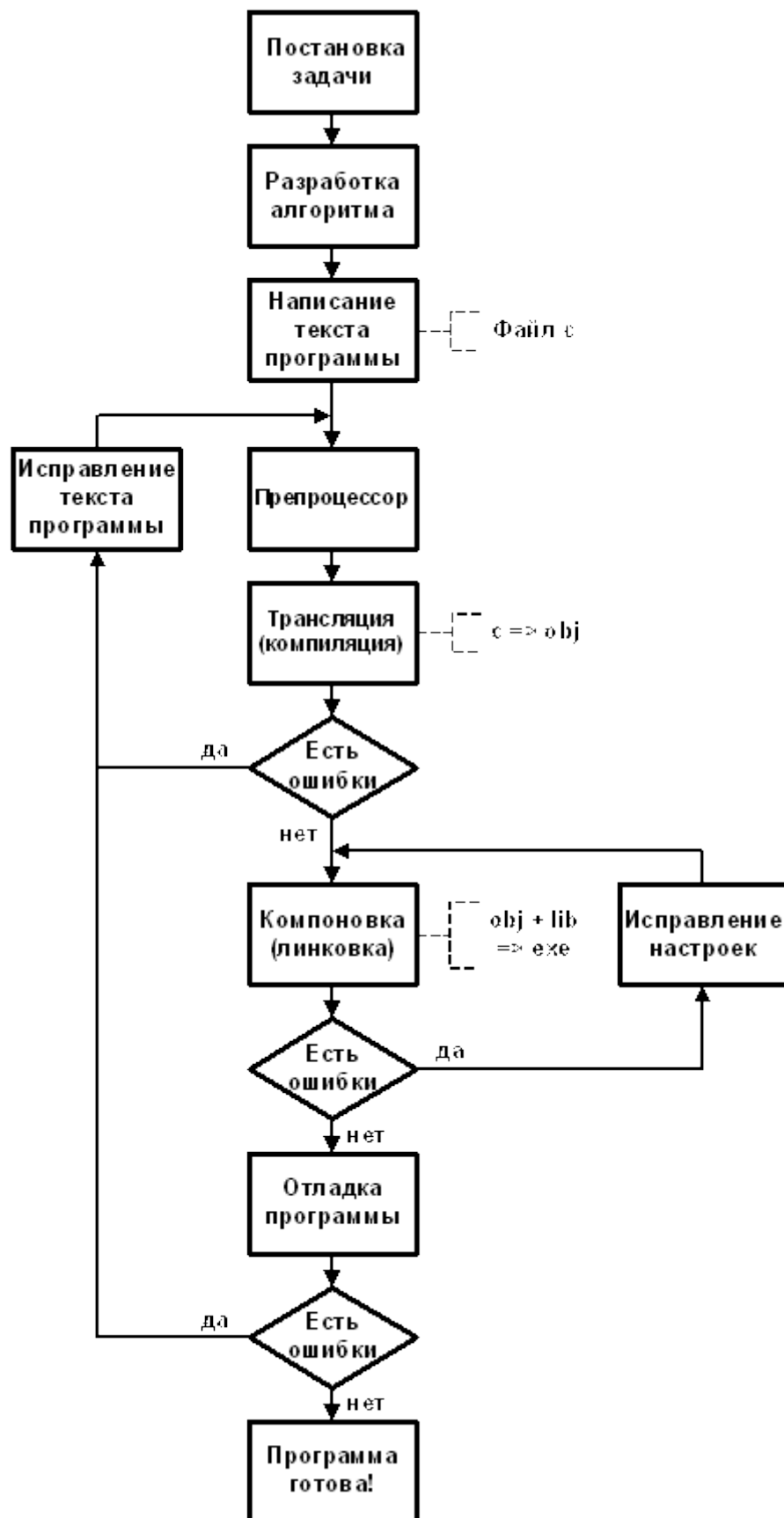
Начиная с 1970-х годов стали появляться способы, заменившие включение файлов. В языках [Java](#) и [Common Lisp](#) используются пакеты (ключевое слово `package`) (см. [package в Java](#)), в языке [Паскаль](#) — [англ. units](#) (ключевые слова `unit` и `uses`), в языках [Modula](#), [OCaml](#), [Haskell](#) и [Python](#) — модули. В языке [D](#), разработанном для замены языков C и C++, используются ключевые слова `module` и `import`.

