Билеты КПО-2

Оглавление

1. Данные типа int

Целочисленные типы данных:

- точные размеры и диапазоны для целочисленных типов данных стандартом не определены;
- есть знаковые и беззнаковые:
  - модификаторы: signed – знаковый, unsigned – беззнаковый;
  - по умолчанию – знаковый (но неточно).

Данные типа int:

Название типа	Минимальный диапазон
int, signed int, signed	[-32767; 32767]
unsigned int, unsigned	[0; 65535]
short, short int, signed short, signed short int	-32767...32767
unsigned short, unsigned short int	0...65535
long, long int, signed long int, signed long	-(2^31 – 1)...(2^31 – 1)
unsigned long, unsigned long int	0...(2^32 – 1)
long long, long long int, signed long long, signed long long int	-(2^63 – 1)...(2^63 – 1)
unsigned long long, unsigned long long int	0...(2^64 – 1)

Требования стандарта к байту:

- байт (минимальный адресуемый элемент) – адресуемый элемент, достаточный для хранения 1 символа базового набора символов;
- каждый байт можно адресовать независимо;
- количество бит в байте определяется реализацией.

Количество бит в байте задаётся константой CHAR_BIT, определенной в модуле limits.h.
По стандарту минимальное значение CHAR_BIT равно 8.
Минимальный размер байта – 8 бит. Может быть и больше.

Название типа	Минимальный размер
short	2 байта
int	2 байта
long	4 байта
long long	8 байт

Целочисленные типы в Microsoft:

Название типа	Описание
__int8	8 бит, знаковый
unsigned __int8	8 бит, беззнаковый
__int16	16 бит, знаковый
unsigned __int16	16 бит, беззнаковый
__int32	32 бит, знаковый
unsigned __int32	32 бит, беззнаковый
__int64	64 бит, знаковый
unsigned __int64	64 бит, беззнаковый
intN_t	знаковый, равно N
uintN_t	беззнаковый, равно N
int_leastN_t	знаковый, не менее N
uint_leastN_t	беззнаковый, не менее N

Определяются разработчиками компилятора в модуле stdint.h.

Целочисленные литералы:

- Десятичная система счисления: 42, 659, 4843, …
- 8-чная система счисления: 037, 0765, 02314, …
- 16-чная система счисления: 0xAB, 0XC0DE, 0xC001C0DE, …

Могут использоваться постфиксы: 0x15CLU (long unsigned), 0xF48Du (unsigned), 0x42l (long).

Типы целочисленных литералов:

- в зависимости от значения и постфиксов (если есть);
- не менее int.

2. Данные типа char

Целочисленные типы данных:

- точные размеры и диапазоны для целочисленных типов данных стандартом не определены;
- есть знаковые и беззнаковые:
    - модификаторы: signed – знаковый, unsigned – беззнаковый;
    - по умолчанию – знаковый (но неточно).

Данные типа char:

Название типа	Минимальный диапазон
signed char	-127...127
unsigned char	0...255

Тип char может быть как знаковым, так и беззнаковым в зависимости от настроек компилятора.

Требования стандарта к байту:

- байт (минимальный адресуемый элемент) – адресуемый элемент, достаточный для хранения 1 символа базового набора символов;
- каждый байт можно адресовать независимо;
- количество бит в байте определяется реализацией.

Количество бит в байте задаётся константой CHAR_BIT, определенной в модуле limits.h.
По стандарту минимальное значение CHAR_BIT равно 8.
Минимальный размер байта – 8 бит. Может быть и больше.

Размер типа char – 1 байт.

Символьные литералы:

- записываются в апострофах;
- может использоваться префикс L;
- задаёт число, равное коду символа.

Примеры: ‘a’, ‘\0’, ‘\n’, ‘\xCD’ (16 CC), ‘\123’ (8 CC), L’σ’, L’Ͼ’.

Типы символьных литералов:

- с префиксом L – wchar_t (для Unicode);
- без префикса – int.

3. Модификаторы доступа const и volatile

const – переменная не должна изменяться.

Примеры:
```
const int maxCount = 32767;
const int step = 70;
const double pi = 3.1415926536;
const int* p1; // указатель на константный int
int* const p2; // константный указатель на int
```

volatile – обращения к переменной не должны оптимизироваться.
Примеры:
```
volatile unsigned int timer;
volatile unsigned int event;
```
Модификатор может использоваться для работы с аппаратной частью или многопоточностью.

Пример:
```
for (int i = 0; i <= 1000; i++)
{
  ...
  done = i;
}
```

С модификатором:
```
      xor eax, eax
@@:
      ...
      mov [done], eax ; done – метка, задающая переменную
      inc eax
      cmp eax, 1000
      jl @B
```

Без модификатора (один из возможных вариантов):
```
      xor eax, eax
@@:
      ...
      mov reg, eax ; reg – регистр
      inc eax
      cmp eax, 1000
      jl @b
```

4. Данные вещественного типа (с плавающей точкой)

Точные размеры и диапазоны стандартом не определены, заданы только минимальные.
Минимальное основание системы счисления для всех вещественных типов 2.

Название типа	Размер, бит	Название по IEEE 754
`float`	32	Single
`double`	64	Double
`long double`	80	Extended

Компиляторы не обязаны использовать IEEE совместимые форматы.
У Microsoft long double эквивалентен double.

Примеры:

- 1.42, -0.2
- 5.2e-8, -3.5E+6
- 268E16, 14e-6
- 0xDB.84A1p2, 0x4F5.45P4 (p – разделитель мантиссы и порядка)

Типы вещественных литералов:

- без постфикса – double;
- постфикс F – float;
- постфикс L – long double.

5. Элементарный ввод-вывод

Функции консольного вывода:

1. **printf**

   Формат:

   ```
   printf(“формат”, парам1, парам2, …) /
   int printf(const char * restrict format, ...)
   ```

   Примеры:

   ```
   printf(“Hello, world!”);
   printf(“Ответ: %d”, answer);
   printf(“Символ: %c”,  symbol);
   printf(“Получено %f  МБ (%f%%)”, done, done / total * 100);
   ```

   Формат вывода:

Символ	Способ вывода
%d, %i	Целое 10-чное число со знаком
%o	Целое 8-чное число без знака
%u	Целое 10-чное число без знака
%x	Целое 16-чное число без знака (символы: 0123456789abcdef)
%X	Целое 16-чное число без знака (символы: 0123456789ABCDEF)
%f	Вещественное со знаком вида [-]dddd.dddd
%c	Одиночный символ (параметр задаёт код символа)
%s	Строка (параметр задаёт адрес строки)
%%	Символ процента
%p	Указатель (формат вывода зависит от модели памяти)

2. **puts**

   Формат:

   ```
   puts(“<строковые данные>”); /
   int puts(const char *s);
   ```

   Примеры:

   ```
   puts(“Hello, world!”);
   puts(“Выполнено 0%”);
   puts(mystr);
   ```

3. **putchar**

   Формат:

   ```
   putchar(‘<символ>’); /
   int putchar(int c);
   ```

   Примеры:

   ```
   putchar(‘a’);
   putchar(symbol);
   ```

Функции консольного ввода:

1. **scanf**

   Формат:

   ```
   scanf(“формат”, &парам1, &парам2, ...); /
   int scanf(const char * restrict format, ...);
   ```

   Примеры:

   ```
   scanf(“%d”, &y);
   ```

```
scanf(“%c”, &symbol);
```

2. **gets и getchar**
   Формат:
   ```
   gets(str); /
   char *gets(char *s);
   getchar(); /
   int getchar(void);
   ```

   Примеры:
   ```
   char symbol = getchar();
   printf(“%c”, symbol);
   symbol = getchar();
   putchar(symbol);
   ```

6. Арифметические операции

Выражение – это последовательность:
- операндов;
- операций;
- символов-разделителей.

Символы-разделители: [] {} () , ; : … * = #

Операции:
- один или несколько операндов;
- результат вычисления;
- побочные эффекты.

Побочный эффект операции – любые действия, не связанные с вычислением результата.

Арифметические операции:

Символ	Операция	Аналог в Pascal
+	Сложение Сохранение знака	+
-	Вычитание Изменение знака	-
*	Умножение	*
/	Деление	div /
%	Взятие по модулю (остаток)	mod

Примеры:
```
int x = 10, y = 3;
int a = x + y; // 13
int b = z – x; // 3
int c = z * y; // 9
```
Операции деления и взятия остатка:
```
int x = 10, y = 3;
printf(“%d”, x / y); // 3
printf(“%d”, x % y); // 1

float x = 10, y = 3;
printf(“%f”, x / y); // 3.33333
printf(“%f”, x % y); // Ошибка!
```

Инкремент и декремент:
- обозначения:
  - ++ инкремент (увеличить на 1);
  - -- декремент (уменьшить на 1);
- две формы:
  - ++x префиксная;
  - x++ постфиксная.

В **префиксной** форме (++x, --x):
- увеличить/уменьшить;
- значение выражения равно новому значению переменной.

В **постфиксной** форме (x++, x--):
- значение выражения равно старому значению переменной;
- увеличить/уменьшить.

Побочный эффект: запись нового значения в изменяемую переменную.

Примеры:

```
x = 10;
y = ++x;
// x = 11, y = 11

x = 10;
y = x++;
// x = 11, y = 10

x = 42;
y = ++x + 5;
// x = 43, y = 48

x = 42;
y = x++ + 5;
// x = 43, y = 47
```

7. Побитовые логические операции

Побитовые/поразрядные операции:

Символ	Операция	Аналог в Pascal
&	Побитовое И	and
|	Побитовое ИЛИ	or
^	Побитовое исключающее ИЛИ	xor
~	Побитовая инверсия	not
>>	Сдвиг вправо	shr
<<	Сдвиг влево	shl

- вычисляются всегда;
- не имеют побочных эффектов;
- компилятор может изменять порядок вычисления.

Примеры:

```
short i = 0xAB00;
short j = 0xABCD;
short n;

n = i & j; // 0xAB00
n = i | j; // 0xABCD
n = i ^ j; // 0xCD

char a = 65;
char b = ~a; // -66
```

8. Операции сдвига

Формат:

- **сдвиг вправо**: `переменная >> число_сдвигов`
- **сдвиг влево**: `переменная << число_сдвигов`

Принцип работы:

- В случае сдвига влево для освобождаемых правых битов задаётся значение 0.
- В случае сдвига вправо освобождающиеся левые биты заполняются в зависимости от типа первого операнда после преобразования:
  - Если тип имеет значение unsigned, то для них задано значение 0.
  - В противном случае они заполнены копиями бита знака.
- Операция `expr1 << expr2` эквивалентна умножению на 2^{expr2}.
- Операция `expr1 >> expr2` эквивалентна делению на 2^{expr2}, если операнд expr1 не имеет знака или имеет неотрицательное значение.
- Результат операции сдвига не определён, если второй операнд имеет отрицательное значение или если правый операнд больше или равен ширине сдвигаемого левого операнда в битах.

Пример:

```
int x = 7;
x = x << 2; // 28
```

9. Операция присваивания

Формат: `<L-value> = <выражение>`

Поведение:

- результат вычисления – значение правого операнда;
- побочный эффект – левому операнду присваивается значение.

