Язык программирования C

Содержание

1 Предварительные замечания

Представленный материал основан на книге [1].

Подробное описание использованных в данном документе функций стандартной библиотеки системы программирования C представлено, в частности, в [1] и [2].

2 История создания языка программирования C

Язык программирования C был создан в 1972 году сотрудником Bell Laboratories Деннисом Ритчи во время совместной работы с Кеном Томпсоном над операционной системой UNIX для машины PDP-11.

Многие важные идеи языка программирования C взяты из языка программирования BCPL (Basic Combined Programming Language — базовый комбинированный язык программирования), который был создан Мартином Ричардсом (Кембрджский университет) в 1967 году. Влияние языка программирования BCPL на язык программирования C было косвенным —- через язык программирования B, разработанный Кеном Томпсоном в 1970 году для первой системы UNIX, реализованной на PDP-7.

Происхождение языка программирования C схематично представлено на рисунке 1.

Рис. 1: Происхождение языка программирования C

Хотя язык программирования Алгол появился только на несколько лет позже языка программирования Фортран, он представляет собой гораздо более развитый язык программирования, и по этой причине он в огромной степени повлиял на разработку других языков программирования. Его авторы много внимания уделили правильности синтаксиса и модульной структуре программ. Но вероятно потому, что язык программирования Алгол казался слишком абстрактным и общим, он не получил широкого распространения. Разработка языка программирования CPL была, по словам его создателей, попыткой «сохранить контакт... с реальным компьютером». Как и Алгол, язык программирования CPL был громоздким языком с множеством свойств, которые незначительно увеличивали его «мощность», но делали трудным для изучения и реализации. Язык программирования BCPL попытался преодолеть эту проблему выделением из языка программирования CPL его основных свойств. Язык программирования B, представлял собой дальнейшее упрощение языка CPL и действительно являлся очень экономным языком программирования, подходившим для использования на имевшихся в то время вычислительных машинах. Но экономия средств языков программирования BCPL и B привела к тому, что они стали очень ограниченными, пригодными только для узкого круга задач. Достижением Ритчи при разработке языка программирования C было восстановление части потерянной общности, в основном за счет удачной системы типов данных. Он сделал это без потери простоты или «контакта с компьютером», что и было целью разработки языка программирования CPL.

3 Особенности языка программирования C

В отличие от языков программирования BCPL и B, которые являются «безтиповыми» языками программирования, в языке программирования C определены следующие основные типы данных: литеры, целые числа, числа с плавающей точкой (вещественные), указатели, массивы, структуры и объединения.

Изначально язык программирования С создавался как язык системного программирования, который мог бы заменить язык ассемблера. В связи с этим, в языке программирования С присутствуют операции, которые непосредственно связаны с командами языка ассемблера (см. таблицу 1).

Команды языка ассемблера	**Операции языка программирования С**
1. Команда увеличения операнда на единицу **INC**	Операция увеличения (инкремент) ++
2. Команда уменьшения операнда на единицу **DEC**	Операция уменьшения (декремент) --
3. Команда двоичного сдвига влево **SHL**	Операция двоичного сдвига влево <<
4. Команда двоичного сдвига вправо **SHR**	Операция двоичного сдвига вправо >>

Таблица 1: Связь операций языка программирования С с командами языка ассемблера микропроцессора Intel.

С другой стороны, язык программирования С предоставляет полный набор операторов для реализации базовых алгоритмических конструкции, на основе которых, в частности, строятся языки высокого уровня (см. таблицу 2).

Базовые алгоритмические конструкции	**Операторы языка программирования С**
1. Следование	... *оператор*$_i$; *оператор*$_{i+1}$; ...
2. Условный выбор	**if(** *условие* **)** *оператор*$_i$; **else** *оператор*$_{i+1}$;
3. Циклические конструкции	
3.1. Цикл с предусловием	**while(** *условие* **)** *оператор*;
3.2. Цикл с постусловием	**do** *оператор*; **while(** *условие* **);**
3.3. Цикл с параметром	**for(** *инициализация параметра*; *условие*; *изменение параметра* **)** *оператор*;

Таблица 2: Связь языка программирования С с языками программирования высокого уровня.

Но все же язык программирования С нельзя назвать языком программирования высокого уровня в полном смысле этого слова по следующим причинам:

1. В языке программирования С отсутствуют операции над составными объектами, такими как строки символов, множества, списки и массивы.
2. В языке программирования С нет каких-либо средств распределения памяти, помимо возможности определения статических переменных и стекового механизма при выделении места для локальных переменных функций.
3. В языке программирования С нет средств ввода-вывода и каких-либо методов доступа к файлам.

Все эти механизмы высокого уровня обеспечиваются в языке программирования С исключительно с помощью явно вызываемых функций (подробное описание функций стандартной библиотеки системы программирования С см. в [1], [2]).

В качестве результата функции могут возвращать значения основных типов данных, поддерживаемых языком программирования С. Любая функция допускает рекурсивное обращение к себе. Функции программы, составленной на языке программирования С могут храниться в отдельных исходных файлах и компилироваться независимо.

Большинство, если не все, из реализованных систем программирования С содержат в себе стандартный набор функций, выполняющих действия высокого уровня.

Таким образом, язык программирования С является относительно низкоуровневым языком, который для достижения максимальной эффективности использования ЭВМ позволяет определить каждую деталь в логике программы.

С другой стороны, это относительно высокоуровневый язык, скрывающий подробности архитектуры ЭВМ и, таким образом, повышающий эффективность программирования.

Этот феномен можно понять, рассматривая место языка программирования С среди следующих классов языков программирования:

- языки ассемблера;
- проблемно-ориентированные языки программирования;
- машинно-ориентированные языки программирования.

Языки ассемблера, которые обеспечивают довольно простой способ программирования прямо в системе команд машины, заставляют мыслить в терминах аппаратуры и описывать каждую операцию в терминах машины. Программирование на языках ассемблера утомительно и способствует появлению ошибок. Ранние языки программирования высокого уровня, такие как Фортран и Алгол, были созданы как альтернатива ассемблерным языкам. Но языки программирования Фортран и Алгол слишком абстрактны для работы на уровне системы: это проблемно-ориентированные языки, которые использовались для решения задач в технике, науке или управлении. В связи с этим были созданы машинно-ориентированные языки программирования. Такие языки превосходны для программирования на уровне, близком к машинному, но они слишком привязаны к архитектуре конкретной ЭВМ. Таким образом, язык программирования С занимает промежуточное положение между классами проблемно-ориентированных и машинно-ориентированных языков программирования.

4 Структура программы

Программа на языке программирования С состоит из одного или нескольких программных модулей, которые могут компилироваться раздельно (см. рисунок 2).

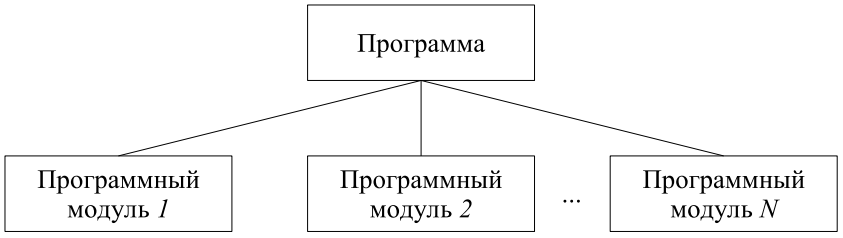


Рис. 2: Структура программы

Структура типичного **программного модуля**, составленного на языке программирования С представлена на рисунке 3.

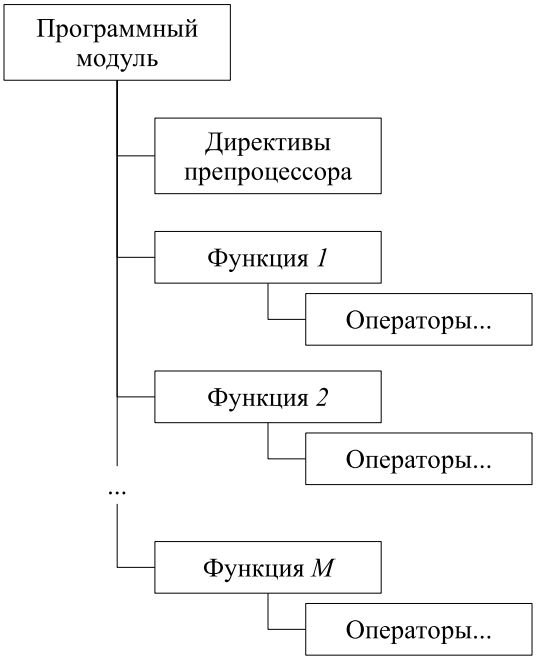


Рис. 3: Структура программного модуля

В одном из программных модулей, составляющих программу на языке С, обязательно должна присутствовать функция с именем **main()**.

Препроцессор выполняет предварительную обработку исходного текста на языке программирования С, осуществляя в нём замены в соответствии со специальными **директивами**. После работы препроцессора начинает работу собственно компилятор, который уже переводит программу в машинные коды.

В системе программирования С препроцессор входит в состав компилятора и обеспечивает выполнение макроподстановок, условной компиляции и включения файлов в исходный текст соответствующего программного модуля. В частности, наличие препроцессора позволяет:

- использовать в исходном тексте именованные константы;
- разрабатывать программы с учетом того, что их исходные тексты будут компилироваться в различных системах программирования С;
- описывать объявления часто используемых функций и переменных в отдельных заголовочных файлах.

Функции являются базовыми программными единицами в программах, составленных на языке программирования С. Язык программирования С проектировался так, чтобы функции были эффективными и простыми в использовании. Обычно программы на языке программирования С состоят из большого числа небольших функций, а не наоборот.

Но это не значит, что язык С является языком функционального программирования!

Функции состоят из **операторов**. Операторы языка программирования С подразделяются на следующие пять типов:

- операторы описания — служат для описания типов переменных и объявления функций;
- операторы присваивания — служат для изменения значений переменных;
- операторы вызова функции — передают управление соответствующей функции;
- операторы управления — служат для изменения естественного порядка выполнения программы.

Пример фрагмента программного модуля на языке программирования С, содержащего реализацию функции **main()** приведён в листинге 1.

Листинг 1: Пример кода программы на языке программирования С

```
#include <stdio.h>                  ⟵ директива препроцессора
int main()                          ⟵ имя функции
{                                   ⟵ начало тела функции
  int num;                          ⟵ оператор описания
  num = 25;                         ⟵ оператор присваивания
  printf("num = %d\n", num);        ⟵ оператор вызова функции
  return 0;                         ⟵ оператор управления
}                                   ⟵ завершение тела функции
```

На рисунке 4 отражена схема работы препроцессора для приведённого фрагмента исходного кода.

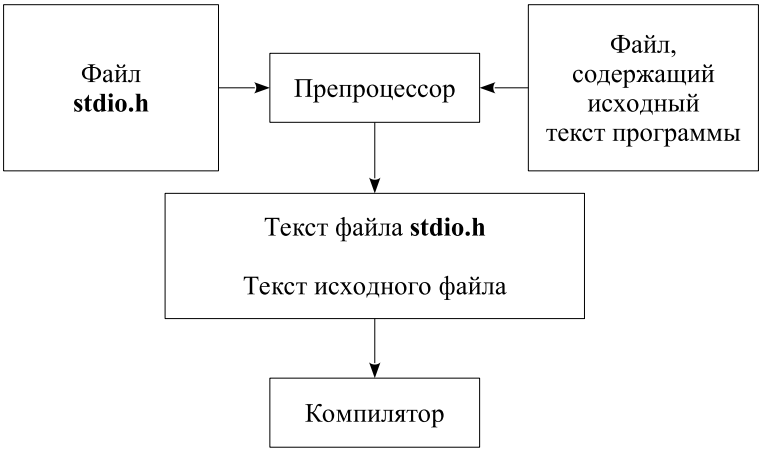
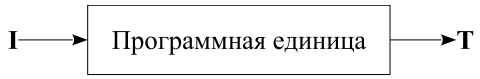


Рис. 4: Схема работы препроцессора

5 Типы данных, переменные и константы

Как правило, любая целенаправленная деятельность носит характер перевода какого-либо объекта из одного состояния (начального) в другое (конечное).

Аналогично, цель любой программной единицы заключается, в общем случае, в преобразовании информации, представленном на рисунке 5.