Для того чтобы исходная программа на Си была переведена в машинный код (файл с расширением `exe` в операционной системе DOS), она должна пройти через три этапа: обработку препроцессором, компиляцию и компоновку.

В задачи препроцессора (`preprocess` – предварительно обрабатывать) входит подключение при необходимости к данной программе на Си внешних файлов, указываемых при помощи директивы `#include`, и подстановку макросов (`macro` – общий, `macros` – макроопределение).

Компилятор (`compile` – собирать) за несколько этапов транслирует (`translate` – переводить) то, что вырабатывает препроцессор, в объектный файл (файл с расширением `obj`), содержащий оптимизированный машинный код, при условии, что не встретились синтаксические или семантические ошибки. Если в исходном файле с программой на Си обнаруживаются ошибки, то программисту выдается их список, в котором ошибки привязываются к номеру строки, в которой они появились. Программист циклически выполняет действия по редактированию и компиляции до тех пор, пока не будут устранены все ошибки в исходном файле.

Компоновщик связывает между собой объектный файл, получаемый от компилятора, с программами из требуемых библиотек и, возможно, с другими файлами. Компоновщик часто называют редактором связей или линковщиком (link – соединять, связывать). В результате компоновки получается файл с расширением exe (exe-файл), который может быть исполнен компьютером. Полученный exe-файл может быть запущен на выполнение из интегрированной среды разработки Турбо Си аналогично запуску из командной строки DOS.



Компиляция!

Компиляция относится к обработке файлов исходного кода (.c, .cc, или .cpp) и созданию объектных файлов проекта. На этом этапе не создается исполняемый файл. Вместо этого компилятор просто транслирует высокоуровневый код в машинный язык. Например, если вы создали (но не скомпоновали) три отдельных файла, у вас будет три объектных файла, созданные в качестве выходных данных на этапе компиляции. Расширение таких файлов будет зависеть от вашего компилятора, например *.obj или *.o. Каждый из этих файлов содержит машинные инструкции, которые эквивалентны исходному коду. Но вы не можете запустить эти файлы! Вы должны превратить их в исполняемые файлы операционной системы, только после этого их можно использовать. Вот тут за дело берётся компоновщик.

Компоновка! (линковка)

Из нескольких объектных файлов создается единый исполняемый файл. На этом этапе полученный файл является единственным, а потому компоновщик будет жаловаться на найденные неопределенные функции. На этапе компиляции, если компилятор не мог найти определение для какой-то функции, считается, что функция была определена в другом файле. Если это не так, компилятор об этом знать не будет, так как не смотрит на содержание более чем одного файла за раз. Компоновщик, с другой стороны, может смотреть на несколько файлов и попытаться найти ссылки на функции, которые не были упомянуты.

Вы спросите, почему этапы компиляции и компоновки разделены. Во-первых, таким образом легче реализовать процесс построения программ. Компилятор делает свое дело, а компоновщик делает свое дело — посредством разделения функций, сложность программы

снижается. Другим (более очевидным) преимуществом является то, что это позволяет создавать большие программы без необходимости повторения шага компиляции каждый раз, когда некоторые файлы будут изменены. Вместо этого, используется так называемая «условная компиляция». То есть объекты составляются только для тех исходных файлов, которые были изменены, для остальных, объектные файлы не пересоздаются. Тот факт, что каждый файл компилируется отдельно от информации, содержащейся в других файлах, существует благодаря разделению процесса построения проекта на этапы компиляции и компоновки.

[Интегрированная среда разработки \(IDE\)](#) эти два этапа берёт на себя и вам не стоит беспокоиться о том, какие из файлов были изменены. IDE сама решает, когда создавать объекты файлов, а когда нет.

Зная разницу между фазами компиляции и компоновки вам будет намного проще находить ошибки в своих проектах. Компилятор отлавливает, как правило, [синтаксические ошибки](#) — отсутствие точки с запятой или скобок. Если вы получаете сообщение об ошибке, множественного определения функции или переменной, знайте, вам об этом сообщает компоновщик. Эта ошибка может означать только одно, что в нескольких файлах проекта определены одна и та же функция или переменная.

Компилятор — [программа](#) или техническое средство, выполняющее *компиляцию*^{[1][2][3]}.