Пример:

```
x = y = z = 0;
```

Составное присваивание:

Полная форма	Сокращённая форма
`x = x + 10;`	`x += 10;`
`x = x - 100;`	`x -= 100;`
`x = x * 2;`	`x *= 2;`
`x = x / 3;`	`x /= 3;`
`x = x & 0x0F;`	`x &= 0x0F;`
`x = x | 45;`	`x |= 45;`
`x = x ^ 0xCC;`	`x ^= 0xCC;`
`x = x << 2;`	`x <<= 2;`
`x = x >> 1;`	`x >>= 1;`

Пример:

```
int x = 42, y = 98;
x ^= y ^= x ^= y;
```

Этот код содержит грубую ошибку, но скорее всего будет работать:

Выражение	x	y
`x ^= y ^= x ^= y;`	x	y
`x ^= y ^= (x ^ y);`	x ^ y	y
`x ^= (x);`	x ^ y	x
`;`	y	x

10. Операция sizeof

Принцип работы:

- возвращает размер переменной или типа в байтах;
- размер определяется по типу операнда;
- имя типа должно быть заключено в скобки;
- результатом является целое число.

Пример:

```
float f;
printf("%d", sizeof f);
printf("%d", sizeof(int));
printf("%d", sizeof(f));
```

11. Преобразование типов в выражениях

Неявное преобразование типов:

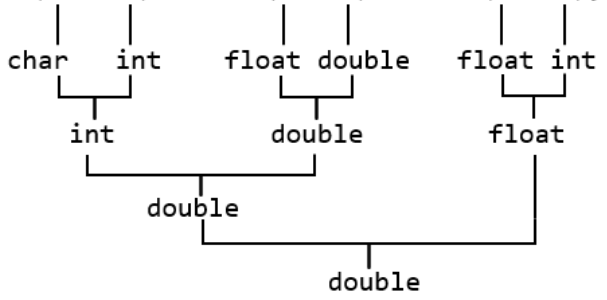
- производится, если в выражении участвуют операнды разных типов;
- операнды приводятся к более «старшему» типу:
  `char -> int -> unsigned int -> long -> unsigned long -> float -> double -> long double`

По стандарту:

- для вещественных типов выбирается более старший;
- целочисленным типам назначается ранг (integer conversion rank):
  - если оба типа знаковые/беззнаковые, то выбирается «более старший»;
  - иначе возможны преобразования «знаковый ⇔ беззнаковый».

Пример:

```
char ch;
int i;
float f;
double d;
result = (ch / i)   +   (f * d)   -   (f + i);
```



12. Операция преобразования типов

Формат: `(тип) выражение`

Принцип работы:

- выполняется явное преобразование типа выражения;
- явное приведение типов ограничено теми же правилами, которые используются для неявных преобразований.

Пример:

```
int x = 5;
float f;

f = x / 2;
printf(“%f”, f); // 2
f = (float) x / 2;
printf(“%f”, f); // 2.5
```

13. Приоритеты в языке Си

1. Постфиксные операции:
    - вызов функции;
    - обращение к элементу массива;
    - обращение к полю структуры (записи);
    - постфиксные инкремент/декремент.
2. Унарные операции:
    - логическое отрицание !x;
    - побитовое отрицание ~x;
    - сохранение знака +x;
    - изменение знака -x;
    - префиксные инкремент/декремент;
    - взятие адреса;
    - обращение по указателю;
    - sizeof x.
3. Явное приведение типа (type) x.
4. Мультипликативные операции:
    - умножение;
    - деление;
    - взятие остатка.
5. Аддитивные операции:
    - сложение;
    - вычитание.
6. Операции побитового сдвига.
7. Операции сравнения > < >= <=
8. Операции сравнения == !=
9. Побитовое И.
10. Побитовое исключающее ИЛИ.
11. Побитовое ИЛИ.
12. Логическое И
13. Логическое ИЛИ.
14. Операция условия ?:
15. Операция присваивания:
    - в том числе составные.
16. Операция запятая.

Для изменения приоритета операторов можно использовать круглые скобки.

14. Оператор if

Синтаксис:

```
// полная форма          // сокращённая форма
if (выражение)           if (выражение)
  оператор_1;              оператор_1;
else
  оператор_2;
```

Принцип работы:

- оператор 1 выполняется, если выражение не равно 0;
- в противном случае выполняется оператор 2.

Примеры:

```
if (a >= b)
  printf("Число a не меньше b.\n");
else
  printf("Число a меньше b.\n");

int x = ...;
if (x)
  printf("x не равно 0.\n");
else
  printf("x равно 0.\n");
```

Особенности работы:

```
C:                       Pascal/Delphi:
if (выражение)           if then
{                        begin
  оператор_1;              оператор_1;
  оператор_2;              оператор_2;
}                        end
else                     else
```

Pascal/Delphi:

- точка с запятой – разделитель:
  `begin оператор_1; оператор_2 end`

C:

- точка с запятой – признак конца оператора;
- закрывающая фигурная скобка } – признак конца составного оператора:
  `{ оператор_1; оператор_2; }`

Пример:

```
int x = 42;
if (x > 5)
{
  if (x < 60)
  {
    x += 2;
  };;;;;          // OK
  x++;
};                // Syntax error
else
  x--;
```

Вложенные операторы if:

- else сопоставляется с ближайшим if, не имеющим else.

Пример:

```
int x = 5, y = 10;
if (x < 5)
  if (y < 10)
    printf("1\n");
  else
    printf("2\n");
```

15. Операции отношения

Работа с логическими значениями:

- в отличие от Pascal в языке С нет полноценного логического типа;
- при проверке на истинность:
  - ненулевое значение – True;
  - нулевое значение – False.
- при вычислении результата:
  - 1, если результат – True;
  - 0, если результат – False.

Операции отношения:

Символ	Операция	Аналог в Pascal
==	Равно	=
!=	Не равно	<>
>	Больше	>
<	Меньше	<
>=	Больше или равно	>=
<=	Меньше или равно	<=

Пример:

```
int x = 93;

if (39 == x)
  printf("then");
else
  printf("else");
```

16. Логические операции

Работа с логическими значениями:

- в отличие от Pascal в языке C нет полноценного логического типа;
- при проверке на истинность:
  - ненулевое значение – True;
  - нулевое значение – False.
- при вычислении результата:
  - 1, если результат – True;
  - 0, если результат – False.

Побитовые и логические операции:

- побитовые операции:
  - вычисляются всегда;
  - компилятор может изменять порядок вычисления операндов.
- логические операции:
  - порядок вычисления операндов строго «слева направо»;
  - сокращённое вычисление.

Логические операции:

Символ	Операция	Аналог в Pascal
&&	Логическое И	and
||	Логическое ИЛИ	or
!	Логическое отрицание	not

Примеры:

```
int x = 2, y = 18;

int x = 45;

if (45 == x || 1 == printf(“1”)) // printf не выполнится
  printf(“1”);

if (45 == x | 1 == printf(“2”)) // printf выполнится
  printf(“2”);

if (x && 18 == y)       // 2 && 1 = 1
  printf(“1”);

if (x & 18 == y)        // 2 & 1 = 0
  printf(“2”);

int x = 42;
printf(“%d\n%d”, ~x, !x);

// Вывод:
// -43
// 0
```

17. Операция запятая

Символ «запятая» в языке С может быть:

- разделителем;
- операцией.

Операция запятая:

- два операнда;
- результат – значение 2-го операнда.

Пример:

```
int x;
int y = (x = 3, 3 * x); // x = 3, y = 9
```

18. Операция условия ?:

Операция условия – эквивалент оператора if для использования в выражениях.

Синтаксис: `операнд_1 ? операнд_2 : операнд_3`

Принцип работы:

- вычислить операнд_1;
- если операнд_1 – истина, результат – операнд_2;
- иначе результат – операнд_3.

Примеры:

```
z = x > 5 ? y : -y;

z = (x > 5 ? 1 : -1) * y;

(a >= b) ? printf(“True”)
         : printf(“False);

printf(a >= b ? “True” : “False”);
```

19. Оператор безусловного перехода goto

Оператор goto позволяет выполнить безусловный переход к метке.
Объявлять метку не требуется.

Ограничение: переход возможен только в пределах функции (из-за наличия стековых фреймов):

- есть и другие ограничения.

Примеры:

```
void main()
{
  int i = 1;
start:
  if (i > 5)
    goto finish;
  printf(“  %d   “, i++);
  goto start;
finish:
  putchar(‘\n’);
}
```

20. Оператор switch

Формат:

```
switch (целое_выражение)
{
  case константа_1: операторы_1; break;
  case константа_2: операторы_2; break;
  ...
  case константа_n: операторы_n; break;
  default: операторы_n+1;
}
```

Ограничения и особенности:

- только для целочисленных типов;
- значения в case – константы;
- может быть указана ветвь default, переход к которой выполняется, если нет подходящего case;
- если не указан break, выполнение переходит в следующую ветвь.

Пример:

```
switch (x)
{
  case 1:
  case 2:
    printf(“Good”);
    break;
  default:
    printf(“Bad”);
}

int a = 1, b = 3;
switch (a)
{
  case 1: b++;
  case 2: b++;
  case 3: b++;
}
printf(“%d”, b); // 6
```

21. Операторы цикла. Оператор for. Оператор while. Оператор do...while

Оператор for
Формат:

- В заголовке – три предложения:
  - первое – инициализация цикла;
  - второе – условие входа в цикл;
  - третье – вычисления в конце итерации.
- Предложения могут быть:
  - выражениями – все три;
  - объявлением переменных – первое.

Примеры:

```
for (int i = 0; i < 5; i++)
  printf(“%d”, i);

int l;
for (l = 5; l != 0; l -= 3)
{
  printf(“%d”, l);
  if (-1 == l)
    l = 3;
}

for (i = 0, sum = 0; i <= 100; i++)
{
  // Вычисление суммы
}

// Бесконечный цикл
for ( ; ; )
{
  // Тело цикла
}
```

Оператор while
Формат:

```
while (выражение)
  оператор;
```

Пример:

```
int k = 2;
while (k <= 100)
  k = k * 2;
printf("%d\n", k);
```

Оператор do...while
Формат:

```
do
  оператор;
while (выражение);
```

В отличие от Pascal/Delphi задаётся условие продолжения, а не выхода из цикла.

Пример:

```
int i = 1;
```

```
do
{
  printf("%d", i);
}
while (++i <= 5);
```

22. Оператор break. Оператор continue

Оператор break:

- может использоваться для досрочного выхода из цикла;
- применяется в составе оператора switch для пропуска последующих ветвей.

Пример:

```
#include <stdio.h>
int main()
{
   char c;
   for( ; ; ) {
      printf( "\nНажмите любую клавишу, Q для выхода: " );
      scanf("%c", &c);
      if (c == 'Q')
          break;
   }
}
```

Оператор continue:

- переход на следующую итерацию ближайшего внешнего оператора цикла.

Пример:

```
for (int i = -5; i <= 5; i++)
{
    if (0 == i)
       continue;
    printf(“%f”, 25.0 / i);
}
```

23. Одномерные массивы и их инициализация

Массивы в C:

- индексация элементов всегда начинается с нуля;
- имя массива может быть неявно преобразовано в константный указатель на первый элемент массива (с индексом 0).

Примеры:

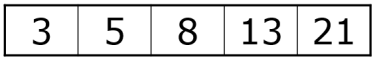
```
float arr[10];
int users[100], items[100];

a = 2;
b = 3;
arr[a + b] = arr[a + b] + 2;

printf(“%f”, arr[0] + arr[1]);
```

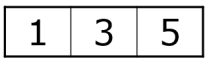
Инициализация одномерных массивов:

- `int arr[5] = {3, 5, 8, 13, 21}`



- `int buf[256] = {0}` ⇔ `int buf[256] = {}`

- `int x[] = {1, 3, 5}`



24. Многомерные массивы и их инициализация

Массивы в С:

- индексация элементов всегда начинается с нуля;
- имя массива может быть неявно преобразовано в константный указатель на первый элемент массива (с индексом 0).