I – исходная (входная) информация
T – результирующая (выходная) информация

Рис. 5: Преобразование информации

В качестве информации, которая преобразуется программной единицей, в языках программирования используются **переменные**.

Имена (идентификаторы) переменных в языке программирования C составляются из букв и цифр, причем первый символ должен быть буквой. Символ подчеркивания '_' считается буквой. Прописные и строчные буквы различаются. В качестве имён переменных запрещается использовать ключевые слова языка программирования C.

Что же это за объект «переменная» по отношению к ЭВМ?

На рисунке 6 отражено противоречие между формой представления информации человеком и ЭВМ.

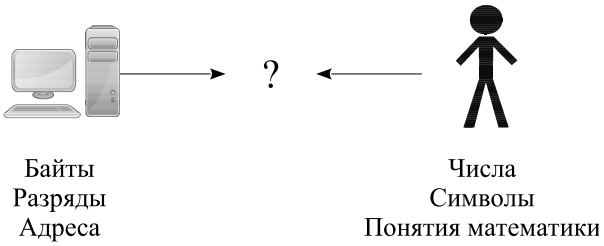


Рис. 6: Противоречие между формой представления информации

Для устранения данного противоречия необходим механизм отображения понятий, которыми «оперирует» ЭВМ в понятия, которыми оперирует прикладной программист и наоборот.

Такого рода механизм в языках программирования реализуется с помощью введения понятия **типа данных**.

Простые типы данных в языке программирования C представлены в таблице 3.

Ключевое слово	**Пояснение**
char	один байт, где можно хранить один символ из множества символов конкретной ЭВМ или однобайтовое целое число.
int	целое число, соответствующее естественному размеру целых чисел в конкретной ЭВМ (слово).
float	число с плавающей точкой одинарной точности.
double	число с плавающей точкой двойной точности.

Таблица 3: Простые типы данных

В языке программирования C для целых чисел введены следующие три модификатора:

- short;
- long;
- unsigned.

Модификаторы **short** и **long** связаны с размером целых чисел.

Модификатор **unsigned** позволяет интерпретировать содержимое соответствующих ячеек памяти как целое число без знака. Числа без знака всегда положительны.

В таблице 4 приведены количество байт, занимаемых простыми типами данных, и соответствующие им диапазоны значений для 64-х разрядной системы программирования C **gcc**.

Тип данных	Количество байт	Диапазон значений
char	1	$-128 \dots 127$
unsigned char	1	$0 \dots 255$
int	4	$-2\ 147\ 483\ 648 \dots 2\ 147\ 483\ 647$
unsigned int	4	$0 \dots 4\ 294\ 967\ 295$
short int	2	$-32\ 768 \dots 32\ 767$
unsigned short int	2	$0 \dots 65\ 535$
long int	8	$-9\ 223\ 372\ 036\ 854\ 775\ 808 \dots 9\ 223\ 372\ 036\ 854\ 775\ 807$
unsigned long int	8	$0 \dots 18\ 446\ 744\ 073\ 709\ 551\ 615$
float	4	$\pm 3.4 * 10^{-38} \dots \pm 3.4 * 10^{+38}$
double	8	$\pm 1.7 * 10^{-308} \dots \pm 1.7 * 10^{+308}$

Таблица 4: Размеры и диапазоны значений типов данных

Необходимо отметить, что в разных системах программирования C приведённые в таблице 4 данные могут отличаться.

Таким образом, переменная характеризуется двумя атрибутами, а именно:

- именем (идентификатором);
- типом данных.

Описание переменных в программах на языке программирования C обеспечивается посредством операторов описания.

Каждый оператор в языке программирования C должен оканчиваться символом '**;**'. Но не каждая синтаксическая конструкция языка программирования C, оканчивающаяся символом '**;**' является оператором. Например:

`2 + (c = printf("............"));` — выражение, а не оператор.

Общий вид операторов описания переменных:

тип_данных␣*имя_переменной [, имя_переменной]*;

Примеры описания переменных представлены в листинге 2.

Листинг 2: Примеры объявления переменных

```
int inum, jnum;
char c;
float a;
unsigned int unum;
```

Естественно предположить, что переменные должны содержать конкретные данные соответствующего типа. Явно задаваемые значения (данные) представляют из себя **константы**.

В соответствии с основными типами данных, в языке программирования C существуют **следующие виды констант**:

- Целые числа.
  - Десятичные целые числа — состоят из десятичных цифр. Примеры:

    `1234` ;

    `123456789L`, `3214567891` — целочисленные константы, относящиеся к типу **long int**;

    `321U`, `321u`, `4956768UL` — беззнаковые целочисленные константы;
  - Восьмеричные целые числа — состоят из восьмеричных цифр, перед которыми стоит цифра `0`. Примеры:

    `0377` ;

    `0377775L` ;

    `0444444UL` .
  - Шестнадцатеричные целые числа — представляются шестнадцатеричными цифрами, перед которыми записывается пара знаков `0x` или `0X`. Примеры:

    `0xFF` ;

    `0xABCDEF12L` ;

`0X1FU` .

- Числа с плавающей точкой — состоят из десятичных цифр, имеют десятичную точку или экспоненциальную часть, или же и то, и другое. Представляют собой данные типа **double**. Окончание `f` или `F` указывает на то, что данная константа должна интерпретироваться как данные типа **float**. Примеры:

  `23.23` ;

  `3e2` ($= 3 * 10^2$);

  `2.e3` ($= 2.0 * 10^3$) ;

  `2.435f` — вещественная константа, относящаяся к типу данных **float** .

- Символьные (литерные) константы. Литерная константа есть **целое значение**, записанное в виде литеры, обрамленной одиночными кавычками. Значением литерной константы является числовой код литеры из набора литер на конкретной ЭВМ. Литерные константы могут участвовать в операциях над числами точно так же, как и любые другие целые, хотя чаще они используются для сравнения с другими литерами. Некоторые литеры в литерных и строковых константах записываются с помощью эскейп-последовательности, которая представляется в виде `\L`, где `L` может быть:

  – буквой;
  – числом из одной, двух или трех восьмеричных цифр, определяющих код символа;
  – числом из одной, двух или трех шестнадцатеричных цифр с предшествующей буквой x или X, определяющих код символа.

  Примеры символьных констант:

  `'A'`, `'Д'`, `'g'`;

  `'\n'` — символ новой строки;

  `'\\'` — обратная дробная черта (обратный «слэш»);

  `'\''` — апостроф;

  `'\47'` — апостроф;

  `'\x27'` — апостроф;

  `'\"'` — двойная кавычка.

  Эскейп-последовательность интерпретируется как один символ (литера).

- Строковые константы — это нуль или более литер, заключенных в двойные кавычки. Пример:

  ```
  "The Beatles\n"
  ```

  В конец строки компилятор всегда добавляет 0-символ `'\0'`, что является признаком конца строки. Каждая пара соседних строк трактуется как одна строка. Например:

  ```
  "All You Need " "Is" " Love"
  ↓
  "All You Need Is Love"
  ```

  Фактически, строковая константа представляет собой тип данных «массив символов».

При описании переменной, данная переменная может быть **инициализирована** (см. листинг 3)

Листинг 3: Примеры инициализации переменных

```
int num = 25;
char c = 'A';
```

6 Основные операции

6.1 Арифметические операции и выражения

Бинарными арифметическими операциями являются `+` (сложение), `-` (вычитание), `*` (умножение), `/` (деление), `%` (деление по модулю — получение значения остатка от целочисленного деления, применяется только для целых чисел).

Унарными арифметическими операциями являются операции изменения знака `+` и `-`.

Арифметические выражения формируются из совокупности операций и соответствующих им операндов. Пример:

```
5 + '\20' + 2e3 * 2
```

Бинарные операции `+` и `-` имеют одинаковый приоритет, который ниже приоритета операций `*`, `/`, `%`, приоритет которых, в свою очередь, ниже приоритета унарных операций `+` и `-`.

6.2 Операции инкремента и декремента

Операции инкремента и декремента являются унарными операциями.

Операция инкремента `++` увеличивает соответствующий операнд на `1`.

Операция декремента `--` уменьшает соответствующий операнд на `1`.

Операции инкремента и декремента можно использовать как префиксные операции (операция помещается перед операндом, например, `++n`), и как постфиксные (операция помещается после операнда, `n++`). Префиксная операция изменяет значение операнда до того, как его значение будет использовано, а постфиксная операция изменяет значение операнда после того, как его значение использовано (см. листинг 4).

Листинг 4: Примеры постфиксной и префиксной операции инкремента

```
...
int a, b;
...
b = 1;
a = b++;    // a принимает значение 1, b принимает значение 2
a = ++b;    // a принимает значение 3, b принимает значение 3
...
```

В контексте, где требуется только увеличить или уменьшить значение переменной, безразлично, какую выбрать операцию — префиксную или постфиксную (см. листинг 5).

Листинг 5: Примеры независимого использования постфиксной и префиксной операции инкремента

```
...
int a, b;
...
b++;
a = b;
++a;
b = a;
...
```

6.3 Побитовые операции

В языке программирования C имеются шесть операций для манипулирования с битами. Их можно применять только к целочисленным операндам, т.е. к операндам типов **char**, **short**, **int** и **long**, знаковым и беззнаковым (**signed**, **unsigned**):

`&` — побитовое **И** (побитовое умножение);

`|` — побитовое **ИЛИ** (побитовое сложение);

`^` — побитовое **исключающее ИЛИ**;

`<<` — побитовый **сдвиг влево**;

`>>` — побитовый **сдвиг вправо**;

`~` — побитовое **отрицание**.

Операция `&` (побитовое **И**) часто используется для обнуления (сброса) некоторой группы разрядов. Пример:

```
n = n & 0x000F;
```

Операция `|` (побитовое **ИЛИ**) применяется для установки определенных разрядов в `1`. Пример:

```
x = n | 0x0101;
```

Операция `^` (побитовое **исключающее ИЛИ**) в каждом разряде устанавливает `1`, если соответствующие разряды операндов имеют различные значения, и `0` в противном случае. Пример:

```
y = x ^ 0x1010;
```

Операция `<<` выполняет **сдвиг влево** своего левого операнда на число битовых позиций, задаваемое правым операндом, которое должно быть положительным. Пример:

```
y = y << 2;
```

Операция `>>` выполняет **сдвиг вправо** своего левого операнда на число битовых позиций, задаваемое правым операндом, которое должно быть положительным. Сдвиг вправо беззнаковой величины всегда сопровождается заполнением освобождающихся разрядов нулями. Сдвиг вправо знаковой величины в одних системах программирования C происходит с «размножением» знака («арифметический сдвиг»), на других — с заполнением освобождающихся разрядов нулями («логический сдвиг»). Пример:

```
y = y >> 2;
```

Унарная операция `~` (побитовое **отрицание**) превращает единичные биты в нулевые и наоборот (дополнение целого до единиц по всем разрядам). Пример:

```
x = ~y;
```

Примеры использования побитовых операций также представлены листингом 6.

Листинг 6: Примеры использования побитовых операций

```
#include <stdio.h>

char * prtbits(char c){
  static char result[9];
  unsigned uc;
  int i;

  uc = 0x80;

  for(i = 0; i<8; i++, uc = uc>>1){
    if(c & uc){
      result[i] = '1';
    }
    else{
      result[i] = '0';
    }
  }

  return result;
}

int main(){
  unsigned char bits;
  char sbits;

  bits = 0xFF;
  printf("bits = 0xFF\nbits = %s\n\n", prtbits(bits));

  bits = bits & 0x0F;
  printf("bits = bits & 0x0F\nbits = %s\n\n", prtbits(bits));
```

```
    bits = bits | 0xF0;
    printf("bits = bits | 0xF0\nbits = %s\n\n", prtbits(bits));

    bits = bits ^ 0x0F;
    printf("bits = bits ^ 0x0F\nbits = %s\n\n", prtbits(bits));

    bits = bits << 2;
    printf("bits = bits << 2\nbits = %s\n\n", prtbits(bits));

    sbits = bits;
    printf("sbits = bits\nsbits = %s\n\n", prtbits(sbits));

    bits = bits >> 2;
    printf("bits = bits >> 2\nbits = %s\n\n", prtbits(bits));

    sbits = sbits >> 2;
    printf("sbits = sbits >> 2\nsbits = %s\n\n", prtbits(sbits));

    bits = ~bits;
    printf("bits = ~bits\nbits = %s\n", prtbits(bits));

    return 0;
}
```

6.4 Операции присваивания

Основной операцией, с помощью которой можно изменить значение переменной, является **операция присваивания** `=` (см. листинг 7).