Компиляция — сборка программы, включающая [трансляцию](#) всех модулей программы, написанных на одном или нескольких исходных [языках программирования высокого уровня](#) и/или [языке ассемблера](#), в эквивалентные программные модули на [низкоуровневом языке](#), близком [машинному коду](#) ([абсолютный код](#), [объектный модуль](#), иногда на [язык ассемблера](#))^{[2][3][4]} или непосредственно на машинном языке или ином [двоичнокодовом](#) низкоуровневом командном языке и последующую сборку исполняемой машинной программы. Если компилятор генерирует исполняемую машинную программу на машинном языке, то такая программа непосредственно исполняется физической программируемой машиной (например компьютером). В других случаях исполняемая машинная программа выполняется соответствующей [виртуальной машиной](#). Входной информацией для компилятора ([исходный код](#)) является описание алгоритма или программы на [предметно-ориентированном языке](#), а на выходе компилятора — эквивалентное описание алгоритма на машинно-ориентированном языке (объектный код)^[5].

Компилировать — проводить трансляцию машинной программы с предметно-ориентированного языка на машинно-ориентированный язык.^[3]

Виды компиляторов

- *Векторизующий*. Базируется на трансляторе, транслирующем [исходный код](#) в [машинный код](#) компьютеров, оснащённых [векторным процессором](#).
- *Гибкий*. Сконструирован по [модульному](#) принципу, управляется таблицами и запрограммирован на [языке высокого уровня](#) или реализован с помощью [компилятора компиляторов](#).
- *Диалоговый*. См.: [диалоговый транслятор](#).
- *Инкрементальный*. Пересобирает программу, заново транслируя только изменённые фрагменты программы без перетрансляции всей программы.
- *Интерпретирующий (пошаговый)*. Последовательно выполняет независимую компиляцию каждого отдельного [оператора](#) (команды) исходной программы.
- *Компилятор компиляторов*. Транслятор, воспринимающий формальное описание [языка программирования](#) и генерирующий компилятор для этого языка.
- *Отладочный*. Устраняет отдельные виды [синтаксических ошибок](#).
- *Резидентный*. Постоянно находится в оперативной памяти и доступен для повторного использования многими задачами.
- *Самокомпилируемый*. Написан на том же языке программирования, с которого осуществляется трансляция.
- *Универсальный*. Основан на формальном описании синтаксиса и семантики входного языка. Составными частями такого компилятора являются: ядро, [синтаксический](#) и [семантический](#) загрузчики.

Виды компиляции

- *Пакетная*. Компиляция нескольких исходных модулей в одном задании.
- *Построчная*. Машинный код порождается и затем исполняется для каждой завершённой грамматической конструкции языка. Внешне воспринимается как [интерпретация](#), но устройство имеет иное.
- *Условная*. Компиляция, при которой транслируемый текст зависит от условий, заданных в исходной программе директивами компилятора. (Яркий пример — работа препроцессора языка C и производных от него.) Так, в зависимости от значения некой константы некая заданная часть исходного текста программы транслируется или не транслируется.

Структура компилятора

Процесс компиляции состоит из следующих этапов:

1. Трансляция программы — трансляция всех или только изменённых модулей исходной программы.
2. [компоновка](#) машинно-ориентированной программы.

В первом случае компилятор представляет собой пакет программ, включающий в себя трансляторы с разных языков программирования и компоновщики. Такой компилятор может компилировать программу, разные части исходного текста которой написаны на разных языках программирования. Нередко такие компиляторы управляются встроенным интерпретатором того или иного командного языка. Яркий пример таких компиляторов — имеющийся во всех UNIX-системах (в частности в Linux) компилятор [make](#).

Во втором случае компилятор де-факто выполняет только трансляцию и далее вызывает компоновщик как внешнюю подпрограмму, который и компоует машинно-ориентированную программу. Этот факт нередко служит поводом считать компилятор разновидностью транслятора, что естественно неверно, — все современные компиляторы такого типа поддерживают организацию импорта программой процедуры (функции) из уже оттранслированного программного модуля, написанного на другом языке программирования. Так в программу на C/C++ можно импортировать функцию написанную например [Pascal](#) или [Fortran](#). Аналогично и напротив написанная на C/C++ функция может быть импортирована в Pascal- или Fortran-программу соответственно. Это как правило было бы невозможно без поддержки многими современными компиляторами организации обработки входных данных в процедуру (функций) в соответствии с соглашениями других языков программирования. Например современные компиляторы с языка Pascal помимо соглашения самого Pascal поддерживает организацию обработки процедура/функцией

входных в соответствии с соглашениями языка C/C++. (Чтобы на уровне машинного кода написанная на Pascal процедура/функция работала с входными параметрами в соответствии с соглашениями языка C/C++, — оператор объявления такой Pascal-процедуры/Pascal-функции должен содержать ключевое слово **cdecl**.) Примерами таких компиляторов являются компиляторы со всех без исключения языков программирования, используемые непосредственно.