Многомерные массивы:

```
int [5, 8] – ошибка
int[5][8] – правильно
```

Инициализация многомерных массивов:

- ```
  int y[4][3] = {
  {1, 3, 5},
  {2, 4, 6},
  {3, 5, 7}
  }
  ⇔
  int y[4][3] = {
  1, 3, 5, 2, 4, 6, 3, 5, 7
  }
  ```

y[0]	1	3	5
y[1]	2	4	6
y[2]	3	5	7
y[3]	0	0	0

- ```
  int z[4][3] = {
  {1}, {2}, {3}, {4}
  }
  ```

z[0]	1	0	0
z[1]	2	0	0
z[2]	3	0	0
z[3]	4	0	0

25. Объявление указателей. Операции над указателями

здоровья погибшим, кому попадётся

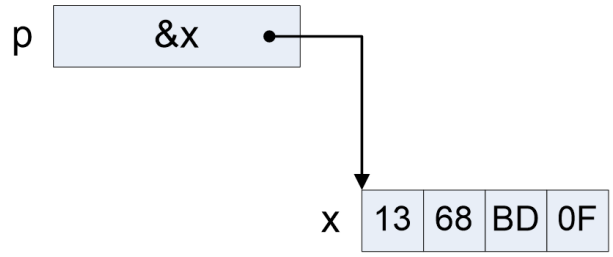
Доступ к данным в памяти:

- прямой (по имени переменной);
- косвенный (через указатель).

Указатель – переменная, значением которой является адрес другой переменной.

Пример:

```
int x;
int *p = &x;
```



Объявление указателей:

- способ 1: `int *p;`
- способ 2: `int* p;`

Пример:

```
int* p1, p2;
```

- p1 – указатель на int;
- p2 – имеет тип int.

Лучше:

```
int *p1, p2;
```

```
тип *идентификатор;
```

Примеры:

```
int *p1; // указатель на int
void *p; // указатель на данные неизвестного типа
```

Нулевой указатель:

- NULL;
- совместим с указателем любого типа;
- фактическое значение зависит от платформы:
- свойства (в стандарте только для литерала 0):
    - `(void *)0 == NULL;`
    - `0 == NULL;`

```
int x = 0;
if (NULL == (void *)x) // не всегда
  ...
if (NULL == x) // не всегда
  ...
```

Операции над указателями:

- взятие адреса;
- косвенная адресация (разыменование указателя);

- увеличение/уменьшение указателя на целое число;
- разность указателей;
- сравнение указателей;
- присваивание указателей.

Взятие адреса:

```
int *p;
int x;

p = &x;
```

```
var
  p: ^Integer;
  x: Integer;

p := @x;
```

`&(a + 1)` – выражение не имеет адреса.

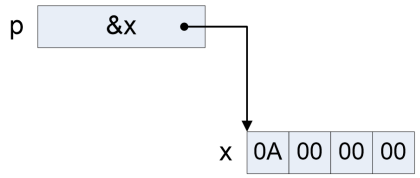
`&5` – константа не имеет адреса.

```
int c[40];
&c:
```

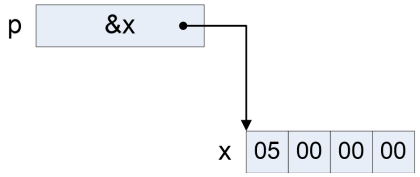
- имя массива не является указателем на его первый элемент;
- может компилироваться (в этом случае игнорируется).

Косвенная адресация (разыменование указателя):

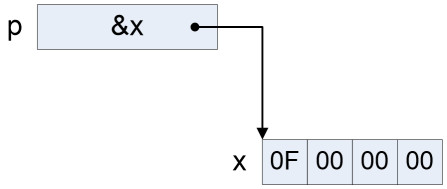
```
int x = 10;
int *p;
p = &x;
```




```
*p = 5;
```




```
*p = *p * 3;
```




```
printf("%d", x);  // 15
printf("%d", *p); // 15
```

Аналог в Pascal/Delphi – p^.

Увеличение/уменьшение указателя на целое число:

```
int *p;
int a[10];
p = &a[0];
p = p + 5;
// значение p увеличилось на 5 * sizeof(int) = 10, т.е. p указывает на a[5]
p++;
p--;
```

Разность указателей:

```
int *p1, *p2;
int x;

p1 = &a[0];
p2 = &a[5];
x = p2 – p1;
// x будет присвоено значение 5
```

Сравнение указателей:

```
p1 = &a[0];
p2 = &a[3];
if (p1 > p2)
  printf(“Случилось невозможное!”);
```

Применение вычислительных и сравнительных операций над указателями ограничивается только в случае, если участвующие в процессе указатели указывают на память в пределах одного объекта (переменной, массива, структуры и т.п.).
Если один из указателей выходит за пределы объекта, то результат <u>не гарантируется</u>.

Присваивание указателей:

```
p1 = p;
```

Ограничения:

- указатели должны иметь один и тот же тип;
- для присваивания указателей разных типов можно использовать приведение типов;
- исключение: тип void *.

26. Тип void

Тип void:

- множество значений типа данных – пустое;
- нет значений, нет операций, нет свойств значений;
- данный тип не является полноценным типом данных и используется для обозначения отсутствия информации о типе.

void и void *:

- ключевое слово void обозначает отсутствие информации о типе:
  `void *p;`
- указателю void * можно присваивать указатели любых других типов;
- аналог void * в Pascal/Delphi – тип Pointer;
- с указателем void * нельзя применять операцию разыменования указателя.

Пример:

```
unsigned short a = 0x1234;
void *p = &a;
char b;
unsigned short c;

b = *(char *)p;
c = *(unsigned short *)p;

printf(“%02X\n”, b); // 34
printf(“%04X\n”, c); // 1234
```

27. Связь между указателями и массивами

```
int a[5];
int *p;
```

Связь между массивами и указателями:

- присваивание указателю адреса первого элемента:
  - `p = a;`
  - `p = &a[0];`
- доступ к элементам массива:
  - `a[3]`
  - `*(p + 3)`
- имя массива – указатель:

```
int x1 = *(a + 3);
int x2 = a[3];
```

- указатель можно использовать как имя массива:

```
int x3 = *(p + 3);
int x4 = p[3];
```

- изменение массива как указателя:

```
p += 3; // работает
a += 3; // ошибка
```

- при проведении операций над массивами происходит неявное преобразование к типу указатель, который указывает на первый элемент массива (для большинства, кроме взятия адреса и sizeof):

```
int a[3][3];
a[0] -> a[0][0] a[0][1] a[0][2]
a[1] -> a[1][0] a[1][1] a[1][2]
a[2] -> a[2][0] a[2][1] a[2][2]

a[i] + j == &a[i][j]
*(a[i] + j) == a[i][j]
*(*(a + i) + j) == a[i][j]
```

28. Динамическое распределение памяти

Функции динамического распределения памяти доступны при подключении stdlib.h.

Прототипы функций:

```
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
```

Пример:

```
char *p;
float *c;

p = (char *)malloc(1000); // выделяется область памяти размером 1000 байт
...
free(p);

  c = (float *)calloc(10, sizeof(float)); // выделяется область памяти
                                          размером 10 * <размер float>
                                          байт и заполняется нулями
...
free(c);
```

Пример:

```
#include <stdlib.h>
void main()
{
      double **a;
      int n, m, i;

      scanf("%d%d", &m, &n);
      a = (double **)calloc(m, sizeof(double *));
      for (i = 0; i < m; i++)
            a[i] = (double *)calloc(n, sizeof(double));
      ...
      for (i = 0; i < m; i++)
            free(a[i]);
      free(a);
}
```

Все функции выделения памяти в случае ошибки возвращают NULL.

`realloc`:

- если параметр-указатель равен NULL, работает аналогично `malloc`;
- если новый размер меньше старого, часть данных теряется;
- если новый размер больше старого, новые байты заполняются нулями.

Пример:

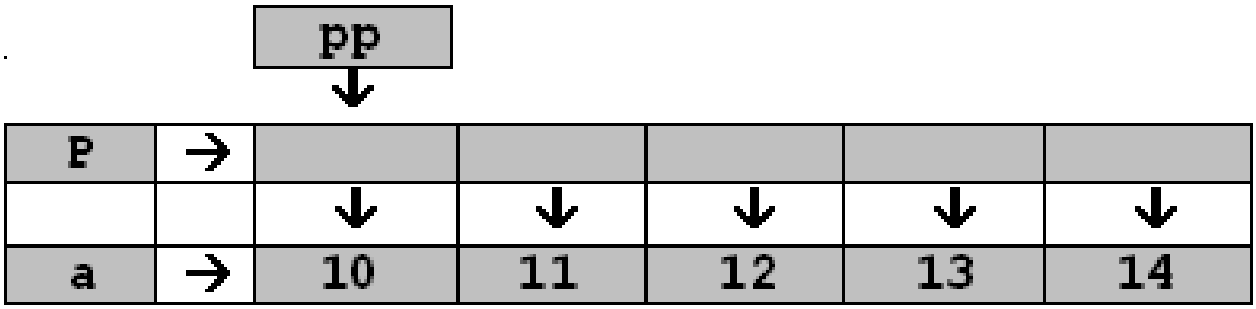
```
int *p;

p = (int *)malloc(1024);
...
  p = (int *)realloc(p, 1 << 30); // надо вводить новую переменную-указатель
...
  free(p);
```

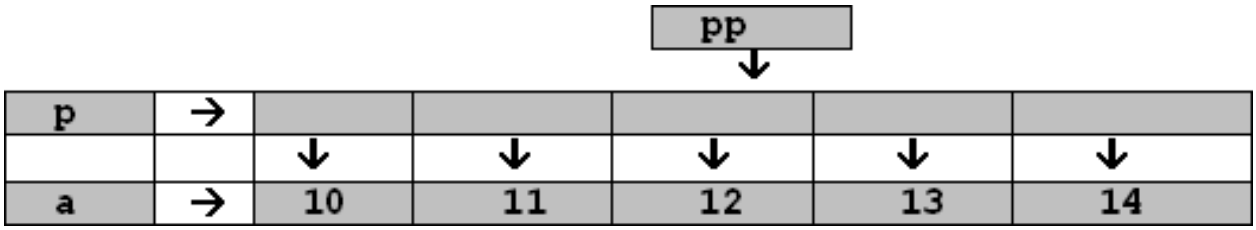
Пример:

```
int a[] = {10, 11, 12, 13, 14};
int *p[] = {a, a + 1, a + 2, a + 3, a + 4};
int **pp = p;
```

Представление данных в памяти:




```
pp += 2;
```



30. Функции. Область видимости переменных. Локальные и глобальные переменные

Объявление функций

Объявление функции состоит из:

- прототипа функции;
- тела функции.

Прототип – «заголовок» функции.

Прототип включает:

- имя функции;
- тип возвращаемого значения;
- список и типы параметров.

Примеры:

- `int myFunction();`
  - параметры: нет
  - возвращаемое значение: int
- `void myFunction();`
  - параметры: нет
  - возвращаемое значение: нет (void)
- `int myFunction(int a, float b);`
  - Параметры: a (тип int), b (тип float)
  - возвращаемое значение: int

Опережающее объявление функции:

```
int myFunc2();

int myFunc1()
{
	...
	myFunc2();
	...
}

int myFunc2()
{
	...
	myFunc1();
	...
}
```

Возврат значения из функции:

- осуществляется с помощью оператора return;
- не используется в функциях с типом возвращаемого значения void.