Листинг 7: Пример операции простого присваивания

```
...
int a, b;
...
a = (b = 13);
...
```

В рамках фрагмента, приведённого в листинге 7 выполняется следующая последовательность действий (см. также рисунок 7):

1. выполняется операция присваивания `(b = 13)`;
2. переменная `b` получает значение `13`;
3. значением операции `(b = 13)` в целом является значение `13`;
4. таким образом, весь оператор присваивания может рассматриваться, как `a = 13`;
5. в результате, переменная `a` получает значение `13`.

Рис. 7: Операция и оператор присваивания

Общий вид операции простого присваивания:

имя_переменной = *выражение*

Большинству бинарных операций соответствуют операции присваивания вида *op*=, где *op* является одной из операций: +, -, *, /, %, <<, >>, &, ^, | .

Общий вид операции присваивания такого рода:

имя_переменной op= *выражение*

Это эквивалентно записи следующего вида:

имя_переменной = *имя_переменной op* (*выражение*)

Разница заключается в том, что в первом случае значение переменной вычисляется один раз.

Пример использования такого рода операции присваивания приведён в листинге 8.

Листинг 8: Пример операции присваивания

```
a *= y + 2; // эквивалентно оператору: a = a * (y + 2);
```

При использовании таких операций присваивания компилятор генерирует, как правило, более эффективный код, например, выражение

операнд1 += *операнд2*

преобразуется в команду языка ассемблера

ADD␣*операнд1*␣*операнд2*

6.5 Операции отношения и логические операции

Операциями **отношения** являются > (больше), >= (больше или равно), < (меньше), <= (меньше или равно).

Операциями **сравнения на равенство** являются == (равно), != (не равно).

Приоритет операций отношения выше приоритета операций сравнения на равенство.

Операции отношения имеют более низкий приоритет, чем арифметические операции. Например, выражение

```
i < n - 1
```

равносильно выражению

```
i < (n - 1)
```

Логическими операциями являются:

- бинарные операции && (логическое умножение) и || (логическое сложение);
- унарная операция ! (отрицание).

Приоритет операции && выше, чем приоритет операции ||, однако их приоритет ниже, чем приоритет операций отношения и сравнения на равенство.

По определению результат вычисления выражения отношения или логического выражения представляется числом 0 (**ложь**) или 1 (**истина**). В общем случае **любое ненулевое значение является истиной**.

Унарная операция ! преобразует ненулевой операнд в 0, а нуль — в 1.

6.6 Операция «запятая»

Пара выражений, разделенных запятой, вычисляется слева направо, значение левого выражения теряется. Типом и значением результата являются тип и значение правого операнда. Например, значением выражения

```
t = 3, t + 2
```

является значение 5 .

При описании формальных параметров функции и описании переменных запятая не рассматривается как операция, а рассматривается как разделитель.

Основная область использования операции «запятая» — цикл с параметром.

6.7 Условные выражения

Условное выражение в общем виде представляется следующим образом:

выражение1 `?` *выражение2* `:` *выражение3*

Первым вычисляется *выражение1*. Если его значение не нуль (истина), то вычисляется *выражение2*, и значение этого выражения становится значением всего условного выражения. В противном случае вычисляется *выражение3* и его значение становится значением условного выражения. Пример приведён в листинге 9

Листинг 9: Пример использования условного выражения

```
y *= z + ((a >= 0) ? a : -a);
```

Условное выражение можно использовать в любом месте, где допускается выражение. Если *выражение2* и *выражение3* принадлежат разным типам, то тип результата определяется правилами преобразования.

6.8 Преобразование типов и операция приведения типа

Если операнды операции принадлежат разным типам, то они приводятся к некоторому общему типу. Приведение выполняется в соответствии с небольшим числом правил. Обычно автоматически производятся лишь те преобразования, которые без какой-либо потери информации превращают операнды с меньшим диапазоном значений в операнды с большим диапазоном значений.

Например, для арифметических операций имеет место следующий набор правил преобразования:

- Если какой-либо из операндов принадлежит типу **double**, то другой приводится к типу **double**.
- Иначе, если какой-либо из операндов принадлежит типу **float**, то другой приводится к типу **float**.
- Иначе операнды типов **char** и **short** приводятся к типу **int**.
- Если один из операндов типа **long**, то другой приводится к типу **long**.

В данном случае преобразования выполняются так, чтобы не происходило потери точности.

При присваивании значение правой части присваивания приводится к типу левой части, который и является типом результата.

В данном случае возможна потеря точности, вплоть до получения непредсказуемого результата.

Для любого выражения можно явно указать преобразование его типа, используя **унарную операцию приведения типа**:

(*имя_типа*) *выражение*

Выражение как бы присваивается некоторой переменной указанного типа и эта переменная используется вместо всей конструкции.

6.9 Приоритет и порядок выполнения операций

В таблице 5 представлен приоритет и порядок выполнения операций, включая операции, которые будут рассмотрены позднее. Операции, перечисленные в одной строке таблицы, имеют одинаковый приоритет. Строки упорядочены по убыванию приоритета.

Операции	**Порядок выполнения**
`() [] -> .`	слева направо
`! ~ ++ -- + - * &` (*тип*) `sizeof`	справа налево
`* / %`	слева направо
`+ -`	слева направо
`<< >>`	слева направо
`< <= > >=`	слева направо
`== !=`	слева направо
`&`	слева направо
`^`	слева направо
`|`	слева направо
`&&`	слева направо
`||`	слева направо

?:	справа налево
= += -= *= /= %= &= ^= |= <<= >>=	справа налево
,	слева направо

Таблица 5: Приоритет и порядок выполнения операций

7 Конструкции управления

Порядок, в котором выполняются вычисления, определяется операторами управления.

7.1 Блоки операторов

Фигурные скобки { и } используются для объединения описаний и операторов в **составной оператор**, или **блок**, чтобы с точки зрения синтаксиса эта новая конструкция воспринималась как один оператор. После правой закрывающей фигурной скобки в конце блока точка с запятой не ставится (см. листинг 10).

Листинг 10: Пример блока операторов

```
...
{
  int j, k;
  j = 123;
  k = j * 23;
}
...
```

7.2 Конструкция if-else

Конструкция **if-else** используется для принятия решения. Синтаксис конструкции **if-else** представлен листингом 11.

Листинг 11: Синтаксис конструкции **if-else**

```
if (выражение)
    оператор1
[ else
    оператор2 ]
```

Отсутствие в одной из вложенных друг в друга **if**-конструкций **else**-части может привести к неоднозначному толкованию записи. Эта неоднозначность разрешается тем, что **else** связывается с ближайшим **if**, у которого нет своего **else** (см. листинг 12).

Листинг 12: Пример вложенной конструкции **if-else**

```
if (n>0)
    if (a<b)
        z = a;
    else
        z = b;
```

При необходимости иной интерпретации требуется должным образом использовать блок операторов (см. листинг 13).

Листинг 13: Пример вложенной конструкции **if-else** с использованием блока операторов

```
if (n>0){
  if (a<b)
      z = a;
}
else
  z = b;
```

7.3 Конструкция else-if

Конструкция **else-if** представляет собой наиболее общий способ описания многоступенчатого принятия решения. Синтаксис конструкции **else-if** представлен листингом 14.

Листинг 14: Синтаксис конструкции **else-if**

```
if (выражение1)
    оператор1
else if (выражение2)
    оператор2
else if (выражение3)
    оператор3
    ...
[ else
    операторN ]
```

Выражения вычисляются по порядку; как только встречается выражение со значением **истина**, выполняется соответствующий ему оператор; на этом последовательность проверок завершается.

Последняя **else**-часть выполняется, если все предыдущие условия не выполняются. Листингом 15 представлен пример использования конструкции **else-if** при определении типа символа (цифра — 1, пробел — 2, другой символ — 3).

Листинг 15: Определение типа символа с использованием конструкции **else-if**

```
if((c == '0') || (c == '1') || (c == '2') || (c == '3') || (c == '4') ||
   (c == '5') || (c == '6') || (c == '7') || (c == '8') || (c == '9') ){
   ntype = 1;
}
else if(c == ' '){
   ntype = 2;
}
else{
   ntype = 3;
}
```

7.4 Конструкция switch (переключатель)

Конструкция **switch** используется для выбора одного из многих путей выполнения программы в зависимости от значения выражения, результат которого должен быть целым или символом. Синтаксис конструкции **switch** представлен листингом 16.

Листинг 16: Синтаксис конструкции **switch**

```
switch (выражение){
  case константа1 : [ операторы1 ]
  case константа2 : [ операторы2 ]
  ...
  case константаN : [ операторыN ]
  [ default : операторы ]
}
```

Конструкция **switch** проверяет, совпадает ли значение выражения с одним из значений, входящих в некоторое множество целых констант, и выполняет соответствующие этому значению операторы. Каждая ветвь **case** помечена одной или несколькими целочисленными константами или константными выражениями. Вычисления начинаются с той ветви **case**, в которой константа совпадает со значением выражения. Константы всех ветвей **case** должны отличаться друг от друга. Если ни одна из констант не подходит, то выполняется ветвь, помеченная зарезервированным словом **default**, если таковая имеется, в противном случае ничего не делается. Ветви **case** и **default** можно располагать в любом порядке. Листингом 17 представлен пример использования конструкции **switch** при определении типа символа (цифра — 1, пробел — 2, другой символ — 3).

Листинг 17: Определение типа символа с использованием конструкции **switch**

```
switch(c){
  case '0': case '1': case '2': case '3': case '4': case '5': case '6':
```

```
    case '7':
    case '8':
    case '9': ntype = 1;
              break;
    case '_': ntype = 2;
              break;
    default : ntype = 3;
              break;
  }
```

Оператор **break** вызывает немедленный выход из конструкции **switch**. Поскольку выбор ветви **case** реализуется как переход на метку, то после выполнения одной ветви **case**, если ничего не предпринять, программа будет выполнять следующую ветвь **case**. Инструкции **break** и **return** являются наиболее распространенными средствами выхода из конструкции **switch**.

7.5 Конструкции цикла

7.5.1 Цикл с предусловием (while)

Синтаксис конструкции цикла с предусловием **while** представлен листингом 18.

Листинг 18: Синтаксис конструкции цикла с предусловием **while**

```
while (выражение)
    оператор
```

Сначала вычисляется *выражение*. Если его значение **истина** (не равно 0), то выполняется *оператор*, и вычисление *выражения* повторяется. Этот цикл повторяется до тех пор, пока значение *выражение* не станет **ложь** (равно 0), после чего управление передается на следующий за конструкцией цикла оператор. Листингом 19 представлен пример использования цикла с предусловием **while**.

Листинг 19: Пример использования конструкции цикла с предусловием **while**

```
i = 2;
s = 1.0;
while(i<= 5) {
   s *= a;
   i++;
}
```

7.5.2 Цикл с постусловием (do-while)

Синтаксис конструкции цикла с постусловием **do-while** представлен листингом 20

Листинг 20: Синтаксис конструкции цикла с постусловием **do-while**

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

Тело цикла с постусловием **do-while** выполняется хотя бы один раз. Сначала выполняется *оператор*, затем вычисляется *выражение*. Если значение *выражения* **истина**, то *оператор* выполняется снова. Когда значение *выражения* становится **ложь**, цикл заканчивает работу. Листингом 21 представлен пример использования цикла с постусловием **do-while**.

Листинг 21: Пример использования конструкции цикла с постусловием **do-while**

```
s = 1.0;
i = 1;
do{
   s *= a;
   i++;
}while(i<5);
```

7.5.3 Цикл с параметром (for)

Синтаксис конструкции цикла с параметром **for** представлен листингом 22.

Листинг 22: Синтаксис конструкции цикла с параметром **for**

```
for([выражение1]; [выражение2]; [выражение3])
    оператор
```

Конструкция, представленная листингом 22, может быть смоделирована с помощью конструкции, представленной листингом 23.