Трансляция программы как неотъемлемая составляющая компиляции включает в себя:

1. [Лексический анализ](#). На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем.
2. [Синтаксический \(грамматический\) анализ](#). Последовательность лексем преобразуется в дерево разбора.
3. [Семантический анализ](#). Дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д. Результат обычно называется «промежуточным представлением/кодом», и может быть дополненным деревом разбора, новым деревом, абстрактным набором команд или чем-то ещё, удобным для дальнейшей обработки.
4. [Оптимизация](#). Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может быть на разных уровнях и этапах — например, над промежуточным кодом или над конечным машинным кодом.
5. [Генерация кода](#). Из промежуточного представления порождается код на целевом машинно-ориентированном языке.

Генерация кода

Генерация машинного кода

Большинство компиляторов переводит программу с некоторого [высокоуровневого языка программирования](#) в [машинный код](#), который может быть непосредственно выполнен физическим [процессором](#). Как правило, этот код также ориентирован на исполнение в среде конкретной [операционной системы](#), поскольку использует предоставляемые ею возможности ([системные вызовы](#), библиотеки функций). Архитектура (набор программно-аппаратных средств), для которой компилируется (собирается) машинно-ориентированная программа, называется *целевой машиной*.

Результат компиляции — исполнимый программный модуль — обладает максимально возможной производительностью, однако привязан к конкретной операционной системе (семейству или подсемейству ОС) и процессору (семейству процессоров) и не будет работать на других.

Для каждой целевой машины ([IBM](#), [Apple](#), [Sun](#), [Эльбрус](#) и т. д.) и каждой операционной системы или семейства операционных систем, работающих на целевой машине, требуется написание своего компилятора. Существуют также так называемые [кросс-компиляторы](#), позволяющие на одной машине и в среде одной ОС генерировать код, предназначенный для выполнения на другой целевой машине и/или в среде другой ОС. Кроме того, компиляторы могут оптимизировать код под разные модели из одного семейства процессоров (путём поддержки специфичных для этих моделей особенностей или расширений наборов команд). Например, код, скомпилированный под процессоры семейства [Pentium](#), может учитывать особенности распараллеливания инструкций и использовать их специфичные расширения — [MMX](#), [SSE](#) и т. п.

Некоторые компиляторы переводят программу с языка высокого уровня не прямо в машинный код, а на [язык ассемблера](#). (Пример: [PureBasic](#), транслирующий бейсик-код в ассемблер [FASM](#).) Это делается для упрощения части компилятора, отвечающей за генерацию кода, и повышения его переносимости (задача окончательной генерации кода и привязки его к требуемой целевой платформе перекладывается на [ассемблер](#)), либо для возможности контроля и исправления результата компиляции (в т. ч. ручной оптимизации) программистом.

Генерация байт-кода

Результатом работы компилятора может быть программа на специально созданном [низкоуровневом](#) языке двоично-кодовых команд, выполняемых [виртуальной машиной](#). Такой язык называется псевдокодом или [байт-кодом](#). Как правило, он не есть

машинный код какого-либо компьютера и программы на нём могут исполняться на различных архитектурах, где имеется соответствующая виртуальная машина, но в некоторых случаях создаются аппаратные платформы, напрямую выполняющие псевдокод какого-либо языка. Например, псевдокод языка Java называется [байт-кодом Java](#) и выполняется в [Java Virtual Machine](#), для его прямого исполнения была создана спецификация процессора [picoJava](#). Для платформы [.NET Framework](#) псевдокод называется [Common Intermediate Language](#) (CIL), а среда исполнения — Common Language Runtime (CLR).