Пример:

```
int getGCD(int a, int b)
{
	while (0 != a && 0 != b)
	{
		if (a > b)
			a = a % b;
		else
			b = b % a;
	}
	return a + b;
}
```

Области видимости переменных

Область видимости переменной – блок, в котором она объявлена.

- Блок – последовательность операторов между фигурными скобками (область видимости локальных переменных).
- Если объявление вне блоков, область видимости – файл (область видимости глобальных переменных).

Область видимости метки – функция.

Пример:

```
#include <stdio.h>

int x = 5, y = 7;     // объявление глобальных переменных

void myFunc()
{
  int y = 10, x = 15; // объявление локальных переменных
  printf("\n x from myFunc(): %d", x);
  printf("\n y from myFunc(): %d", y);
}

void main()
{
  printf("x from main(): %d\n", x);
  printf("y from main(): %d\n", y);
  myFunc();
  printf("Return from myFunc\n");
  printf("x from main(): %d\n", x);
  printf("y from main(): %d\n", y);
}
```

31. Передача параметров в функцию

Передача параметров

В языке С параметры всегда передаются по значению:

- все остальные способы передачи организуют, используя те или иные возможности языка.

Пример:

```
void swap(int x, int y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}

void swapP(int *x, int *y)
{
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}

void swapA(int b[])
{
  int temp;
  temp = b[0];
  b[0] = b[1];
  b[1] = temp;
}
```

```
void main()
{
  int a[2] = {4, 5}, x = 4, y = 5;
  swap(x, y);
  printf("После swap(): %d %d\n",
         x, y);  // 4 5
  swapP(&x, &y);
  printf("После swapP(): %d %d\n",
         x, y);  // 5 4
  swapA(a);
  printf("После swapA(): %d %d\n",
         x, y);  // 5 4
}
```

Прототип функции

В прототипе можно не указывать имена параметров:

- `int myFunc(int a, float b);`
- `int myFunc(int, float).`

При описании самой функции:

- имена параметров должны быть указаны;
- их типы должны соответствовать указанным в прототипе.

Если функция без параметров, можно указывать пустые скобки:

```
int myFunc();
```

В новых версиях стандарта это называется old-style и считается устаревшим.
Ключевое слово void вместо списка параметров означает, что функция – без параметров:

```
int someFunc(void);
```

Объявление переменной и опережающее описание функции имеют один и тот же синтаксис.

Пример:

```
int f(void);
int *fip(float x);
int f(void), *fip(float x);
```

32. Рекурсивные функции

Рекурсия – это определение какого-либо понятия через само это понятие.
Существует два **вида рекурсии**:

- явная – рекурсия, при которой обращение к функции содержится в теле самой функции;
- неявная (взаимная) – рекурсия, при которой обращение к функции содержится в теле другой функции, к которой производится обращение из данной функции. При организации взаимной рекурсии возникает проблема:
  - вызов функции возможен только после ее описания;
  - проблема решается с помощью опережающих описаний.

При каждом рекурсивном вызове (активации рекурсивной функции) создаётся новый набор:

- фактических параметров;
- локальных переменных.

На время выполнения текущего рекурсивного вызова данные всех предыдущих вызовов становятся недоступными.
Рекурсивность является не свойством функции, а лишь свойством её описания. Ту же функцию можно организовать и без рекурсии.
Преимущество рекурсии: как правило, запись такой функции короче и нагляднее.
Недостаток рекурсии: как правило, такая функция выполняется медленнее, чем эквивалентная ей нерекурсивная.

Пример:

```
#include <stdio.h>

int fib(int);

void main()
{
  int n, answer;
  printf(“Введите порядковый номер числа: ”);
  scanf(“%d”, &n);
  answer = fib(n);
  printf(“\nОтвет: %d”, answer);
}

int fib(int n)
{
  if (n < 3)
    return 1;
  else
    return(fib(n – 2) + fib(n – 1));
}
```

33. Использование функций в качестве параметров функций

Результат функции как параметр

```
int cube (int h, int s)
{
      return h * s;
}

int square(int l, int w)
{
      return l * w;
}
...
int answer = cube(high, square(length, width));
...
```

Указатель на функцию как параметр

```
int sum(int x, int y)
{
    return x + y;
}

int subtract(int x, int y)
{
    return x - y;
}

int operation(int (*op)(int, int), int a, int b)
{
    return op(a, b);
}
...
int a = 42, b = 65;
int result1 = operation(sum, a, b), result2 = operation(substract, a, b);
...
```

34. Указатели на функции

Объявление переменной-указателя на функцию:

```
// указатель на функцию long someFunc(int)
long (*pFunc)(int);
// без скобок будет объявлять функцию p, возвращающую указатель на значение
   типа long
```

Указатель на функцию содержит адрес первого байта в памяти, по которому располагается выполняемый код функции.

Пример:

```
#include <stdio.h>
void mult(int, int);
void main()
{
      int x, y;
      void (*p)(int, int);

      p = mult;
      scanf(“%d %d”, &x, &y);
      p(x, y);
}

void mult(int x, int y)
{
      printf(“%d”, x * y);
}
```

35. Структура программы на языке С

Программа на языке С представляет собой последовательность объявлений переменных и функций.
- кроме того, могут встречаться специальные директивы.

Объявление переменных:

```
<Объявление_переменной> ::=
      <Задание_типа> <Список_переменных> “;”.
<Список_переменных> ::=
      <Переменная> {“,” <Переменная>}.
<Переменная> ::=
      <Идентификатор> [“=” <Инициализатор>].
```

Примеры:

```
int a = 8, b;
signed long int x;
unsigned char mychar;
_int16 somevar;
```

Объявление функций

Объявление функции состоит из:
- прототипа функции;
- тела функции.

Прототип – «заголовок» функции.

Прототип включает:
- имя функции;
- тип возвращаемого значения;
- список и типы параметров.

Примеры:

```
int f(void);
int *fip(float);

int *fip(float x)
{
  int *y;
  ...
  return y;
}

int f(void)
{
  int x;
  ...
  return x;
}
```

36. Передача параметров в функцию main() и её возвращаемое значение

Функция с именем main является **точкой входа** в программу:
- т.е. выполнение начинается с неё.

В зависимости от компилятора и типа проекта она может называться иначе.

Параметры:
- количество аргументов, переданных в командной строке при запуске;
- сами аргументы.

Возвращаемое значение – код результата:
- 0 – успешно;
- другое значение – признак ошибки.

Виды:
- `int main()`
- `void main()`
- `int main(int argc)`
- `void main(int argc)`
- `int main(int argc, char **argv)`
- `void main(int argc, char **argv)`
- `int main(int argc, char *argv[])`
- `void main(int argc, char *argv[])`

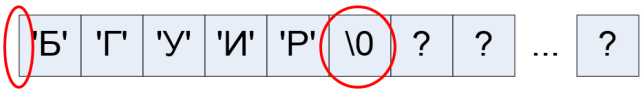
37. Строки

не дай боже

Строковые данные в языке С:

- в отличие от Pascal в языке С нет полноценного строкового типа;
- вместо строкового типа используются массивы символов.

Представление строк в С – **строки с терминальным элементом** (элемент с кодом 0):



Unicode:

- для хранения символа может отводиться больше 1 байта.

В C/C++ для поддержки Unicode вводятся дополнительные типы:

- wchar (wchar_t)
- TCHAR

Для Unicode-строк написаны отдельные реализации функций.

- для TCHAR тоже.

Пример:

- `strcpy`
- `strcpy_s`
- `wcscpy_s`
- `_tcscpy_s`

Строки в языке С

Пример:

```
char *str = “Hello, world!”;
char str[] = “Hello, world!”;
char str[] = {‘H’, ‘e’, ‘l’, ...};
char str[] = {72, 101, 108, ...};
```

Модификаторы типов

Пример:

```
char *s                 = “Hello!”;
char *const s           = “Hello!”; // константный указатель
const char *s           = “Hello!”; // указатель на константу
const char *const s     = “Hello!”; // всё вместе
```

Строковые функции

В современных реализациях стандартной библиотеки С:

- «классические» функции:
  - принимают просто указатель на строку;
- safe-функции:
  - имеют дополнительные параметры, задающие размеры строк;
  - имеют постфикс _s в имени.

Строковые функции с * доступны при подключении библиотеки string.h.

1. Функция strcpy()*

Формат:

```
char *strcpy(
      char *dest,
      const char *src
);
```

Принцип работы: копирует строку из src в dest.

Возвращаемое значение: dest.

Пример реализации:
```
char *user_strcpy(char *dest, const char *src)
{
  char *result = dest;
  do
  {
    *dest++ = *src;
  }
  while (*src++);
  return result;
}
```

2. Функция strlen()*
Формат:
```
size_t strlen(
      const char *src
);
```

Принцип работы: вычисляет длину строки src.

Возвращаемое значение: количество символов в строке src, предшествующих терминальному символу.

Пример реализации:
```
size_t user_strlen(const char* src)
{
  size_t result = 0;
  while (*src++) result++;
  return result;
}
```

3. Конкатенация строк
Пример:
```
char *str1 = “Student”;
char *str2 = “BSUIR”;
char *str3;

str3 = str1 + str2; // нельзя
```

Функция strcat()*
Формат:
```
char *strcat(
      char *dest,
      const char *src
);
```

Принцип работы:
- конкатенирует строки dest и src;
- терминальный символ из конца строки dest перезаписывается строкой src.

Возвращаемое значение: dest.

Пример реализации:
```
char *user_strcat(char *dest, const char* src)
{
  strcpy(dest + strlen(dest), src);
  return dest;
}
```

4. Функция strchr()*
Формат:
```
char *strchr(
      char *s,
      int c
);
```

Принцип работы: находит первое вхождение символа c в строке s.

Возвращаемое значение:
- указатель на найденный символ, если c содержится в s;
- NULL в противном случае.

Пример реализации:
```
char *user_strchr(char *s, int c)
{
  do
  {
    if (*s == c) return s;
  }
  while (*s++);
  return 0;
}
```

5. Функция strdup()
Формат:
```
char *strdup(
      const char *s
);
```

Принцип работы: создаёт копию строки s.

Возвращаемое значение:
- указатель на копию строки s, если создание копии успешно;
- NULL в противном случае.

Не является частью стандарта языка C, но поддерживается многими компиляторами.

Пример реализации:
```
char *user_strdup(const char *s)
{
  char *result = malloc(strlen(s) + 1);
  return result ? strcpy(result, s) : 0;
}
```

Использование:
```
char *temp = strdup(str);
...
free(temp);
```

6. Функция strcmp()*
Формат:
```
int strcmp(
      const char *s1,
      const char *s2
);
```

Принцип работы: сравнивает строки s1 и s2.

Возвращаемое значение:
- положительное целое число, если s1 больше s2;
- 0, если s1 и s2 равны;
- отрицательное целое число, если s1 меньше s2.

Пример реализации:
```
int strcmp(const char *s1, const char *s2)
{
  int result;
  do
  {
    result = *s1 - *s2;
  }
  while (*s1++ && *s2++ && !result);
  return result;
}
```

7. Функция ssprintf()
Формат:
```
int sprintf(
      char *s,
      const char *format,
      ...
);
```

Принцип работы: аналогичен printf за исключением того, что данные записываются в строку s, а не в стандартный поток вывода stdout.

Возвращаемое значение:
- количество записанных символов без учёта терминального, если запись успешна;
- отрицательное число в противном случае.