Листинг 23: Моделирование конструкции цикла с параметром **for** посредством конструкции цикла с предусловием **while**

```
выражение1;
while(выражение2){
  оператор;
  выражение3;
}
```

С точки зрения грамматики языка программирования С *выражение1*, *выражение2*, *выражение3* представляют сбой произвольные выражения, но чаще *выражение1* и *выражение3* — это присваивания или вызовы функций, а *выражение2* — выражение отношения. Любое из трех выражений может отсутствовать, но точку с запятой опускать нельзя. При отсутствии *выражения1* или *выражения3* считается, что их просто нет в конструкции цикла; при отсутствии *выражения2* полагается, что его значение как бы всегда **истина**. Листингом 24 представлен пример использования цикла с параметром **for**.

Листинг 24: Пример использования конструкции цикла с постусловием **for**

```
for(i = 1, s = 1.0; i<=5; i++){
  s *= a;
}
```

7.6 Операторы break и continue

Оператор **break** позволяет немедленно выйти из цикла или переключателя. Этот оператор вызывает немедленный выход из самого внутреннего из объемлющих его циклов или переключателей. Листингом 25 представлен пример использования оператора **break**.

Листинг 25: Пример использования оператора **break**

```
i = 1;
s = 1.0;
for(;;){
  if(i>5){
    break;
  }
  s *= a;
  i++;
}
```

Оператор **continue** вынуждает объемлющий его цикл (**for**, **while**, **do-while**) начать следующий шаг итерации. Для циклов **while** и **do-while** это означает немедленный переход к проверке условия, а для цикла **for** — к приращению шага. Оператор **continue** можно применять только к циклам, но не к конструкции **switch**. Будучи помещенным внутрь конструкции **switch**, расположенной в цикле, этот оператор вызовет переход к следующей итерации этого цикла. Листингом 26 представлен пример использования оператора **break**.

Листинг 26: Пример использования оператора **continue**

```
for(i=0; i<n; i++){
  if (a[i]<0 ){
    continue;
  }
  ...
}
```

7.7 Оператор перехода goto и метки

Синтаксис оператора **goto**:

`goto` *метка*`;`

Оператор **goto** осуществляет передачу управления на оператор помеченный *меткой*. *Метка* должна быть расположена в текущей функции. Нельзя передавать управление во внутренний блок. Наиболее типичная ситуация для применения оператора перехода — необходимость прервать обработку в некоторой глубоко вложенной структуре и выйти сразу из двух или большего числа вложенных циклов.

Метка может стоять перед любым оператором. Листингом 27 представлен пример использования **goto**.

Листинг 27: Пример использования оператора **goto**

```
for(...){
   for(...){
      if(...)
           goto met;
   }
}
...
met: ...
```

8 Функции

Как известно, в практической деятельности решение любой сложной задачи требует, чтобы данная задача была разбита на ряд подзадач меньшей размерности и, затем, каждая из подзадач решается автономно.

Аналогично, при создании сложных программных систем рассматриваемая система разбивается на ряд подсистем, которые разрабатываются автономно.

В языке программирования С элементарной программной подсистемой является функция. Функции подразделяют большие вычислительные задачи на более мелкие и позволяют воспользоваться тем, что уже сделано другими разработчиками. В выбранных должным образом функциях скрыты несущественные для других частей программы детали их функционирования, что делает программу в целом более ясной и облегчает внесение в нее изменений.

Обычно программы на языке С состоят из большого числа небольших функций, а не наоборот. Программу можно располагать в одном или нескольких исходных файлах. Эти файлы можно компилировать отдельно, а загружать вместе, в том числе и с ранее откомпилированными библиотечными функциями.

Функция — самостоятельная единица программы, спроектированная для реализации конкретной задачи. В языке программирования С разрешается рекурсивный вызов функций.

Связи по данным между функциями осуществляются через аппарат формальных параметров (аргументы), возвращаемые значения и внешние (глобальные) переменные.

Для определения функции в языке С необходимо (см. рисунок 8):

1. Задать **описание** функции ("черный ящик").
2. Задать **реализацию** функции.

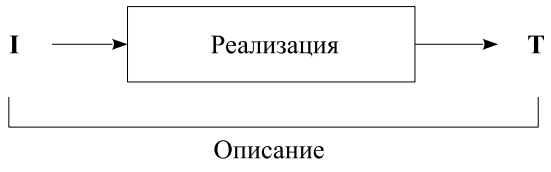


Рис. 8: Описание и реализация функции

Описание любой функции в языке программирования С имеет следующий вид:

`[`*тип_результата*`]` *имя_функции*`([`*список_параметров*`]);`
где
список_параметров `::` *описание_параметра1*, *описание_параметра2*, ...
описание_параметра `::` *тип_параметра имя_параметра* .

При описании функции *тип_результата* может быть опущен. По умолчанию *тип_результата* является **int**.

Описание параметров похоже на описание переменных.

Листинг 28 представляет примеры описания функций.

Листинг 28: Пример описания функций

```
long int fact(int n);
long int sqr(int a, int n);
double fsqr(float a, int n);
```

Определение **реализации** функции в общем случае имеет вид, представленный листингом 29.

Листинг 29: Определение реализации функции

```
[тип_результата] имя_функции ([список_параметров]) {
  тело_функции
}
```

Тело_функции представляет собой совокупность операторов языка программирования C.

Возврат из функции и **передача результата** в вызывающую программную единицу осуществляется оператором **return**.

Общий вид оператора **return**:

```
return [выражение];
```

Тип возвращаемого значения должен совпадать с типом результата функции.

Листинг 30 представляет реализацию функции возведения вещественного значения в целую степень.

Листинг 30: Функция возведения в степень

```
double fsqr(float a, int n){
  double ret;
  int s;
  ret = 1.0;
  if(n != 0 ){
    for(s = (n>0)?n:(-n); s>0; s--)
        ret *= a;
    if(n<0)
        ret = 1.0/ret;
  }
  return ret;
}
```

В исходном файле определения реализаций функций разрешается располагать в любом порядке.

Кроме того, исходная программа может быть разбита на любое число файлов, так чтобы в различных файлах содержались реализации различных функций (см. листинги 31 и 32).

Листинг 31: Файл 1

```
/* Реализация функции возведения в степень */
double fsqr(float a, int n){ // a, n - формальные параметры функции fsqr()
  ...
}
```

Листинг 32: Файл 2

```
/* Описание функции возведения в степень */
double fsqr(float b, int m);
...
/* Программная единица, в которой вызывается функция fsqr() */
main(){
  int i, s;
  double b;
  ...
  b = fsqr(i, s); // i, s - фактические параметры функции fsqr()
  ...
}
```

8.1 Передача параметров

В языке программирования С **параметры всегда передаются по значению**. Область оперативной памяти, используемая для передачи параметров имеет организацию типа **стек** (см. листинг 33 и соответствующую схему на рисунке 9).

Листинг 33: Передача параметров при заданном в описании функции списке параметров

```
double fsqr(float p, int k);
main(){
  short int i;
  int s;
  double b;
  ...
  b = fsqr(i, s); // происходит автоматическое преобразование типа short int
                  // фактического параметра i к типу float
  ...
}
```

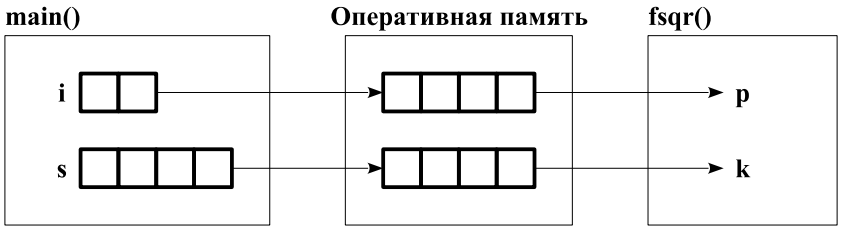


Рис. 9: Передача параметров при заданном в описании функции списке параметров

В частном случае, при описании любой функции может отсутствовать список её формальных параметров, что может привести к непредсказуемым последствиям (см. листинг 34 и соответствующую схему на рисунке 10).

Листинг 34: Передача параметров при отсутствии в описании функции списка параметров

```
double fsqr();
main(){
  short int i;
  int s;
  double b;
  ...
  b = fsqr (i, s); // не происходит автоматическое преобразование типа short int
                   // фактического параметра i к типу float
  ...
}
```

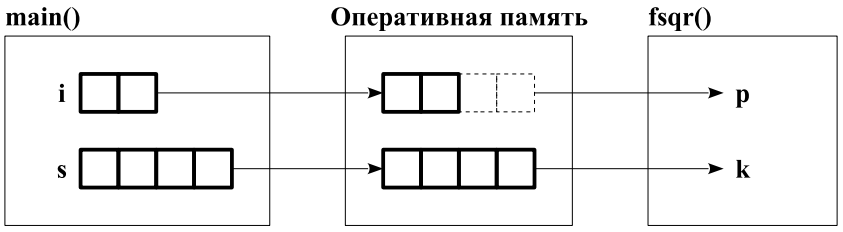


Рис. 10: Передача параметров при отсутствии в описании функции списка параметров

9 Указатели и адреса

Указатель — это переменная, содержащая адрес памяти, по которому расположена, например, какая-либо другая переменная.

Описание переменной-указателя:

*тип * имя_переменной*

Пример объявления переменной-указателя:

```
int * pi;
```

9.1 Операция получения адреса &

Унарная операция **&** возвращает в качестве результата адрес соответствующей переменной (в качестве примера см. листинг 35 и соответствующую схему на рисунке 11)).

Листинг 35: Пример использования операции получения адреса переменной

```
...
int * pi;
int c;
pi = &c; // переменной p присваивается значение адреса области памяти,
         // занимаемой переменной c
...
```

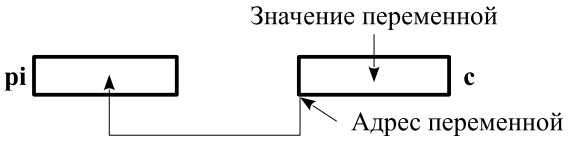


Рис. 11: Пример использования операции получения адреса переменной

9.2 Операция раскрытия указателя *

Унарная операция ***** представляет собой операцию раскрытия указателя. Применённая к переменной-указателю эта операция возвращает значение, хранящееся в области памяти, адрес которой хранится в качестве значения этой переменной-указателя. Листинг 36 и рисунок 12 представляют пример использования операции раскрытия указателя.

Листинг 36: Пример использования операции раскрытия указателя

```
...
int x = 1, y = 2;
int * ip;
...
ip = &x; // адрес переменной x присваивается переменной ip
y = *ip; // значение, хранящееся в области памяти, адрес которой
         // содержится в переменной ip, присваивается переменной y;
         // переменная y получает значение 1
*ip = 0; // области памяти, адрес которой содержится в переменной ip,
         // присваивается значение 0;
         // переменная x получает значение 0
...
```

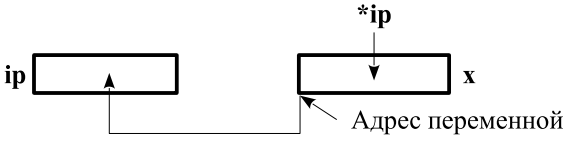


Рис. 12: Пример использования операции раскрытия указателя

9.3 Изменение значений переменных вызывающей программной единицы

Для изменения значений переменных вызывающей программной единицы, в качестве **фактических параметров** должны передаваться **адреса** соответствующих переменных, а в качестве **формальных параметров** функции должны использоваться **параметры-указатели**.

Рассмотрим сначала пример реализации функции обмена значениями двух переменных, в котором не соблюдается это правило и параметры передаются как обычные переменные (см. листинг 37. На рисунке 13) представлена схема передачи параметров и изменения их значений (цифры на схеме обозначают номера строк с соответствующими операторами из листинга 37).

Листинг 37: Пример функции обемена значениями, формальные параметры которой не являются параметрами-указателями

```
1   /* Функция обмена значениями */
2   int swap(int a, int b) {
3     int tmp;
4     tmp = a;
5     a = b;
6     b = tmp;
7     return 0;
8   }
9
10  main() {
11    int x, y;
12    x = 5;
13    y = 10;
14    swap( x, y);
15    printf("x = %d; y = %d\n", x, y); // Результат: x = 5; y = 10
16    return 0;
17  }
```

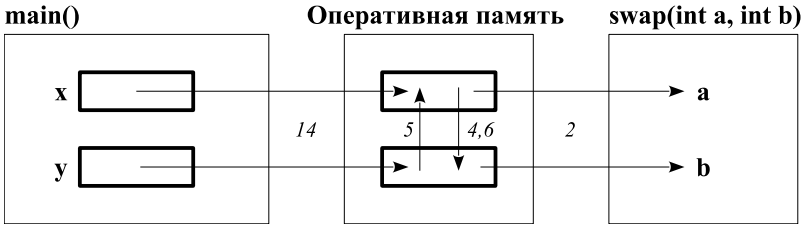


Рис. 13: Схема передачи параметров для листинга 37

Теперь рассмотрим пример реализации функции обмена значениями двух переменных, в котором фактические параметры передаются как адреса соответствующих переменных, а в качестве формальных параметров функции используются параметры-указатели (см. листинг 38. На рисунке 14) представлена схема передачи параметров и изменения их значений (цифры на схеме обозначают номера строк с соответствующими операторами из листинга 38).