Некоторые реализации интерпретируемых языков высокого уровня (например, Perl) используют байт-код для оптимизации исполнения: затратные этапы синтаксического анализа и преобразование текста программы в байт-код выполняются один раз при загрузке, затем соответствующий код может многократно использоваться без перекомпиляции.

Динамическая компиляция

Из-за необходимости интерпретации байт-код выполняется значительно медленнее машинного кода сравнимой функциональности, однако он более переносим (не зависит от операционной системы и модели процессора). Чтобы ускорить выполнение байт-кода, используется *динамическая компиляция*, когда виртуальная машина транслирует псевдокод в машинный код непосредственно перед его первым исполнением (и при повторных обращениях к коду исполняется уже скомпилированный вариант).

Наиболее популярной разновидностью динамической компиляции является [JIT](#). Другой разновидностью является [инкрементальная компиляция](#)^[en].

CIL-код также компилируется в код целевой машины JIT-компилятором, а библиотеки [.NET Framework](#) компилируются заранее.

Трансляция байт-кода в машинный код

Трансляция байт-кода в машинный код специальным транслятором байт-кода как указано выше неотъемлемая фаза динамической компиляции. Но трансляция байт-кода применима и для простого преобразования программы на байт-коде в эквивалентную программу на машинном языке. В машинный код может транслироваться как заранее скомпилированный байт-код. Но также трансляция байт-кода в машинный код может выполняться компилятором байт-кода сразу следом за компиляцией байт-кода. Практически всегда в последнем случае трансляция байт-кода выполняется внешним транслятором, вызываемым компилятором байт-кода.

Декомпиляция

Существуют программы, которые решают обратную задачу — перевод программы с низкоуровневого языка на высокоуровневый. Этот процесс называют декомпиляцией, а такие программы — [декомпиляторами](#). Но поскольку компиляция — это процесс с потерями, точно восстановить исходный код, скажем, на C++, в общем случае невозможно. Более эффективно декомпилируются программы в байт-кодах — например, существует довольно надёжный декомпилятор для [Flash](#). Разновидностью декомпиляции является [дизассемблирование](#) машинного кода в код на языке ассемблера, который почти всегда благополучно выполняется (при этом сложность может представлять [самомодефицирующийся код](#) или код, в котором собственно код и данные не разделены). Связано это с тем, что между кодами машинных команд и командами ассемблера имеется практически взаимно-однозначное соответствие.

Раздельная компиляция

Раздельная компиляция ([англ.](#) *separate compilation*) — трансляция частей программы по отдельности с последующим объединением их [компоновщиком](#) в единый загрузочный модуль.^[2]

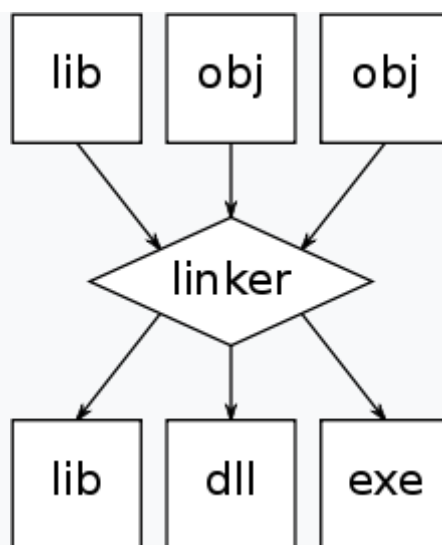
Исторически особенностью компилятора, отражённой в его названии ([англ.](#) *compile* — собирать вместе, составлять), являлось то, что он производил как [трансляцию](#), так и компоновку, при этом компилятор мог порождать сразу [машинный код](#). Однако позже, с ростом сложности и размера программ (и увеличением времени, затрачиваемого на перекомпиляцию), возникла необходимость разделять программы на части и выделять [библиотеки](#), которые можно компилировать независимо друг от друга. В процессе трансляции программы сам компилятор или вызываемый компилятором транслятор порождает [объектный модуль](#), содержащий дополнительную информацию, которая потом — в

процессе компоновки частей в исполнимый модуль — используется для связывания и разрешения ссылок между частями программы. Раздельная компиляция также позволяет писать разные части исходного текста программы на разных языках программирования.