6. Функция sscanf()
Формат:
```
int sscanf(
      const char *s,
      const shar *format,
      ...
);
```

Принцип работы: аналогичен scanf за исключением того, что данные считываются из строки s, а не из стандартного потока ввода stdin.

Возвращаемое значение:
- количество считанных переменных, если чтение успешно;
- EOF в противном случае.

38. Классы хранения и видимость переменных. Классы auto и register

Классы хранения:

- автоматический (auto);
- регистровый (register);
- статический (static);
- внешний (extern).

Блок – последовательность операторов между фигурными скобками (область видимости локальных переменных).
Если объявление вне блоков, область видимости – файл (область видимости глобальных переменных).

Автоматический класс хранения

Переменная автоматического хранения:

- создаётся при входе в блок;
- уничтожается при выходе из блока;
- до инициализации имеет неопределённое значение.

Используется по умолчанию для локальных переменных.

Пример:

```
int f1(void)
{
  auto int x = 10;
}
int f2(void)
{
  int x = 35;
}
int main(void)
{
  printf(“x = %d”, x); // не скомпилируется
  return 0;
}
```

В C++11 ключевое слово `auto` изменило семантику: используется для указания, что тип переменной определяется по инициализатору: `auto x = someFunc(a, b, c);`

Регистровый класс хранения:

- время жизни и область видимости аналогичны auto-переменным;
- по возможности переменной ставится в соответствие регистр.

Пример:

```
void main()
{
      register long sum = 0;
      for (register int i = 1; i <= 100; i++)
            sum = sum + i;
      printf(“\nsum[100] = %d”, sum);
}
```

Классы хранения:

- только для локальных переменных:
  - `auto`
  - `register`
- для локальных и глобальных переменных:
  - `static`
  - `extern`

Класс хранения	Место хранения	Время жизни	Область видимости
auto	стек	блок	блок
register	регистры (если доступны)	блок	блок
static (локальная переменная)	сегмент/секция данных	программа	блок
static (глобальная переменная)	сегмент/секция данных	программа	файл
extern	сегмент/секция данных	программа	блок/программа

39. Классы хранения и видимость переменных. Классы static и extern

Классы хранения:

- автоматический (auto);
- регистровый (register);
- статический (static);
- внешний (extern).

Блок – последовательность операторов между фигурными скобками (область видимости локальных переменных).
Если объявление вне блоков, область видимости – файл (область видимости глобальных переменных).

Статический класс хранения

Переменные статического класса хранения размещаются в статической памяти:

- т.е. в секции/сегменте данных.

Могут быть:

- локальные;
- глобальные.

Пример:

```
int f1(void);
int f2(void);

void main(void)
{
  int i;
    for (i = 1; i <= 5; i++)
    {
      printf("%d %d\n", f1(), f2());
    }
}
int f1(void)
{
  static int i = 0; // инициализируется только в момент создания (запуска
                       программы)
  i++;
  return i;
}
int f2(void)
{
  int i = 0;
  i++;
  return i;
}
```

Область видимости глобальной переменной с классом хранения `static` – файл.

- но есть нюансы.

`static` – класс хранения по умолчанию для глобальных переменных.
Переменные с классом хранения `static` по умолчанию инициализируются нулевыми значениями.

Внешний класс хранения

Используется, чтобы:

- сделать доступной в блоке переменную, определённую глобально;
- использовать глобальные переменные, объявленные в разных файлах.

Пример:

```
...
void f1()
{
  extern double i;
  ...
}
double i = 9.36;
...
// A.c
...
int x = 42;
void f1() { ... }
...
// B.c
...
extern int x;
void f2() { ...; x += 5; ... }
...
```

Ограничение:

- нельзя задавать начальное значение (инициализировать).

Классы хранения:

- только для локальных переменных:
  - `auto`
  - `register`
- для локальных и глобальных переменных:
  - `static`
  - `extern`

Класс хранения	Место хранения	Время жизни	Область видимости
auto	стек	блок	блок
register	регистры (если доступны)	блок	блок
static (локальная переменная)	сегмент/секция данных	программа	блок
static (глобальная переменная)	сегмент/секция данных	программа	файл
extern	сегмент/секция данных	программа	блок/программа

40. Структуры. Инициализация структурных переменных. Вложенные структуры

Структура – совокупность переменных (возможно, различных типов), сгруппированных под одним именем.

Пример:
```
struct TItem
{
  int n;
  char name[20];
  float cost;
  int count;
  char note[100];
}
struct TItem curr;
```

Объявление структурного типа имеет ту же область видимости, что и объявление переменных:
- т.е. блок (если объявлен в блоке) или файл (если объявлен вне блока).

Объявление структурных переменных
```
struct item
{
  int n;
  char name[20];
  float cost;
  int count;
  char note[100];
} m1, m2, m3;
```

Инициализация структурных переменных
- ```
  struct item m1 =
  {1, “генератор”, 100.5, 100, “частота 460 Гц”};
  ```
- ```
  struct item
  {
    int n;
    char name[20];
    float cost;
    int count;
    char note[100];
    } m1 = {1, “генератор”, 1.2, 100, “сдан в ремонт”};
  ```

Обращение к полям структур
```
m1.n = 1;
m1.name = “генератор”; // так нельзя
strcpy(m1.name, “генератор”);
m1.cost = 1.2;

struct item *p1 = &m1;
(*p1).cost = 1.5; ⇔ p1->cost = 1.5;
```

Присваивание структур
```
struct item m1, m2, m3;
m1 = m2;
m1 = m2 = m3;
```

Вложенные структуры

```
struct item1

{
  int count;
  float cost;
  char note[100];
}

struct item2
{
  int n;
  float cost;
  char note[100];
}
```

```
struct item3
{
  float cost;
  char note[100];
}

struct item1
{
  int count;
  struct item3 common;
}

struct item2
{
  int n;
  struct item3 common;
}
```

```
struct item2 m;
m.common.cost = 19.84;
```

41. Указатели на структуры. Массивы структурных переменных

Указатель на структуру

```
struct item *p;

p = (struct item *)malloc(sizeof(struct item));
...
free(p);

++p->cost ⇔ ++(p->cost)

(++p)->cost ⇔ (p++)->cost
```

Массивы структур

```
struct item arr[100];
...
arr[i].n = 10;
arr[i].note[j] = ‘a’;

p = &arr[0];
(p + i)->n = 10;
```

42. Передача функциям структурных переменных. Класс хранения typedef

Передача функциям структурных переменных:

```
struct item
{
  int n;
  char name[20];
  float cost;
  int count;
  char note[100];
} m1;
```

- ```
  mult(m1.cost, m1.count);
  ...
  float mult(float cost, int count)
  {
    return (cost * count);
  }
  ```
- ```
  mult(m1);
  ...
  float mult(struct item m1)
  {
    return (m1.cost * m1.count);
  }
  ```
- ```
  mult(&m1);
  ...
  float mult(struct item *pItem)
  {
    return (pItem->cost * pItem->count);
  }
  ```
- ```
  struct item arr[5];

  f1(arr, 5);

  int f1(struct item arr[], int nItems)
  {
    ...
    arr[i].n = 1;
    ...
  }
  ```
- ```
  struct item arr[5];

  f1(arr, 5);

  int f1(struct item *p, int nItems)
  {
    ...
    (p + i)->n = 1;
    ...
    p[i].n = 1;
    ...
  ```
- ```
  }
  ```

Класс хранения typedef
Формат:

```
typedef тип новое_имя_типа;
```

Пример:

```
typedef int integer;
typedef int *PINTEGER;

typedef struct _item // _item – тег, прописывать необязательно
{
  ...               // выше есть, лень писать
} item;

item m1, m2, m3; ⇔ struct _item m1, m2, m3;
```

Возможности языка C:

```
typedef int Integer;
int typedef Integer;
short unsigned typedef word;
typedef enum {c};       // только в C
typedef;              ┐
typedef int;          ├ // только в версиях языка до стандарта C89
typedef int short;    ┘
```

43. Битовые поля

Битовые поля в структурах

```
struct
{
  unsigned int y:1;
  unsigned m:1;
  int k:2;
} stat;

stat.y = 0;
```

Работа с битовыми полями:

- диапазон значений поля соответствует его размеру;
- поля размером 1 бит всегда беззнаковые;
- битовые поля по возможности упаковываются в одну и ту же единицу хранения;
- есть 2 способа заставить компилятор упаковывать битовые поля с новой единицы хранения:
  - добавление битового поля в 0 бит либо не-битового поля:

```
struct test0
{
  unsigned int x:5;
  unsigned int al:0;
  unsigned int:0;
  unsigned int y:8;
};
```

```
struct testN
{
  unsigned int x:5;
  unsigned int al;
  unsigned int y:8;
}
```

  - добавление вложенной структуры с битовым полем:

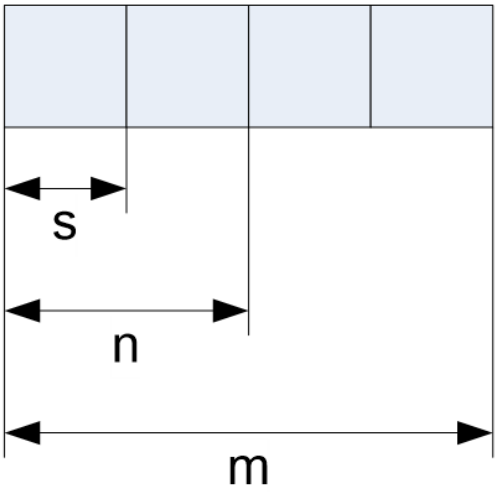
```
struct
{
  char a;
  int b:5, c:11, :0, d:8;
  struct { int ee:8; } e;
}
```

44. Объединения

Объявление объединения

```
union cell
{
  short n;
  float m;
  char s;
} var;
```

Представление объединения в памяти

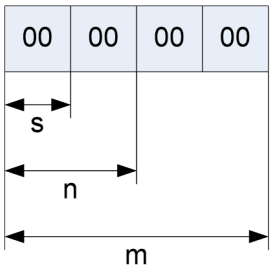


Обращение к полям объединения:

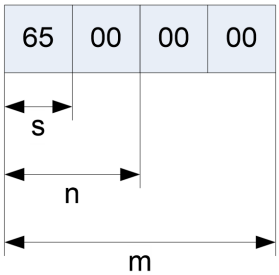
```
var.m = 1;

void f1(union cell *p)
{
  ...
  p->m = 2;
  ...
}

var.m = 0;
```




```
var.s = 0x65;
```



45. Перечисления

Объявление перечислений:

```
enum month {Jan, Feb, Mar, ...};
// Pascal/Delphi
type
  month = (Jan, Feb, Mar, ...);
```

Представление значений:

- нумерация значений начинается с 0;
- можно изменять числовое представление значений:
  - в этом случае все последующие значения также будут изменены.

Объявление переменных:

```
enum month m1;

m1 = Jan;
```

Пример:

```
if ((m1 < Mar) || (m1 > Nov))
  printf(“Winter”);
else
{
  if (m1 < Jun)
    printf(“Spring”);
  else
  {
    if (m1 < Sep)
      printf(“Summer”);
    else
      printf(“Autumn”);
  }
}
```

46. _Односвязные списки

Структура данных – программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных.
Различные СД и их реализации характеризуются:

- способом размещения элементов;
- способом представления связей между элементами;
- набором операций, определённых над СД.