Листинг 38: Пример функции обемена значениями, формальные параметры которой являются параметрами-указателями

```
1   /* Функция обмена значениями */
2   int swap(int * a, int * b) {
3     int tmp;
4     tmp = *a;
5     *a = *b;
6     *b = tmp;
7     return 0;
8   }
9
10  main() {
11    int x, y;
12    x = 5;
13    y = 10;
14    swap( &x, &y);
15    printf("x = %d; y = %d\n", x, y); // Результат: x = 10; y = 5
16    return 0;
17  }
```

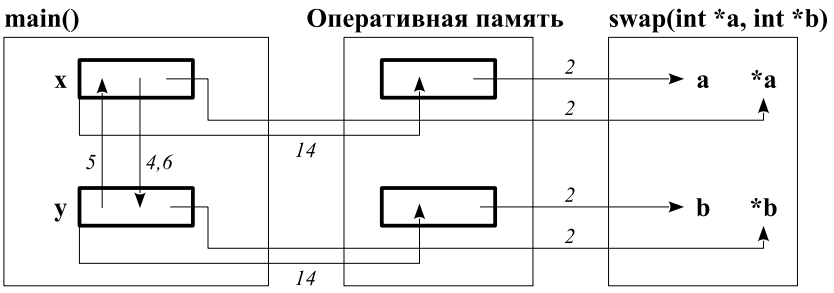


Рис. 14: Схема передачи параметров для листинга 38

10 Классы памяти

Классом памяти переменной называется **совокупность свойств**, определяющая множество функций, которые могут иметь доступ к переменной, и время существования переменной.

В языке программирования С выделяются следующие классы памяти: **автоматическая**, **регистровая**, **статическая**, **внешняя**, **внешняя статическая**:

- **автоматическая** — описывается ключевым словом **auto**. Все переменные по умолчанию являются автоматическими. Они могут быть описаны только внутри некоторого блока.
  Свойства:
  - переменные появляются только в момент входа в блок и их начальное значение не определено;
  - переменные исчезают в момент выхода из блока и присвоенное им значение теряется;
  - эти переменные доступны только внутри блока, где они описаны (другие блоки, в частности охватывающие, могут содержать переменные с теми же именами, но это будут другие переменные).
- **регистровая** — описывается ключевым словом **register** . Этот класс эквивалентен классу **auto** за тем исключением, что компилятор будет пытаться разместить переменную в регистре процессора.
- **статическая** — описывается ключевым словом **static**. Переменные этого класса могут быть описаны как внутри так и вне блока.
  Свойства:
  - переменные появляются в момент запуска программы, и исчезают в момент ее завершения, начальное значение переменной устанавливается в 0;
  - если переменная описана внутри блока, то она известна только внутри этого блока;
  - если в каком-либо исходном файле переменная описана вне блока, то она известна во всех функциях, представленных в этом файле и следующих за описанием переменной, но на неё не могут ссылаться функции из других файлов.
- **внешняя** — может быть описана только вне блока и перед ней **отсутствует** ключевое слово **static**.
  Свойства:
  - переменные появляются в момент запуска программы, и исчезают в момент ее завершения. Начальное значение переменной устанавливается в 0 .
  - переменная известна во всех функциях, идущих после описания переменной, при этом на нее могут ссылаться функции из других файлов (единиц компиляции). Для этого в соответствующем файле данная переменная может быть описана с ключевым словом **extern** .
- **внешняя статическая** — описывается ключевым словом **static**. Переменные этого класса могут быть описаны в исходном файле только вне блока и перед ними присутствует ключевое слово **static**.
  Свойства:
  - переменные появляются в момент запуска программы, и исчезают в момент ее завершения. Начальное значение переменной устанавливается в 0;
  - переменная известна во всех функциях, представленных в исходном файле, где она объявлена, и идущих после описания переменной. При этом на неё не могут ссылаться функции из других файлов (единиц компиляции).

Для **функций** доступны только два класса памяти — **внешний статический** и **внешний**. Если функция описана с ключевым словом **static**, то она может вызываться только из файла где она описана, в противном случае она доступна из других файлов.

Описание переменной с указанием ее класса памяти имеет вид:

класс_памяти тип_данных имя_переменной

Листинг 39 демонстрирует пример описания переменных с явным указанием класса памяти.

Листинг 39: Пример явного указания класса памяти для переменных

```
auto int a;
static float b;
register j;
extern int * ptr;
```

11 Массивы

11.1 Одномерные массивы

Массив — это набор переменных, имеющих одно и то же базовое имя и отличающихся одна от другой числовым признаком. Такие переменные называются элементами массива, а числовой признак — индексом элемента.

Описание массива имеет вид:

тип_элементов имя_массива[*количество_элементов*]

Листинг 40 показывает пример объявления переменных, которые являются массивами.

Листинг 40: Объявления массивов

```
float b[10];
int sup[56];
char buffer[81];
```

Все элементы массива пронумерованы начиная с 0. Таким образом индексы элементов пробегают значения от 0 до N-1, где N — количество элементов в массиве. Для обращения к некоторому элементу массива используется следующая запись:

имя_массива[*индекс_элемента*]

где *индекс_элемента* — арифметическое выражение, результатом которого является целое число; значение выражения используется как номер элемента в массиве.

Листинг 41 представляет пример обращения к элементу массива по его индексу.

Листинг 41: Обращение к элементам массива

```
int a[4]; // объявление массива a, состоящего из четырёх элементов
...
... a[0] ... // обращение к первому элементу массива
... a[4-1] ... // обращение к последнему элементу массива
```

Как и обычные переменные, элементы массива могут находится как слева от оператора присваивания так и свободно использоваться в арифметических выражениях (см. листинг 42).

Листинг 42: Использование элементов массивов

```
float tmp, b[3];
...
b[0] = b[1] + b[0] * b[1] / 2;
tmp = b[0]>[b1] ? b[0] : b[1];
```

11.2 Указатели и массивы

На этапе сборки программы каждой переменной назначается адрес оперативной памяти, по которому будет хранится значение этой переменной (см. рисунок 15).

Рис. 15: Адрес памяти переменной

Элементы массива расположены в последовательных ячейках памяти, начиная с некоторого базового адреса (см. рисунок 16).

Рис. 16: Адреса элементов массива

Для обозначения базового адреса массива используется имя массива без следующего за ним индекса (см. листинг 43). Имя массива без следующего за ним индекса является адресной **константой**, а **не переменной**.

Листинг 43: Базовый адрес массива

```
int b[100];
b; // базовый адрес массива b
```

Как указывалось ранее (см. раздел 9.1 на странице 23), для получения адреса переменной используется унарная операция &. За операцией & должно следовать имя переменной, или, например, элемента массива, и результатом выполнения этой операции является физический адрес этой переменной (элемента массива) (см. листинг 44).

Листинг 44: Получение адреса элемента массива

```
int a, b[10];
double val;
...
&a;      // адрес переменной a
&b[2];   // адрес 3-го элемента массива b
&b[0];   // адрес первого элемента массива b (его базовый адрес)
```

Также, как указывалось ранее (см. раздел 9 на странице 22), для хранения адресов памяти в языке программирования C существуют специальные типы данных — указатели.

Таким образом, указателем называется тип данных, значениями которого являются адреса памяти.

11.3 Операции над указателями

Увеличение указателя. К указателю можно прибавить (вычесть) целое число. Результат такого выражения имеет тип исходного указателя. Обычно такая операция используется для указателей на элементы массива. Если исходный указатель ссылается на i-й элемент массива, то после прибавления к нему целого n, результат будет ссылаться на элемент с индексом $(i + n)$ (см. листинг 45).

Листинг 45: Увеличение указателя

```
int b[50];
...
b;              // адрес первого элемента
b + 1;          // адрес второго элемента
b + 49;         // адрес последнего элемента
/* следующие записи эквивалентны приведенным выше */
```

```
&b[0];
&b[1];
&b[49];
...
```

Разность указателей. Если указатели ссылаются на элементы одного массива, то в результате вычитания из одного указателя другого, будет получено количество элементов массива, расположенных между этими указателями (расстояние между ними). Результатом является целое число (может быть и отрицательным). Например, см. листинг 46.

Листинг 46: Разность указателей

```
int b[100], *tmp1, *tmp2;
...
tmp1 = b;
tmp2 = &b[25];
tmp2 - tmp1; // разность равна 25
...
```

Сравнение указателей. Указатели можно сравнивать на равенство (==). Указатели равны тогда и только тогда, когда они ссылаются на одну и ту же ячейку памяти (см. листинг 47).

Листинг 47: Сравнение указателей

```
...
int b[100];
...
b == &b[0]; // истина
...
```

Получение значения (разыменование, раскрытие указателя) .

Унарная операция * используется для получения значения, которое находится по указанному адресу (см. также раздел 9.2 на странице 23). За операцией * должно следовать выражение с типом указателя на данные требуемого типа (см. листинг 48).

Листинг 48: Раскрытие указателя

```
...
int a, b[100], *tmp;
...
tmp = &a;
a = 100;
*tmp;       // результат равен 100
*(b + 99); // значение последнего элемента массива b (b[99])
...
```

11.4 Многомерные массивы

Многомерным массивом в языке программирования C является массив элементы которого суть массивы.

Описание многомерного массива имеет вид:

тип_данных имя_массива[*размер_1*][*размер_2*]...

Каждое целое число *размер_i* соответствует *i*-му измерению массива, а количество измерений массива называется его размерностью.

размер_1 можно рассматривать как количество элементов массива, а остальные размеры — как описание размерности массива-элемента (см. листинг 49).

Листинг 49: Описание многомерных массивов

```
int a[3][4]; // Матрица 3x4;
             // Массив из 3-х векторов, где каждый вектор имеет длину 4
```

```
int b[2][3][4]; // Массив из 2-х матриц 3x4;
                // Массив из двух элементов, каждый из которых является
                // массивом из трех векторов, где каждый вектор имеет длину 4
```

Для обращения к элементу многомерного массива используется запись вида:

имя_массива`[`*индекс_1*`][`*индекс_2*`]`...

Количество индексов при обращении к элементу массива должно совпадать с количеством измерений массива, которое было указано при его описании.

11.5 Многомерные массивы и указатели

Многомерный массив занимает в памяти непрерывную область, и располагается так, что самый правый индекс меняется быстрее всех остальных (для матрицы это соответствует размещению по строкам).

Например, размещение двумерного массива `int b[2][3]` схематично показано на рисунке 17.

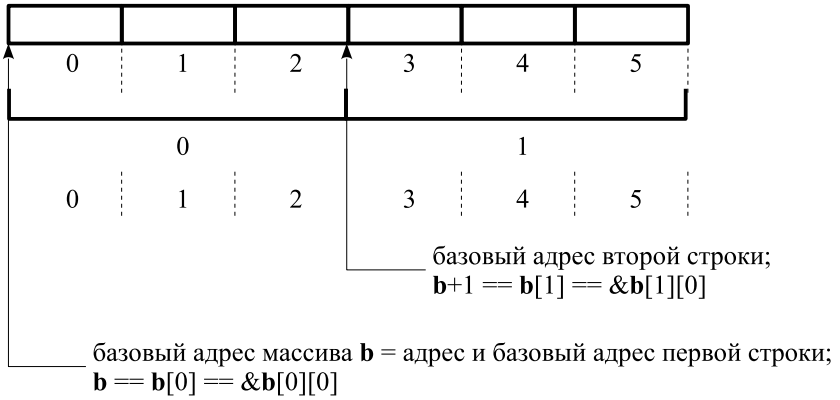


Рис. 17: Размещение двумерного массива

Можно использовать различные интерпретации такого массива (см. листинги 50, 51, 52).