Появление раздельной компиляции и выделение компоновки как отдельной стадии произошло значительно позже создания компиляторов. В связи с этим вместо термина «компилятор» иногда используют термин «транслятор» как его синоним: либо в старой литературе, либо когда хотят подчеркнуть его способность переводить программу в машинный код (и наоборот, используют термин «компилятор» для подчёркивания способности собирать из многих файлов один). Вот только использование в таком контексте терминов «компилятор» и «транслятор» неправильно. Даже если компилятор выполняет трансляцию программы самостоятельно, поручая компоновку вызываемой внешней программе-компоновщику, такой компилятор не может считаться разновидностью транслятора, — транслятор выполняет трансляцию исходной программы и только. И уж тем более не являются трансляторами компиляторы вроде системной утилиты-компилятора [make](#), имеющейся во всех UNIX-системах. Утилита

Собственно утилита [make](#) — яркий пример довольно удачной реализации раздельной компиляции. Работа утилиты **make** управляется сценарием на интерпретируемом утилитой входном языке, известном как [makefile](#), содержащемся в задаваемом при запуске утилиты входном текстовом файле. Сама утилита не выполняет ни трансляцию ни компоновку, — де-факто утилита **make** функционирует как диспетчер процесса компиляции, организующий компиляцию программы в соответствии с заданным сценарием. В частности в ходе компиляции целевой программы утилита **make** вызывает трансляторы с языков программирования транслирующие разные части исходной программы в объектный код, и уже после этого вызывается тот или иной компоновщик, компоновующий конечный исполняемый программный или библиотечный программный модуль. При этом разные части программы, оформляемые в виде отдельных файлов исходно текста, могут быть написаны как на одном языке программирования так и на разных языках программирования. В процессе перекомпиляции программы транслируются только измененные части-файлы исходного текста программы, в следствие чего длительность перекомпиляции программы значительно (порой на порядок) сокращается.

Компоновщик (также *редактор связей*, от [англ.](#) *link editor, linker*) — [инструментальная программа](#), которая производит **компоновку** («линковку»): принимает на вход один или несколько [объектных модулей](#) и собирает по ним [исполнимый модуль](#).



Общая схема процесса линковки. Из объектных файлов и статических библиотек собираются исполняемые файлы или новые статические или динамические библиотеки.

Изначально, до появления [динамических библиотек](#), [загрузчики](#) могли выполнять некоторые функции компоновщика^[1], однако сейчас, чаще всего, загрузка программ выделяется в отдельный [процесс](#)^[2].

Для связывания модулей компоновщик использует [таблицы символов](#), созданные [компилятором](#) в каждом из [объектных модулей](#). Эти таблицы могут содержать символы следующих типов:

- *Определённые* или *экспортируемые* имена — функции и переменные, определённые в данном [модуле](#) и предоставляемые для использования другим модулям;
- *Неопределённые* или *импортируемые* имена — функции и переменные, на которые ссылается модуль, но не определяет их внутри себя;
- *Локальные* — могут использоваться внутри объектного файла для упрощения процесса [настройки адресов](#)^[en].

Для большинства компиляторов, один объектный файл является результатом компиляции одного файла с [исходным кодом](#). Если программа собирается из нескольких объектных файлов, компоновщик собирает эти файлы в единый [исполнимый модуль](#), вычисляя и подставляя [адреса](#) вместо [символов](#), в течение *времени компоновки* (статическая компоновка) или во *время исполнения* (динамическая компоновка).

Компоновщик может извлекать объектные файлы из специальных [коллекций](#), называемых [библиотеками](#). Если не все символы, на которые ссылаются пользовательские объектные файлы, определены, то компоновщик ищет их определения в [библиотеках](#), которые пользователь подал ему на вход. Обычно, одна или несколько системных библиотек используются компоновщиком по умолчанию. Когда объектный файл, в котором содержится определение какого-либо искомого символа, найден, компоновщик может включить его (файл) в исполнимый модуль (в случае статической компоновки) или отложить это до момента запуска программы (в случае динамической компоновки).

Работа компоновщика заключается в том, чтобы в каждом [модуле](#) определить и связать [ссылки](#) на неопределённые имена. Для каждого импортируемого имени находится его определение в других модулях, упоминание имени заменяется на его [адрес](#).

Компоновщик обычно не выполняет проверку типов и количества параметров процедур и функций. Если надо объединить объектные модули программ, написанные на языках со строгой типизацией, то необходимые проверки должны быть выполнены дополнительной утилитой перед запуском редактора связей.