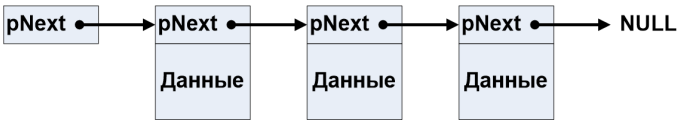
Массивы
Свойства массива:

- элементы располагаются в памяти последовательно;
- сложность обращения к элементу с индексом i: **O(1)**;
- сложность добавления/удаления: **?**
- сложность обмена элементов: **O(S)**, S — размер элемента.

Связные списки
Постановка задачи:

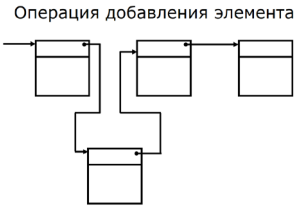
- линейная структура данных (массив);
- возможность быстро производить обмен элементов.

Односвязные списки
Представление данных:

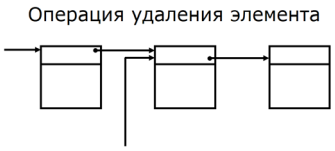


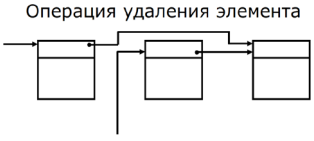
Операции:

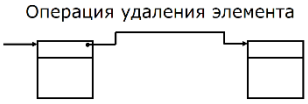
- добавление элемента:



- удаление элемента;







- перебор элементов (обход);
- поиск значения;
- поиск элемента по индексу;
- и др.

Реализация зависит от:

- выбора способа представления всего списка:
  - как указатель на первый элемент;
  - как элемент с неиспользуемым полем данных;
- способа задания места вставки новых элементов:
  - в начало списка;
  - в конец списка;
  - перед заданным элементом;
  - после заданного элемента;

Может потребоваться проверка на принадлежность элемента списку.

Динамические структуры данных

При реализации СД с помощью блоков динамической памяти важно:

- отслеживать правильность связей между элементами;
- избегать потери ссылок на ещё не освобождённые блоки.

Согласованность данных (data consistency) – свойство сложных структур данных удовлетворять условиям согласованности:

- содержать непротиворечивую информацию об элементах данных, их взаимном расположении и связях между ними.

Состояние сложной СД может быть:

- согласованным;
- несогласованным.

Операции над СД переводят СД из одного согласованного состояния в другое:

- в противном случае сопровождение кода резко усложняется.

Во время выполнения операции над СД её состояние может временно быть несогласованным. При этом досрочное завершение операции не должно оставлять СД в несогласованном состоянии.

Условия согласованности односвязного списка:

- все указатели на следующие элементы должны быть адресами элементов этого же списка, записанными явным образом;
- отсутствуют циклы:
  - если ходить по ссылкам (или против ссылок), существует единственный путь между двумя узлами.

47. _Стеки

Структура данных – программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных.
Различные СД и их реализации характеризуются:

- способом размещения элементов;
- способом представления связей между элементами;
- набором операций, определённых над СД.

Стек – структура данных с дисциплиной доступа LIFO (Last In, First Out).

Операции над стеком:

- добавление в стек;
- удаление из стека;
- проверка на пустоту стека.

Может быть реализован:

- как динамический список с ограниченным набором операций:
  - добавление только последним;
  - удаление только последнего;
- как статический массив (если известно максимальное количество элементов);
- как динамический массив.

Динамические структуры данных
При реализации СД с помощью блоков динамической памяти важно:

- отслеживать правильность связей между элементами;
- избегать потери ссылок на ещё не освобождённые блоки.

Согласованность данных (data consistency) – свойство сложных структур данных удовлетворять условиям согласованности:

- содержать непротиворечивую информацию об элементах данных, их взаимном расположении и связях между ними.

Состояние сложной СД может быть:

- согласованным;
- несогласованным.

Операции над СД переводят СД из одного согласованного состояния в другое:

- в противном случае сопровождение кода резко усложняется.

Во время выполнения операции над СД её состояние может временно быть несогласованным. При этом досрочное завершение операции не должно оставлять СД в несогласованном состоянии.

48. _Очереди

Структура данных – программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных.
Различные СД и их реализации характеризуются:

- способом размещения элементов;
- способом представления связей между элементами;
- набором операций, определённых над СД.

Очередь – структура данных с дисциплиной доступа FIFO (First In, First Out).

Операции над очередью:

- добавление в конец очереди;
- удаление из начала очереди;
- проверка на пустоту очереди.

Может быть реализована:

- как динамический список с ограниченным набором операций:
  - добавление только последним;
  - удаление только первого;
- как статический массив (если известно максимальное количество элементов).

Динамические структуры данных
При реализации СД с помощью блоков динамической памяти важно:

- отслеживать правильность связей между элементами;
- избегать потери ссылок на ещё не освобождённые блоки.

Согласованность данных (data consistency) – свойство сложных структур данных удовлетворять условиям согласованности:

- содержать непротиворечивую информацию об элементах данных, их взаимном расположении и связях между ними.

Состояние сложной СД может быть:

- согласованным;
- несогласованным.

Операции над СД переводят СД из одного согласованного состояния в другое:

- в противном случае сопровождение кода резко усложняется.

Во время выполнения операции над СД её состояние может временно быть несогласованным. При этом досрочное завершение операции не должно оставлять СД в несогласованном состоянии.

49. _Бинарные деревья

Структура данных – программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных.
Различные СД и их реализации характеризуются:

- способом размещения элементов;
- способом представления связей между элементами;
- набором операций, определённых над СД.

Графы
Граф – структура данных, состоящая из:

- объектов (вершины);
- связей между ними (рёбра).

Связь может быть:

- ненаправленной;
- направленной (ориентированный граф, рёбра орграфа — дуги).

Может задаваться:

- матрицей смежности;
- матрицей инцидентности;
- списками смежности;
- списками инцидентности.

Деревья

- Принято рассматривать как частный случай графов.
- Используется для представления иерархических отношений между элементами.

Динамические структуры данных
При реализации СД с помощью блоков динамической памяти важно:

- отслеживать правильность связей между элементами;
- избегать потери ссылок на ещё не освобождённые блоки.

Согласованность данных (data consistency) – свойство сложных структур данных удовлетворять условиям согласованности:

- содержать непротиворечивую информацию об элементах данных, их взаимном расположении и связях между ними.

Состояние сложной СД может быть:

- согласованным;
- несогласованным.

Операции над СД переводят СД из одного согласованного состояния в другое:

- в противном случае сопровождение кода резко усложняется.

Во время выполнения операции над СД её состояние может временно быть несогласованным. При этом досрочное завершение операции не должно оставлять СД в несогласованном состоянии.

50. Файлы. Общие сведения. Открытие и закрытие файлов

Файл – именованная область на каком-либо носителе, используемая для хранения данных.

Режимы доступа к файлу:

- текстовый:

31	38	35	30

- бинарный:

3A	07

Структура FILE:

- зависит от реализации;
- полагаться на формат структуры FILE нельзя;
- работа обычно осуществляется с указателем на структуру:
  `FILE *p;`

Функция fopen()
Формат:

```
FILE *fopen(
  const char *filename,
  const char *mode
);
FILE *p;
```

Возвращаемое значение:

- NULL, если при открытии произошла ошибка;
- ненулевой указатель, если файл открыт успешно.

Пример:

```
p = fopen("1.doc", "r+b");
...
```

Режим открытия файла:

Режим	Доступ	Создаёт?	Заменяет?
r	R	–	–
r+	R/W	–	–
w	W	+	+
w+	R/W	+	+
a	A	+	–
a+	R/A	+	–

R – Read, W – Write, A – Append

Режим	Описание
t	Текстовый
b	Бинарный

Функция fclose()
Формат:

```
int fclose(
  FILE *stream
);
```

Возвращаемое значение:

- 0, если файл закрыт успешно;
- EOF, если произошла ошибка.

51. Функции ввода-вывода для работы с текстовыми файлами

Стандартные потоки ввода/вывода:

- stdin;
- stdout;
- stderr.

1. Функция fputs()
Формат:

```
int fputs(
  const char *str,
  FILE *stream
);
```

Возвращаемое значение:

- успех – неотрицательное значение;
- ошибка – EOF.

2. Функция fputc()
Формат:

```
int fputc(
  int c,
  FILE *stream
);
```

Возвращаемое значение:

- успех – код символа;
- ошибка – EOF.

3. Функция fgets()
Формат:

```
char *fgets(
  char *str,
  int n,
  FILE *stream
);
```

Возвращаемое значение:

- успех – str;
- ошибка – NULL;
- конец файла – NULL.

4. Функция fgetc()
Формат:

```
int fgetc(
  FILE *stream
);
```

Возвращаемое значение:

- успех – прочитанный символ;
- ошибка – EOF;
- конец файла – EOF.

5. Функция fprintf()
Формат:
```
int fprintf(
  FILE *stream,
  const char *format,
  [, argument]...
);
```

Возвращаемое значение: количество записанных байт.

6. Функция fscanf()
Формат:
```
int fscanf(
  FILE *stream,
  const char *format,
  [, argument]...
);
```

Возвращаемое значение: количество прочитанных переменных (EOF — ошибка или конец файла).

Пример:
```
void main(void)
{
  char s[25];
  FILE *out;
  out = fopen("d:\\Ex2.txt", "wt");
  for (int i = 0; i < 3; i++)
  {
    gets(s);
    fprintf(out, "%s\n", s);
  }
  fclose(out);
  out = fopen("d:\\Ex2.txt", "rb");
  for (i = 0; i < 3; i++)
  {
    fscanf(out, "%s", s);
    printf("%s\n", s);
  }
  fclose(out);
}
```

52. Произвольный доступ к файлу

Доступ к файлу:
- последовательный;
- произвольный.

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

Параметр `origin`:
- SEEK_SET — начало файла;
- SEEK_CUR — текущее положение;
- SEEK_END — конец файла.

Возвращаемое значение:
- 0 – успешно;
- не 0 – ошибка.

Функция ftell()
Формат:
```
long ftell(
  FILE *stream
);
```

Возвращаемое значение: текущая позиция от начала файла.

Пример:
```
#include <stdio.h>
// Функция возвращает число байтов в файле
long filesize(FILE *stream)
{
  long pos, length;
  pos = ftell(stream);
  fseek(stream, 0L, SEEK_END);
  length = ftell(stream);
  fseek(stream, pos, SEEK_SET);
  return length;
}
int main()
{
  FILE *f;
  f = fopen("D:\\main.cpp","r");
  printf("%ld", filesize(f));
  fclose(f);
  return 0;
}
```

Функция feof()
Формат:
```
int feof(
  FILE *stream
);
```

Возвращаемое значение: ненулевое значение, если установлен признак конца файла для потока.

Функция ferror()
Формат:
```
int ferror(
  FILE *stream
);
```

Возвращаемое значение: ненулевое значение, если установлен признак ошибки для потока.

Признак конца файла: не хранится, возвращается стандартной библиотекой как значение EOF.

Пример:
```
char c;
while (!feof(stdin))
  c = fgetc(stdin);
```
- будет выполняться до нажатия Ctrl+Z (IBM PC), Ctrl+D (Mac)…

53. Функции ввода-вывода для работы с бинарными файлами

Функция fwrite()
Формат:
```
size_t fwrite(
  const void *buffer,
  size_t size,
  size_t count,
  FILE *stream
);
```

Возвращаемое значение: возвращает количество записанных элементов.

Функция fread()
Формат:
```
size_t fread(
  void *buffer,
  size_t size,
  size_t count,
  FILE *stream
);
```

Возвращаемое значение: количество прочитанных элементов.