Листинг 50: Матрица 2x3

```
for(i = 0; i<2; i++) {
  for(j = 0; j<3; j++) {
    printf("%d ", b[i][j]);
  }
  printf("\n");
}
```

Листинг 51: Одномерный массив из 6-ти элементов

```
for(i = 0; i<6; i++) {
  printf("%d ", *((int *)b + i));
}
```

Листинг 52: Два одномерных массива, каждый по 3 элемента

```
/* печать первой строки */
for(i = 0; i<3; i++) {
  printf("%d ", *(b[0] + i));
}
printf("\n");
/* печать второй строки */
for(i = 0; i<3; i++) {
  printf("%d ", *(b[1] + i));
}
```

11.6 Передача массивов в качестве параметров функций

В языке программирования C в функцию передаётся не сам массив, а его базовый адрес. Таким образом соответствующий формальный параметр должен быть описан в следующем виде:

тип_данных * *имя_параметра*

где *тип_данных* совпадает с типом элементов массива.

Альтернативным способом записи является следующий:

тип_данных имя_параметра[*размер*]

Операция [] является альтернативным способом записи операции *.

В силу того, что эти два описания эквивалентны, то размер массива не играет никакой роли и может быть опущен.

Для передачи многомерных массивов из описания формального параметра должен быть ясен размер массива-элемента, то есть пустые квадратные скобки могут быть указаны только для первого измерения (см. листинг 53).

Листинг 53: Передача массивов в качестве параметров функций

```
int fun(int a[2][3]) {
  ...
}

// то же самое, что и
int fun(int a[][3]) {
  ...
}
```

11.7 Инициализация массивов

В языке программирования C разрешается инициализировать массивы в момент их объявления (см. листинги 54 и 55).

Листинг 54: Инициализация одномерных массивов

```
int jarr[5] = {10, 20, 30, 40, 50};
int iarr[] = {5, 10, 15, 20};
char carr[] = {'B', 'E', 'A', 'T', 'L', 'E', 'S'};
char sarr[5] = {'H', 'E', 'L', 'P', '!'};
```

Листинг 55: Инициализация многомерных массивов

```
int iarr[2][5] = {{10, 20, 30, 40, 50},
                  {0,   5, 10, 15, 20}};
int jarr[][4] = {{5, 10, 15, 20},
                 {1,  2,  3,  4}};
```

12 Строки символов

Символьная строка в языке программирования C — это **массив символов**, причем **последний символ** строки, но **не обязательно последний элемент** соответствующего массива, должен быть 0-символом '\0'.

Листинг 56 показывает различные возможности описания и начальной инициализации строк символов и массивов строк символов.

Листинг 56: Описание и инициализация строк символов и массивов строк символов

```
/* Описание строки символов (массива символов) */
char str01[81];

/* Описание массива строк символов */
char var01[2][10];
```

```
char *ptrs01[2];

/* Инициализация массивов строк символов */
char str02[] = {'H', 'e', 'l', 'p', '!', '\0'};

char str03[81] = "Help!";

char var02[2][10] = {{'H','a','y','г','a','д','\0'},
                     {'P','e','п','e','й','\0'}};

char var03[][10] = {"Наугад", "Репей"};

/* Инициализация указателей */
char * ptrs02[2] = {"Наугад", "Репей"};

char * ptrs03[] = {"Наугад", "Репей", "Оптимальный вариант"};
```

Пояснения к листингу 56:

- под строку символов `str01` в памяти выделяется массив из 81-ти элемента типа **char** (элемент типа **char** занимает 1 байт памяти);
- под массив строк символов `var01` в памяти выделяется двумерный массив элементов типа **char** размерностью 2x10, то есть, будет выделено 20 байт памяти;
- массив `ptrs01` представляет собой массив из двух элементов типа **char***, то есть, каждому из элементов может быть присвоен адрес памяти, в которой содержаться символы (тип данных **char**);
- под массив `str02` выделяется 6 байт памяти, заполненных символами `'H'`, `'e'`, `'l'`, `'p'`, `'!'`, `'\0'`;
- под массив `str03` выделяется 81 байт памяти, причём шесть первых байт будут заполнены символами, представленными заданной символьной константой (шестой символ является 0-символом, см. определение строковых констант в разделе 5, стр. 10);
- под массив строк символов `var02` в памяти выделяется двумерный массив элементов типа **char** размерностью 2x10, то есть, будет выделено 20 байт памяти, при этом, первые 7 из 10 байт первой строки будут заполнены символами `'H'`, `'a'`, `'y'`, `'г'`, `'a'`, `'д'`, `'\0'`, а первые 6 из 10 байт второй строки — символами `'P'`, `'e'`, `'п'`, `'e'`, `'й'`, `'\0'`;
- распределение и инициализация памяти под массив `var03` полностью аналогичны массиву `var02`;
- массив `ptrs02` представляет собой массив из двух элементов типа **char***, при этом, первый элемент этого массива будет содержать **адрес области памяти**, в которой будет храниться строковая константа `"Наугад"`, а второй элемент будет содержать **адрес области памяти**, в которой будет храниться строковая константа `"Репей"`;
- массив `ptrs03` представляет собой массив из трёх (определяется количеством строковых констант инициализации) элементов типа **char***, при этом, первый элемент этого массива будет содержать **адрес области памяти**, в которой будет храниться строковая константа `"Наугад"`, второй элемент будет содержать **адрес области памяти**, в которой будет храниться строковая константа `"Репей"`, а третий элемент будет содержать **адрес области памяти**, в которой будет храниться строковая константа `"Оптимальный вариант"`.

На рисунке 18 приведена схема размещения в памяти массивов `var02` и `var03`, представленных в листинге 56 (цифрами на схеме представлены индексы элементов массива).

	0	1	2	3	4	5	6	7	8	9
0	H	a	y	г	a	д	\0			
1	P	e	п	e	й	\0				

Рис. 18: Размещение массива строк символов, представленного в виде двумерного массива символов (char)

На рисунке 19 приведена схема размещения в памяти массива `ptrs02`, представленного в листинге 56 (цифрами на схеме представлены индексы элементов массива).

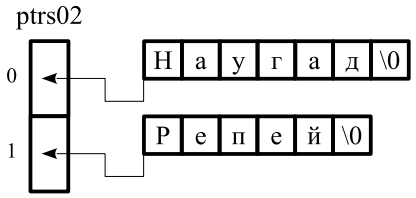


Рис. 19: Размещение массива строк символов, представленного в виде одномерного массива указателей (char*)

12.1 Основные функции работы со строками

Описания функций работы со строками содержаться в файле **string.h** .

В в исходном файле, в котором будут обращения к стандартным функциям обработки строк символов, должна присутствовать следующая директива препроцессора:

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

Подробное описание функций стандартной библиотеки системы программирования C см. в [1], [2].

12.1.1 Определение длины строки.

Описание функции определения длины строки (str*ing* len*gth* → strlen):

int strlen(**char *** str);

В качестве результата функция `strlen` возвращает количество символов, содержащихся в строке `str` (см. листинг 57). Фактически, параметр `str` представляет собой адрес области памяти, начиная с которого необходимо выполнять подсчёт символов. Подсчёт символов проводится начиная с адреса памяти, заданного параметром `str`, до тех пор, пока не встретится 0-символ.

Листинг 57: Пример обращения к функции strlen()

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

int main(){
  char s[25] = "All You need is LOVE!";
  char * ptrs;

  ptrs = "The Beatles";

  printf("%d\n", strlen(s));      // результат: 21
  printf("%d\n", strlen(ptrs));   // результат: 11
  printf("%d\n", strlen(s+16));   // результат: 5

  return 0;
}
```

12.1.2 Копирование строк символов (присваивание).

Описание функции копирования строк символов (str*ing* c*o*py → strcpy):

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

Функция выполняет копирование символов (байт), начиная с адреса памяти, заданного параметром `src`, в область памяти, адрес которой задаётся параметром `dest`. Копирование символов проводится начиная с адреса, заданного параметром `src` до тех пор, пока не встретится 0-символ, который также копируется, после чего выполнение функции завершается. В качестве результата функция возвращает значение параметра `dest` (см. листинг 58).

Листинг 58: Пример обращения к функции strcpy()

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

```
int main(){
  char str[17];
  char * ptrs;

  strcpy(str, "Hello, Goodbye!");
  printf("%s\n", str);

  ptrs = "y, You!";
  strcpy(str+2, ptrs);
  printf("%s\n", str);

  return 0;
}
```

Содержимое строки символов `str` в результате выполнения программы, представленной листингом 58, схематично отражено на рисунке 20

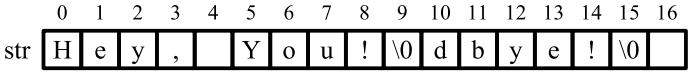


Рис. 20: Результат выполнения функций strcpy()

12.1.3 Копирование определенного количества символов.

Описание функции копирования определённого количества символов (str*ing* n c*o*py → strncpy):

char * strncpy(**char *** dest, **char *** src, **int** len);

Функция выполняется аналогично функции `strcpy()`, за исключением того, что копирование символов проводится не до достижения 0-символа, а ограничивается количеством символов, заданным значением параметра `len`. В качестве результата функция возвращает значение параметра `dest` (см. листинг 59).

Листинг 59: Пример обращения к функции strncpy()

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

int main(){
  char str[10];

  strcpy(str, "Something");
  printf("%s\n", str);

  strncpy(str, "Hello, Goodbye!", 5);
  printf("%s\n", str);

  str[5] = '\0';
  printf("%s\n", str);

  *(strncpy(str, "Hello, Goodbye!", 5) + 5) = '\0'; // другой вариант
  printf("%s\n", str);

  return 0;
}
```

Содержимое строки символов `str` в результате выполнения программы, представленной листингом 59, схематично отражено на рисунке 21

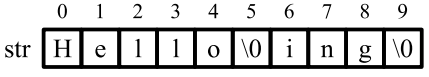


Рис. 21: Результат выполнения функции strncpy()

12.1.4 Сравнение двух строк символов.

Описание функции сравнения двух строк символов (str*ing* c*ompare*):

- **int** strcmp(**char *** str1, **char *** str2);

Функция выполняет лексикографическое сравнение двух строк символов, адреса которых заданы параметрами `str1` и `str2` соответственно. В результате выполнения сравнения функция возвращает следующие целочисленные значение (см. листинг 60):

- отрицательное целочисленное значение — строка символов `str1` лексикографически меньше строки `str2`;
- значение `0` — строка символов `str1` совпадает по длине и содержанию со строкой `str2` (строка `str1` лексикографически равна строке `str2`);
- положительное целочисленное значение — строка `str1` лексикографически больше строки `str2`.

Листинг 60: Пример обращения к функции strcmp()

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

int main(){
  char * str1, * str2;
  int res;

  str1 = "ABCDE";
  str2 = "ABCDE";
  if((res = strcmp(str1+2, str2))<0){
    printf("%s < %s\n", str1+2, str2);
  }
  else if(res>0){
    printf("%s > %s\n", str1+2, str2);
  }
  else{
    printf("%s == %s\n", str1+2, str2);
  }

  return 0;
}
```

12.1.5 Сравнение определённого количества символов двух строк

Описание функции сравнения определённого количества символов двух строк (str*ing* n c*ompare* → strncmp):

int strncmp(**char *** str1, **char *** str2, **int** len);

Функция выполняет лексикографическое сравнение двух строк символов, адреса которых заданы параметрами `str1` и `str2` соответственно, при этом сравнивается только количество символов, заданное значением параметра `len`. В результате выполнения сравнения функция возвращает следующие целочисленные значение (см. листинг 61):

- отрицательное целочисленное значение — `len` символов строки `str1` лексикографически меньше `len` символов строки `str2`;
- значение `0` — `len` символов строки `str1` совпадают содержанию с `len` символами строки `str2` (`len` символов строки `str1` лексикографически равны `len` символам строки `str2`);
- положительное целочисленное значение — `len` символов строки `str1` лексикографически больше `len` символов строки `str2`.

Листинг 61: Пример обращения к функции strncmp()

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

int main(){
  char * str1, * str2;
  int res;

  str1 = "ABCDE";
  str2 = "ABCDE";
  if((res = strncmp(str1+2, str2+2, 2))<0){
    printf("%c%c < %c%c\n", str1[2], str1[3], str2[2], str2[3]);
  }
  else if(res>0){
    printf("%c%c > %c%c\n", str1[2], str1[3], str2[2], str2[3]);
  }
  else{
    printf("%c%c == %c%c\n", str1[2], str1[3], str2[2], str2[3]);
  }

  return 0;
}
```

12.1.6 Сцепление двух строк символов

Описание функции сцепления строк символов (str*ing* cat*enate* → strcat):

char * strcat(**char *** dest, **char *** src);

Функция выполняет сцепление строки символов, адрес начала которой задан параметром `dest`, со строкой символов, адрес начала которой задан параметром `src`. В качестве результата функция возвращает значение параметра `dest` (см. листинг 62 и рисунок 22, на котором схематично представлено изменение состояния строки `str`, при этом, числа в скобках слева обозначают номера соответствующих строк листинга 62).