Функция fflush()
Формат:
```
int fflush(
  FILE *stream
);
```

Принцип работы (C11):

- если stream указывает поток, который открыт в режиме записи или режиме обновления и в котором последней операция была не операция чтения, незаписанное содержимое потока записывается в файл;
- если stream == NULL, функция применяется к каждому открытому потоку, для которого поведение функции определено;
- в противном случае поведение не определено.

Возвращаемое значение:

- 0 – успешно;
- EOF – ошибка.

54. Директива препроцессора #include

Преобразование исходного кода в исполняемый модуль:
- препроцессор;
- компилятор;
- редактор связей (Linker).

Препроцессорная обработка (макрообработка) — преобразование текста программы путём:
- замены препроцессорных переменных их значениями;
- выполнения препроцессорных операторов.

Директивы препроцессора:
- записываются в отдельной строке;
- начинаются с символа #:
  - до символа # в строке могут быть только пробельные символы;
- синтаксис директив препроцессора не связан с остальным языком.

Алгоритм работы препроцессора:
```
while not EndOfProgram do
begin
  L := GetNextToken();
  if IsMacroVariable(L) then
    Replace(L, GetValueOf(L));
  if IsMacroDirective(L) then
    Execute(L);
end;
```

Действия препроцессора:
1. включить в компилируемый файл другие файлы;
2. определить символические константы и макросы;
3. задать режим условной компиляции и условного выполнения директив препроцессора;
4. задать номера строк;
5. выводить сообщения об ошибках;
6. использовать дополнительные команды при помощи директивы #pragma:
   - количество и назначение дополнительных команд зависит от используемого пакета Си.

Директива #include
Три формы записи:
- `#include <some_file>`
- `#include "some_file"`
- `#include preprocessor_tokens`

Пример:
```
#include <stdio.h>
#include "myprog.h"
```

55. Директивы препроцессора #define и #undef

рип

Преобразование исходного кода в исполняемый модуль:

- препроцессор;
- компилятор;
- редактор связей (Linker).

Препроцессорная обработка (макрообработка) — преобразование текста программы путём:

- замены препроцессорных переменных их значениями;
- выполнения препроцессорных операторов.

Директивы препроцессора:

- записываются в отдельной строке;
- начинаются с символа #:
  - до символа # в строке могут быть только пробельные символы;
- синтаксис директив препроцессора не связан с остальным языком.

Алгоритм работы препроцессора:

```
while not EndOfProgram do
begin
  L := GetNextToken();
  if IsMacroVariable(L) then
    Replace(L, GetValueOf(L));
  if IsMacroDirective(L) then
    Execute(L);
end;
```

Действия препроцессора:

1. включить в компилируемый файл другие файлы;
2. определить символические константы и макросы;
3. задать режим условной компиляции и условного выполнения директив препроцессора;
4. задать номера строк;
5. выводить сообщения об ошибках;
6. использовать дополнительные команды при помощи директивы #pragma:
   - количество и назначение дополнительных команд зависит от используемого пакета Си.

Директива #define
Форма записи:

```
#define имя_константы значение
```

Пример:

```
#define PI 3.14159
#define HELLO_TEXT "Hello, world!"
...
printf("%f %s", PI, HELLO_TEXT); // 3.14159 Hello, world!
```

Поддерживает разделение на строки:

```
#define NUMBER 2\
3
...
printf("%d", NUMBER); // 23
```

Заменяет константу заданной лексемой:

```
#define MAXN 12;
int array[MAXN]; ⇔ int array[12;]; // ошибка
```

```
#define MAXN 12
int array[MAXN]; ⇔ int array[12]; // компилируется
```

Преимущества const:

- контроль типов;
- значение отображается в отладчике.

Преимущество #define:

- может использоваться не только для отдельных значений.

Пример:

```
#define PI = 3.14159
...
printf(“%f”, PI); // ошибка
float f PI; // компилируется
//
const float PI = 3.14159;
...
printf(“%f”, PI); // компилируется
float f PI; // ошибка
```

Макросы с параметрами
Объявление макроса:

```
#define имя_макроса(параметры) код
```

Вызов макроса:

```
имя_макроса(параметры)
```

Пример:

```
#define CIRC(r) (3.14 * (r) * (r))
s = CIRC(4);
s = (3.14 * (4) * (4));
```

Скобки используются на случай формирования выражений с приоритетами:

```
#define CIRC(r) (3.14 * r * r)
s = CIRC(4);
s = (3.14 * 4 * 4);
```

```
#define CIRC(r) (3.14 * (r) * (r))
s = CIRC(a + b);
s = (3.14 * (a + b) * (a + b));
```

```
#define CIRC(r) (3.14 * r * r)
s = CIRC(a + b);
s = (3.14 * a + b * a + b);
```

Может использоваться для объявления структур (поскольку замена проводится на этапе лексического анализа):

```
#define STRUCT(type, name) \
  struct type \
  { \
    int value; \
    struct type *next; \
  } name;
STRUCT(MyStruct, s1);
```

Отсутствует контроль типов:

```
#define SUM(x, y) ((x) + (y))
printf("3 + 5 = %d", SUM(3, 5));            // 3 + 5 = 8
printf("3.4 + 5.7 = %0.1f", SUM(3.4, 5.7)); // 3.4 + 5.7 = 9.1
printf("'a' + '0' = '%c'", SUM('a', '0'));  // 'a' + '0' = 'q'
```

Побочные эффекты:

```
a = 2;
s = CIRC(a++);
printf("s = %d, a = %d", s, a);
s = (3.14 * (a++) * (a++));
// s = 18.84, a = 4
```

```
a = 2;
s = CIRC(++a);
printf("s = %d, a = %d", s, a);
s = (3.14 * (++a) * (++a));
// s = 37.68, a = 4
```

Преимущество макросов: нет затрат на вызов функции.
Преимущество функций: нет побочных эффектов.
Решение: использовать inline-функции.

inline-функции

- inline-функции — возможность компилятора (часть языка Си), а не препроцессора;
- директива inline может быть проигнорирована компилятором.

Пример:

```
inline float circ(float r)
{
  return 3.14 * r * r;
}
```

Директива #undef
Форма записи:

```
#undef идентификатор
```

```
#undef PI
#undef CIRC
#undef DELETE
```

56. Предотвращение повторного включения файлов с исходным кодом

Include guard

```
// Library.h
#ifndef LIBRARY_H
#define LIBRARY_H
void myprint(char *s);
#endif

// Library.c
#include “Library.h”

void myprint(char *s) {
       while (*s)
          putc(*s++);
}
```

Pragma once

Проблемой include guard является возможность коллизии имён.

`#pragma once`:

- помещается в файле для тех же целей, но компилятор отслеживает повторные включения самостоятельно;
- директива нестандартная, но поддерживается многими компиляторами.

```
// Library.h
#pragma once
void myprint(char *s);

// Library.c
#include “Library.h”

void myprint(char *s)
{
      while (*s)
            putc(*s++);
}
```

57. Директивы условной компиляции

Преобразование исходного кода в исполняемый модуль:
- препроцессор;
- компилятор;
- редактор связей (Linker).

Препроцессорная обработка (макрообработка) — преобразование текста программы путём:
- замены препроцессорных переменных их значениями;
- выполнения препроцессорных операторов.

Директивы препроцессора:
- записываются в отдельной строке;
- начинаются с символа #:
  - до символа # в строке могут быть только пробельные символы;
- синтаксис директив препроцессора не связан с остальным языком.

Алгоритм работы препроцессора:
```
while not EndOfProgram do
begin
  L := GetNextToken();
  if IsMacroVariable(L) then
    Replace(L, GetValueOf(L));
  if IsMacroDirective(L) then
    Execute(L);
end;
```

Действия препроцессора:
1. включить в компилируемый файл другие файлы;
2. определить символические константы и макросы;
3. задать режим условной компиляции и условного выполнения директив препроцессора;
4. задать номера строк;
5. выводить сообщения об ошибках;
6. использовать дополнительные команды при помощи директивы #pragma:
   - количество и назначение дополнительных команд зависит от используемого пакета Си.

Условная компиляция
Способ записи:
```
#if условие
  фрагмент_кода
#endif
```

Ограничения
В условии могут содержаться только:
- символические константы (#define);
- константные переменные (const).

Примеры:
```
const int x = 5;
#if x == 5
      printf(“x = 5”);
#endif
```

```
#define x 5
#if x == 4
      printf(“x = 5”);
#endif
```

```
#if defined DEBUG && !defined NO_X
  printf("x = %d\n", x);
#endif
```

```
#ifdef DEBUG
  printDebugInfo();
#endif
#ifndef NO_LOG
  writeToLog();
#endif

#ifndef MYLIB_H
#define MYLIB_H
...
#endif

#ifdef UNICODE
#define MessageBox MessageBoxW
#else
#define MessageBox MessageBoxA
#endif
MessageBox(...);
```

Директива eilf
Способ применения:

```
#if условие_1
  фрагмент_кода_1
#elif условие_2
  фрагмент_кода_2
#elif условие_3
  ...
#else
  фрагмент_кода_n
#endif
```

Пример:

```
#if VERSION == 1
  #define INCFILE “version1.h”
#elif VERSION == 2
  #define INCFILE “version2.h”
#else
  #define INCFILE “version_def.h”
#endif
```

58. Компоновщик: назначение, принцип работы

Преобразование исходного кода в исполняемый модуль:

- препроцессор;
- компилятор;
- компоновщик.

Компиляция – преобразование исходных кодов в объектные файлы.
Компоновка – объединение объектных файлов в исполняемый файл.
Сборка = Компиляция + Компоновка.

Пример:

```
// Program.c
#include “Library.h”

void main()
{
      myprint(“Hello!”);
}

// Library.h
void myprint(char *s);

// Library.c
#include “Library.h”

void myprint(char *s)
{
      while (*s)
            putc(*s++);
}
```

Имена файлов (в т.ч. расширения) могут быть произвольными.
Существует соглашение:

- файлы с объявлениями (интерфейсом): *.h, *.hpp;
- файлы с определениями (реализацией): *.c, *.cpp.

Файл с кодом (c, cpp) должен быть добавлен в проект.
Модуль может использовать функции из других модулей:

- для компиляции модуля достаточно прототипов этих функций:
    - прототип функции задаёт её интерфейс.

Включение в модуль реализаций других модулей, как правило, не имеет смысла и не используется.
Файл с реализацией модуля включает в себя соответствующий заголовочный (*.h) файл.
Каждый модуль компилируется по отдельности:

- компилятору на вход подаётся имя файла, с которого следует начать работу;
- производится включение всех необходимых файлов (препроцессор);
- полученный исходный код преобразуется в машинный и записывается в объектный файл.

Объектный файл
Объектный файл содержит:

- скомпилированный код:
    - адреса процедур, которых нет в данном модуле, пропускаются;
    - отсутствующие процедуры добавляются в список импортируемых;
- информацию о неразрешённых ссылках на процедуры и переменные – список импортируемых процедур;

- информацию о расположении переменных и процедур, доступных другим модулям – список экспортируемых процедур.

Принцип работы компилятора
При компиляции каждого модуля компилятор запускается заново.
- заново запускается обработка директив препроцессора;
- #ifndef ... #endif, в которые помещают содержимое заголовочных файлов, срабатывают не более 1 раза при компиляции каждого модуля, но могут срабатывать несколько раз за время сборки всего проекта – для разных модулей.

Библиотеки
Несколько объектных файлов могут быть объединены в библиотеку:
- можно считать библиотеку просто коллекцией объектных модулей, записанной в 1 файл;
- некоторые стандартные библиотеки подключаются по умолчанию средой разработки.

Принцип работы компоновщика
Компоновщик пытается для всех функций, импортируемых всеми модулями, найти соответствующие экспортируемые функции в других модулях.
Результат работы компоновщика – исполняемый модуль (файл).
По сравнению с объектным файлом:
- может иметь другой формат;
- может не содержать информации об отдельных функциях;
- экспортирует одну или несколько функций:
    - в зависимости от целевой ОС.

59. Декорирование имён

Name mangling – декорирование имён.
Приём, заключающийся в «переименовании» функции в зависимости от:

- соглашения вызова;
- набора параметров;
- и т.д.

Позволяет упростить компоновщик: можно искать функции по декорированным именам, не обращая внимания на другие свойства.

Пример:

`int _cdecl f(int x) { return 0; }`	`_f`
`int _stdcall g(int x) { return 0; }`	`_g@4`
`int _fastcall h(int x) { return 0; }`	`@h@4`
`void h(int)`	`?h@@YAXH@Z`
`voidh(int)`	`CXX$_Z1HI2DSQ6A`

Единого стандарта нет, в разных компиляторах могут использоваться разные способы декорирования имён.
Компоновщик выводит сообщение об ошибке, если:

- для функции, импортируемой модулем, не удаётся найти подходящую экспортируемую функцию в других модулях;
- для функции, импортируемой модулем, найдено более одной подходящей экспортируемой функции в других модулях.

60. Точки следования

Точка следования (sequence point) — точка программы, в которой гарантируется, что все побочные эффекты предыдущих вычислений уже проявились, а побочные эффекты последующих ещё отсутствуют.

Точки следования C/C++:

1. Между вычислением левого и правого операндов &&, || и оператора-запятой:

```
*p++ != 0 && *q++ != 0
*p++ != 0 || *q++ != 0
```

2. Между вычислением условия и второго или третьего операнда тернарного оператора:

```
a = (*p++) ? (*p++) : 0
```

3. В конце всего выражения:

```
a = b;
return a + b;
Выражения в if, switch, циклах.
```

4. Перед входом в вызываемую функцию:

```
f(i++) + g(j++) + h(k++)
// Значения i, j и k
// будут увеличены до вызова,
// переданы будут старые значения
```

5. При возврате из функции, когда возвращаемое значение скопировано в вызывающий контекст (только в C++).
6. В объявлении с инициализацией на момент завершения вычисления инициализирующего значения:

```
int a = (1 + i++);
```

7. Вызов перегруженного оператора (т.к. в C++ перегруженный оператор — это функция).

61. Неопределённое поведение: условия возникновения, последствия. Другие случаи вариативного поведения программ

земля железобетоном

Неопределённое поведение (undefined behavior) – свойство некоторых языков программирования, программных библиотек и аппаратного обеспечения в определённых ситуациях выдавать результат, зависящий от различных факторов.
Поведение программы (библиотеки, устройства) в таких случаях зависит от:

- реализации компилятора (библиотеки, устройства);
- случайных параметров:
  - состояние памяти;
  - сработавшие прерывания;
  - и др.

Виды вариативного поведения программ:

- неопределённое поведение: с точки зрения спецификации языка программа считается ошибочной;
- неуточняемое поведение: программа не считается ошибочной, спецификация ограничивает варианты поведения;
- поведение, определяемое реализацией: неуточняемое поведение, которое каждая реализация должна документировать.

Поведение, определяемое реализацией:

- размер типов данных для разных платформ.

Неуточняемое поведение:

- порядок вычисления параметров функции:

```
int f()
{
  printf("F\n");
  return 3;
}

int g()
{
  printf("G\n");
  return 4;
}
```

```
int h(int i, int j)
{
  return i + j;
}

int main()
{
  return h(f(), g());
}
```

- порядок вычисления операндов:

```
f() + g()

int x = 5;

int f()
{
      int ret = x;
      x += 2;
      return ret;
}
```

```
int g()
{
      int ret = x;
      x += 3;
      return ret;
}
```

При наличии неоднозначностей стандарты C и C++ могут:

- указать единственно допустимое поведение компилятора;
- указать несколько допустимых поведений и потребовать явно задокументировать выбранное разработчиком компилятора;
- указать несколько допустимых поведений компилятора (неуточняемое поведение);

- явно указать, что поведение не определено.

Неопределённое поведение:

- компилятор должен сделать что-либо, что может показаться наиболее эффективным/простым.

Примеры неопределённого поведения:

- использование переменной до её инициализации;
- директива #pragma в GCC до версии 1.17 (спецификация языка предоставляет полную свободу в обработке #pragma):
  - компилятор запускал emacs с игрой «Ханойские башни»;
- выполнение более чем одной записи в одну и ту же переменную между двумя точками следования;
- использование неинициализированных переменных;
- переполнение целочисленных переменных со знаком:
  - INT_MAX + 1 может быть не равно INT_MIN;
  - это позволяет оптимизировать:

```
x + 1 > x = true
x * 2 / 2 = x
for (i = 0; i <= N; ++i) { ... }
```

- сдвиг на количество бит больше, чем размер типа;
- разыменование указателей на освобождённую память;
- выходы за границы массивов;
- разыменование нулевого указателя;
- нарушение соответствия типов:
  - например, приведение int * к float * (если необходимо – использовать memcpy()).

Примеры:

```
int i = 5;
i = ++i + ++i;

// Возможные значения: 13 или 14
// или любое другое!

int x = 42, y = 98;

x ^= y ^= x ^= y;

// x - ?
// y - ?

i = i++;

bool p;
...
if (p)
      puts(“p is true”);
if (!p)
      puts(“p is false”);
```

Путешествие во времени:

```
int table[4];

bool exists_in_table(int v)
{
  for (int i = 0; i <= 4; i++)
  {
    if (table[i] == v)
      return true;
  }
  return false;
}
```

Компилятор:

- На первых 4 итерациях функция может вернуть true.
- При i = 4 функция использует неопределённое поведение. Его можно игнорировать и считать, что i никогда не равно 4.
- i никогда не равно 5, потому что сначала оно должно стать равным 4, а этого никогда не произойдёт (↑).
- Значит, все пути возвращают true.

```
int table[4];

bool exists_in_table(int v)
{
  return true;
}

int value_or_fallback(int *p)
{
  printf("*p is %d\n", *p);  // разыменование пустого указателя
  return p ? *p : 42;        // может быть заменено на return *p
}

void unwitting(bool door_is_open)
{
  if (door_is_open)
    walk_on_in();
  else
  {
    ring_bell();
    fallback = value_or_fallback(nullptr);
    wait_for_door_to_open(fallback);
  }
}

void unwitting(bool door_is_open)
{
  walk_on_in();
}
```

- Ветвь else использует неопределённое поведение.
- Значит, можно считать, что выполнение никогда не дойдёт до этой ветви.
- Значит, её можно выбросить.
- Значит, оставшаяся ветвь выполнится всегда.

- Ветвь else использует неопределённое поведение.
- Значит, её можно заменить любым кодом.

```
if (door_is_open)
  walk_on_in();
else
  walk_on_in();

void keep_checking_door()
{
  for ( ; ; )
  {
    printf(“Is the door open?\n”);
    fflush(stdout);

    char response;
    if (scanf(“%c”, &response) != 1)
      return;

    bool door_is_open = (response == ‘Y’);
    unwitting(door_is_open);
    // if (!door_is_open) abort;
      // walk_on_in();
  }
}
```

- Первоначальный код вызывал `ring_bell()` перед тем, как происходило разыменование нулевого указателя и «падение» программы.
- Полученный код «возвращается в прошлое» и «отменяет звонок».
- «Путешествие во времени» может происходить не только из-за того, что удаляются проблемные части кода!
- Компилятор имеет право наоборот сгенерировать дополнительный код, который отменит действие в прошлом.

Другие примеры:

```
// Начиная с C++11
char *p = “wikipedia”;
p[0] = ‘W’;
```

```
// Правильно
char p[] = “wikipedia”;
p[0] = ‘W’;
```

```
int arr[4] = {0, 1, 2, 3};
int *p = arr + 5;

int f()
{
  // Код без return
}
```

Если между точками следования значение переменной изменяется, любые попытки её чтения (между теми же ТС) считаются неопределённым поведением:

```
a[i] = i++;
printf(“%d %d”, ++n, power(2, n));
```

62. Функции с переменным числом параметров

Пример объявления:

```
float average(int count, ...)
{
  // Тело функции
}

void main()
{
  float avg = average(5, 7, 3, 8, 5, 12);
  printf("%0.2f", avg);
}

int sum(int count, int first, ...)
{
  int result = 0;
  int *p = &first;
  for (int i = 0; i < count; i++)
    result += *p++;
  return result;
}
```

Недостатки такого решения:

- значения должны располагаться непосредственно друг за другом:
  - для значений типа char это не сработает из-за ограничений инструкции push в Intel-совместимых процессорах;
- параметры вещественных типов вообще могут находиться не в оперативной памяти:
  - в зависимости от соглашения вызова.

Соглашения вызова

Соглашение вызова – часть двоичного интерфейса приложений, регламентирующая технические особенности:

- вызова подпрограмм;
- передачи параметров;
- возврата из подпрограммы;
- передачи результата вычислений в точку вызова;
- использования регистров подпрограммой.

pascal:

- аргументы передаются через стек слева направо;
- очистку стека выполняет вызываемая процедура;
- результат — в EAX (AX);
- использовалось в Windows до версии 3.1.

cdecl:

- аргументы передаются через стек справа налево;
- очистку стека выполняет вызывающая процедура;
- результат в EAX (AX);
- **как правило, используется для вызова функций с переменным числом аргументов**.

stdcall/winapi:

- аргументы передаются через стек справа налево;
- очистку стека выполняет вызываемая процедура;
- результат в EAX (AX);
- используется в Windows в настоящее время.

манала рот все скрины вставлять, кто захочет – вспомнит асик и распишет

cdecl
x := Func(1, 2, 3, 4);

```
...
push 4
push 3
push 2
push 1
call Func
add  sp, 8
mov  [x], ax
...
```

```
Func:
  push bp
  mov  bp, sp
  ...
  mov  cx, [bp+4]
  ...
  mov  ax, ...
  ...
  pop  bp
  ret
```

pascal
x := Func(1, 2, 3, 4);

```
...
push 1
push 2
push 3
push 4
call Func
mov  [x], ax
...
```

```
Func:
  push bp
  mov  bp, sp
  ...
  mov  cx, [bp+4]
  ...
  mov  ax, ...
  ...
  pop  bp
  ret  8
```

Библиотека stdarg.h:

```
void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);

#ifndef __STDARG_H
#define __STDARG_H

typedef void *va_list;
#define __size(x)          ((sizeof(x)+sizeof(int)-1) & \
                            ~(sizeof(int)-1))
#define va_start(ap, parmN)((void)((ap) = \
                            (va_list)((char *)(&parmN)+__size(parmN))))
#define va_arg(ap, type)   ( \
                           *(type *)( \
                           ((*(char **)&(ap))+=__size(type))-(__size(type))\
                           ) \
                           )
#define va_end(ap)           ((void)0)
#endif
```

Пример:

```
#include <stdarg.h>

int sum(int count, ...)
{
      va_list args;
      int result = 0;
      va_start(args, count);
      for (int i = 0; i < count; i++)
            result += va_arg(args, int);
      va_end(args);
      return result;
}
```