Листинг 62: Пример обращения к функции strcat()

```
1  #include<stdio.h>
2  #include<string.h>
3
4  int main(){
5    static char str[13];
6
7    strcpy(str, "Led");
8    printf("%s\n", str);
9
10   strcat(str, " Zeppelin");
11   printf("%s\n", str);
12
13   return 0;
14 }
```

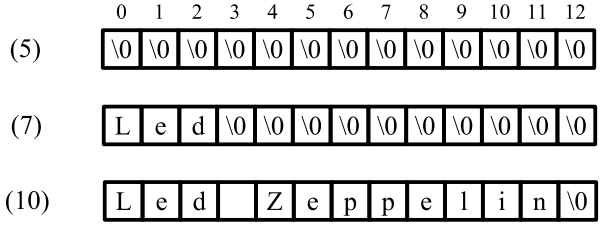


Рис. 22: Изменение состояния строки str

12.1.7 Сцепление определённого количества символов

Описание функции сцепления строки символов с определённым количеством символов из другой строки (str*ing* n cat*enate* → strncat):

char * strncat(**char *** dest, **char *** src, **int** len);

Функция выполняется аналогично функции `strcat()`, за исключением того, что из строки `src` берётся количество символов для сцепления, заданное значением параметра `len`. В качестве результата функция возвращает значение параметра `dest` (см. листинг 63).

Листинг 63: Пример обращения к функции strncat()

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

int main(){
  char str[17];
  char * temp = "A Day In The Life";

  strcpy(str, "Oh, Happy");
  printf("%s\n", str);

  strncat(str, temp+1, 4);
  printf("%s\n", str);

  return 0;
}
```

12.1.8 Заполнение определённого количества байт определенным значением.

Описание функции заполнения определённого количества байт определённым значением (mem*ory* set → memset):

void * memset(**void *** s, **int** c, **int** len);

Тип данных **void *** представляет собой **нетипизированный** указатель (адрес) некоторой области памяти. Как правило, увеличение значения такого указателя на единицу обеспечивает переход к следующему байту памяти. Функция выполняет заполнение памяти, начиная с адреса, заданного значением параметра `s`, символом, заданным значением параметра `c`. Заполняется количество байт, соответствующее значению параметра `len`. В качестве результата функция возвращает значение параметра `s` (см. листинг 64).

Листинг 64: Пример обращения к функции memset()

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

int main(){
  char str[17] = "Led Zeppelin";

  printf("%s\n", str);

  memset((void*) str+3, '!', 9);
  printf("%s\n", str);

  return 0;
}
```

12.1.9 Элементарные функции ввода-вывода строк

Функция ввода строки с экрана. Описание функции ввода строки с экрана:

char * gets(**char *** str);

Функция обеспечивает интерактивный ввод строки символа с экрана (с клавиатуры). Обращение к функции заставляет программу останавливать своё выполнение и ожидать ввода строки. Ввод строки завершается нажатием клавиши `Enter` на клавиатуре. Фактически, нажатие этой клавиши обозначает ввод символа перехода на новую строку (символ `'\n'`). Функция сохраняет считанную строку в памяти, начиная с адреса, заданного значением параметра `str` и при этом заменяет символ `'\n'` на символ `'\0'`, после чего завершает свою работу и в качестве результата возвращает значение параметра `str` (см. листинг 65).

Листинг 65: Пример обращения к функции gets()

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

int main(){
  char str[80];

  gets(str);
  printf("%s\n", str);

  return 0;
}
```

Важное замечание. В современных системах программирования C функция `gets()` не рекомендуется к использованию и является устаревшей. Это связано с тем, что при использовании этой функции нет никакого контроля на количество вводимых символов, которое может превысить размер массива символов, выделенного под вводимую строку символов.

Функция вывода строки. Описание функции вывода строки символов на экран:

int puts(**char** * str);

Функция обеспечивает вывод на экран строки символов, адрес начала которой задаётся значением параметра `str`, при этом, при выводе строки на экран 0-символ в выводимой строке заменяется на символ `'\n'`. В качестве результата функция возвращает целое неотрицательное число в случае отсутствия ошибки вывода, иначе, функция возвращает значение `-1` (см. листинг 66).

Листинг 66: Пример обращения к функции puts()

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

int main(){
  char * str = "Stairway to heaven";

  puts(str);

  return 0;
}
```

13 Структуры

В языке программирования C существует способ построения сложных типов данных путем объединения уже существующих типов в структуры. Описание структуры имеет вид:

struct *имя_типа_структуры*{

тип_данных1 имя_переменной11 [, имя_переменной12, ...];

тип_данных2 имя_переменной21 [, имя_переменной22, ...];

...

};

Переменная, фигурирующая в описании типа структуры называется **полем структуры** (компонентой структуры).

Такое описание вводит новый тип данных именуемый:

struct *имя_типа_структуры*

Таким образом, впоследствии можно будет описывать переменные с введенным типом следующим образом:

struct *имя_типа_структуры имя_переменной1 [, имя_переменной2, ...]*;

Описание структуры может быть объединено с объявлением переменной следующим образом:

struct *имя_типа_структуры*{

...

} *имя_переменной1 [, имя_переменной2, ...]*;

Если структура используется для описания только нескольких переменных в одном месте программы, то можно опустить *имя_типа_структуры*:

struct{

...

} *имя_переменной1 [, имя_переменной2, ...]*;

При этом описание переменных этого же типа в другом месте программы будет невозможно.

Переменную типа структуры можно рассматривать как объединение полей структуры, под одним именем. Значение переменной типа структуры распадается на значения входящих в нее полей. Для доступа к полю переменной используется следующая запись:

имя_переменной_типа_структуры.имя_поля

Листинг 67 представляет пример использования переменных-структур.

Листинг 67: Пример использования переменных-структур

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

struct album{
  char group[33];
  char name[128];
  int year;
  int min, sec;
};

int main(){
  struct album d1, d2;

  strcpy(d1.group, "The Beatles");
  strcpy(d1.name, "Sgt. Pepper's Lonely Hearts Club Band");
  d1.year = 1967;
  d1.min = 40;
  d1.sec = 37;

  strcpy(d2.group, "Led Zeppelin");
  strcpy(d2.name, "Presence");
  d2.year = 1976;
  d2.min = d1.min+4;
  d2.sec = 3;

  printf("Group   : %s\n", d1.group);
  printf("Album   : %s\n", d1.name);
  printf("Year    : %d\n", d1.year);
  printf("Duration: %02d:%02d\n\n", d1.min, d1.sec);
```

```
    printf("Group   : %s\n", d2.group);
    printf("Album   : %s\n", d2.name);
    printf("Year    : %d\n", d2.year);
    printf("Duration: %02d:%02d\n", d2.min, d2.sec);

    return 0;
};
```

Примечание: Иногда структуры называют **неоднородными массивами** (в отличие от обычных массивов, которые являются однородными).

13.1 Присваивание структур

Переменные типа структур можно присваивать друг другу если они относятся к одному и тому же типу, при этом все поля одной структуры будут скопированы в поля другой структуры (см. листинг 68).

Листинг 68: Пример присваивания структур

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

struct album{
  char group[33];
  char name[128];
  int year;
  int min, sec;
};

int main(){
  struct album d1, d2, d3;

  strcpy(d1.group, "The Beatles");
  strcpy(d1.name, "Sgt. Pepper's Lonely Hearts Club Band");
  d1.year = 1967;
  d1.min = 40;
  d1.sec = 37;

  strcpy(d2.group, "Led Zeppelin");
  strcpy(d2.name, "Presence");
  d2.year = 1976;
  d2.min = d1.min+4;
  d2.sec = 3;

  strcpy(d3.group, "Pink Floyd");
  strcpy(d3.name, "The Dark Side Of The Moon");
  d3.year = 1973;
  d3.min = 43;
  d3.sec = 49;

  printf("Struct d1:\n");
  printf("Group   : %s\n", d1.group);
  printf("Album   : %s\n", d1.name);
  printf("Year    : %d\n", d1.year);
  printf("Duration: %02d:%02d\n\n", d1.min, d1.sec);

  printf("Struct d2:\n");
  printf("Group   : %s\n", d2.group);
  printf("Album   : %s\n", d2.name);
  printf("Year    : %d\n", d2.year);
  printf("Duration: %02d:%02d\n\n", d2.min, d2.sec);

  printf("Struct d3:\n");
```

```
    printf("Group   : %s\n", d3.group);
    printf("Album   : %s\n", d3.name);
    printf("Year    : %d\n", d3.year);
    printf("Duration: %02d:%02d\n\n\n", d2.min, d2.sec);

    d2 = d3;

    printf("After d2 = d3:\n\n");

    printf("Struct d1:\n");
    printf("Group   : %s\n", d1.group);
    printf("Album   : %s\n", d1.name);
    printf("Year    : %d\n", d1.year);
    printf("Duration: %02d:%02d\n\n", d1.min, d1.sec);

    printf("Struct d2:\n");
    printf("Group   : %s\n", d2.group);
    printf("Album   : %s\n", d2.name);
    printf("Year    : %d\n", d2.year);
    printf("Duration: %02d:%02d\n\n", d2.min, d2.sec);

    printf("Struct d3:\n");
    printf("Group   : %s\n", d3.group);
    printf("Album   : %s\n", d3.name);
    printf("Year    : %d\n", d3.year);
    printf("Duration: %02d:%02d\n\n", d2.min, d2.sec);

    return 0;
}
```

13.2 Указатели на структуры

Структура располагается в памяти в последовательных ячейках. Размер памяти, занимаемой структурой, является суммой размеров входящих в нее полей, возможно с учётом выравнивания некоторых на границу слова (см. листинг 69 и соответствующую схему размещения полей структуры в памяти, представленную на рисунке 23).

Листинг 69: Размещение структур в памяти

```
struct {
    int i;
    float x;
    char s[3];
} st;
```

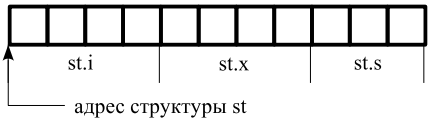


Рис. 23: Размещение полей структуры в памяти

Описание указателя на структуру имеет вид:

struct *имя_типа_структуры* ***** *имя_переменной*;

Если `p` — указатель на структуру, то для получения значения поля структуры, адрес которой хранится в переменной `p`, можно использовать запись следующего вида:

`p ->` *имя_поля*

Также, с этой же целью может быть использована операция раскрытия указателя (см. раздел 9.2 на странице 23):

(*p).*имя_поля*

Листинг 70 представляет пример использования указателя на структуру и варианты получения значений полей такой структуры.

Листинг 70: Использование указателя на структуру

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

struct album{
  char group[33];
  char name[128];
  int year;
  int min, sec;
};

void printalbum(struct album * alb){
  printf("Group   : %s\n", alb->group);
  printf("Album   : %s\n", alb->name);
  printf("Year    : %d\n", alb->year);
  printf("Duration: %02d:%02d\n\n", (*alb).min, (*alb).sec);
}

int main(){
  struct album d1, d2, d3;

  strcpy(d1.group, "The Beatles");
  strcpy(d1.name, "Sgt. Pepper's Lonely Hearts Club Band");
  d1.year = 1967;
  d1.min = 40;
  d1.sec = 37;

  strcpy(d2.group, "Led Zeppelin");
  strcpy(d2.name, "Presence");
  d2.year = 1976;
  d2.min = d1.min+4;
  d2.sec = 3;

  strcpy(d3.group, "Pink Floyd");
  strcpy(d3.name, "The Dark Side Of The Moon");
  d3.year = 1973;
  d3.min = 43;
  d3.sec = 49;

  printalbum(&d1);
  printalbum(&d2);
  printalbum(&d3);

  return 0;
}
```

13.3 Передача структур в качестве параметра

Передача структур в качестве параметров функций обычно происходит через указатели на структуры (см. листинг 70).

13.4 Вложенные структуры

В качестве полей структур могут быть использованы другие структуры (такие структуры называются вложенными). Такие структуры должны быть определены до содержащей их структуры (см. листинг 71).

Листинг 71: Использование вложенных структур

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

struct time{
  int min, sec;
};

struct album{
  char group[33];
  char name[128];
  int year;
  struct time duration;
};

void printalbum(struct album * alb){
  printf("Group   : %s\n", alb->group);
  printf("Album   : %s\n", alb->name);
  printf("Year    : %d\n", alb->year);
  printf("Duration: %02d:%02d\n\n", alb->duration.min, (*alb).duration.sec);
}

int main(){
  struct album d4;

  strcpy(d4.group, "Deep Purple");
  strcpy(d4.name, "Machine Head");
  d4.year = 1972;
  d4.duration.min = 41;
  d4.duration.sec = 00;

  printalbum(&d4);

  return 0;
}
```

13.5 Массивы структур

Описание массива структур аналогично описанию массива любого другого типа.

Обработка элементов массива структур производится аналогично обработке элементов массива любого другого типа (см. листинг 72).

Листинг 72: Использование массива структур

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

struct time{
  int min, sec;
};

struct album{
  char group[33];
  char name[128];
  int year;
  struct time duration;
};

void printalbum(struct album * alb){
  printf("Group   : %s\n", alb->group);
  printf("Album   : %s\n", alb->name);
  printf("Year    : %d\n", alb->year);
  printf("Duration: %02d:%02d\n\n", alb->duration.min, (*alb).duration.sec);
```

```
}

int main(){
  struct album d[5];
  int i;

  strcpy(d[0].group, "The Beatles");
  strcpy(d[0].name, "Sgt. Pepper's Lonely Hearts Club Band");
  d[0].year = 1967;
  d[0].duration.min = 40;
  d[0].duration.sec = 37;

  strcpy(d[1].group, "Led Zeppelin");
  strcpy(d[1].name, "Presence");
  d[1].year = 1976;
  d[1].duration.min = d[0].duration.min+4;
  d[1].duration.sec = 3;

  strcpy(d[2].group, "Pink Floyd");
  strcpy(d[2].name, "The Dark Side Of The Moon");
  d[2].year = 1973;
  d[2].duration.min = 43;
  d[2].duration.sec = 49;

  strcpy(d[3].group, "Deep Purple");
  strcpy(d[3].name, "Machine Head");
  d[3].year = 1972;
  d[3].duration.min = 41;
  d[3].duration.sec = 00;

  for(i=0; i<4; i++){
    printalbum(d+i);
  }

  return 0;
}
```

14 Объединения

Объединение — это средство, позволяющее запоминать данные различных типов в одном и том же месте памяти.

Объединения описываются так же, как и структуры, за исключением того, что вместо ключевого слова **struct** используется ключевое слово **union**.

Доступ к полям объединения осуществляется аналогично доступу к полям структуры.

Поля объединения разделяют одну и ту же область памяти. Размер памяти, занимаемой объединением, равен размеру памяти, занимаемому самым «большим» из полей объединения (см. листинг 73 и соответствующую схему размещения полей объединения в памяти, представленную на рисунке 24).

Листинг 73: Использование объединений

```
#include<stdio.h>

union ttt {
  int digit;
  double bigfl;
  char c;
};

void printun(union ttt * t){
  printf("digit = %d\n", t->digit);
  printf("bigfl = %e\n", t->bigfl);
```

```
    printf("c_____=_\'%c\'\n\n", t->c);
}

int main(){
    static union ttt types;
    double tmp;

    printf("\n");

    types.c = '$';
    printf("types.c_=_\'$\':\n");
    printun(&types);

    types.digit = 77;
    printf("types.digit_=_77:\n");
    printun(&types);

    types.bigfl = 9.952931e-310;
    printf("types.bigfl_=_9.952931e-310:\n");
    printun(&types);

    return 0;
}
```

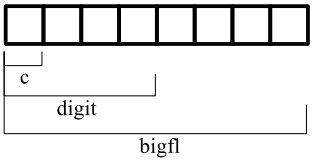


Рис. 24: Размещение полей объединения в памяти

15 Динамическое распределение памяти

Возможность использования динамического распределения памяти позволяет строить достаточно сложные структуры данных, такие как: граф, дерево, стек, очередь и другие.

В системе программирования C **gcc** описания функций распределения памяти находятся в файле **stdlib.h** (в других системах программирования C описания этих функций могут располагаться в других файлах).

15.1 Операция sizeof

Операция **sizeof** позволяет получить размер памяти в байтах, занимаемой типом данных или переменной, которые указаны как операнды данной операции.

Может быть использован один из следующих видов операции **sizeof** (см. листинг 74):

sizeof *имя_переменной*

sizeof (*имя_переменной*)

sizeof (*имя_типа*)

Листинг 74: Использование операции sizeof

```
#include<stdio.h>

struct time{
    int min, sec;
};

struct album{
    char group[33];
```

```
  char name[128];
  int year;
  struct time duration;
};

union ttt {
  int digit;
  double bigfl;
  char c;
};

int main(){
  struct album d[5];
  union ttt types;

  printf("sizeof(int) = %d\n", sizeof(int));
  printf("sizeof d[1].year = %d\n", sizeof d[1].year);
  printf("sizeof(struct time) = %d\n", sizeof(struct time));
  printf("sizeof d[0].duration = %d\n", sizeof d[0].duration);
  printf("sizeof(struct album) = %d\n", sizeof(struct album));
  printf("sizeof d[1] = %d\n", sizeof d[1]);
  printf("sizeof d = %d\n", sizeof d);
  printf("sizeof(struct album[5]) = %d\n", sizeof(struct album[5]));
  printf("sizeof(types.digit) = %d\n", sizeof(types.digit));
  printf("sizeof types.bigfl = %d\n", sizeof types.bigfl);
  printf("sizeof(types.c) = %d\n", sizeof(types.c));
  printf("sizeof types = %d\n", sizeof types);
  printf("sizeof(struct album*) = %d\n", sizeof(struct album*));
  printf("sizeof(char*) = %d\n", sizeof(char*));

  return 0;
}
```

15.2 Основные функции динамического распределения памяти

15.2.1 Выделение блока памяти определённого размера в байтах

Описание функции Выделение блока памяти определённого размера в байтах:

void * malloc(**int** size);

В качестве результата функция возвращает указатель (адрес) на размещенный непрерывный блок памяти. Размер этого блока в байтах, задаётся параметром `size`. Если блок такого размера разместить не удаётся, то функция возвращает значение `NULL` (см. листинг 75).

Листинг 75: Пример обращения к функции malloc()

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

int main(){
  char* groupname;

  if(groupname = (char*)malloc(128)){
    strcpy(groupname, "Omega");
  }
  else{
    printf("No memory\n");
    return 0;
  }

  printf("groupname: %s\n", groupname);
```

```
    printf("strlen(gropname): %d\n", strlen(groupname));

    free(groupname);

    return 0;
  }
```

15.2.2 Выделение блока памяти под определённое количество элементов определённого типа (динамический массив)

Описание функции выделения блока памяти под динамический массив:

void * calloc(**int** nelem, **int** elsize);

Функция обеспечивает выделение непрерывного блока памяти, размер которой в байтах равен произведению значений параметров `nelem` (количество элементов) и `elsize` (количество байт, занимаемых элементом). В качестве результата функция возвращает указатель (адрес) на размещенный блок памяти. Если блок памяти такого размера разместить не удается, то функция возвращает значение NULL. После размещения выделенный блок памяти инициализируется значением `0` (см. листинг 76).

Листинг 76: Пример обращения к функции calloc()

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

struct time{
  int min, sec;
};

struct album{
  char group[33];
  char name[128];
  int year;
  struct time duration;
};

void printalbum(struct album * alb){
  printf("Group   : %s\n", alb->group);
  printf("Album   : %s\n", alb->name);
  printf("Year    : %d\n", alb->year);
  printf("Duration: %02d:%02d\n\n", alb->duration.min, (*alb).duration.sec);
}

int main(){
  struct album * d;
  int i;

  if(!(d = (struct album *)calloc(4, sizeof(struct album)))){
    printf("No memory!\n");
    return 0;
  }

  strcpy(d[0].group, "The Beatles");
  strcpy(d[0].name, "Sgt. Pepper's Lonely Hearts Club Band");
  d[0].year = 1967;
  d[0].duration.min = 40;
  d[0].duration.sec = 37;

  strcpy(d[1].group, "Led Zeppelin");
  strcpy(d[1].name, "Presence");
  d[1].year = 1976;
  d[1].duration.min = d[0].duration.min+4;
```

```
    d[1].duration.sec = 3;

    strcpy((d+2)->group, "Pink Floyd");
    strcpy((d+2)->name, "The Dark Side Of The Moon");
    (d+2)->year = 1973;
    (d+2)->duration.min = 43;
    (d+2)->duration.sec = 49;

    strcpy((*(d+3)).group, "Deep Purple");
    strcpy((*(d+3)).name, "Machine Head");
    (*(d+3)).year = 1972;
    (*(d+3)).duration.min = 41;
    (*(d+3)).duration.sec = 00;

    for(i=0; i<4; i++){
      printalbum(d+i);
    }

    free(d);

    return 0;
  }
```

15.2.3 Перераспределение динамически выделенного блока памяти

Описание функции перераспределение динамически выделенного блока памяти:

void * realloc (**void *** ptr, **int** newsize);

Функция изменяет размер ранее динамически выделенного непрерывного блока памяти, адрес которого соответствует значению параметра `ptr`. Новый размер в байтах задаётся параметром `newsize`. Если пространство за границей исходного блока памяти является недоступным, то функция выделяет новый непрерывный блок памяти размером, соответствующим значению параметра `newsize`, после чего содержимое исходного блока будет скопировано в этот новый блок памяти и, при этом, исходный блок памяти становится доступным для последующих действий, связанных с распределением памяти. В качестве результата функция возвращает значение указателя (адреса) на вновь размещённый блок памяти. Если блок памяти размером в байтах, заданном значением параметра `newsize`, разместить не удаётся, то функция возвращает значение NULL и, при этом, исходный блок памяти никак не изменяется. Обращение к функции `realloc(ptr, newsize)`, где в качестве значения фактического параметра `ptr` используется значение `NULL`, аналогично обращению к функции `malloc(newsize)`. Примеры обращений к функции `realloc(...)` представлены листингом 77).

Листинг 77: Примеры обращения к функции realloc()

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

struct time{
  int min, sec;
};

struct album{
  char group[33];
  char name[128];
  int year;
  struct time duration;
};

void printalbum(struct album * alb){
  printf("Group   : %s\n", alb->group);
  printf("Album   : %s\n", alb->name);
  printf("Year    : %d\n", alb->year);
```

```
  printf("Duration: %02d:%02d\n\n", alb->duration.min, (*alb).duration.sec);
}

int main(){
  struct album * d;
  struct album * tmp;
  int i;

  if(!(d = (struct album *)realloc(NULL, sizeof(struct album)))){
    printf("No memory!\n");
    return 0;
  }

  strcpy(d[0].group, "The Beatles");
  strcpy(d[0].name, "Sgt. Pepper's Lonely Hearts Club Band");
  d[0].year = 1967;
  d[0].duration.min = 40;
  d[0].duration.sec = 37;

  if(!(tmp = (struct album *)realloc(d, sizeof(struct album)*2))){
    printf("No memory!\n");
    return 0;
  }

  d = tmp;
  strcpy(d[1].group, "Led Zeppelin");
  strcpy(d[1].name, "Presence");
  d[1].year = 1976;
  d[1].duration.min = d[0].duration.min+4;
  d[1].duration.sec = 3;

  if(!(tmp = (struct album *)realloc(d, sizeof(struct album)*3))){
    printf("No memory!\n");
    return 0;
  }

  d = tmp;
  strcpy((d+2)->group, "Pink Floyd");
  strcpy((d+2)->name, "The Dark Side Of The Moon");
  (d+2)->year = 1973;
  (d+2)->duration.min = 43;
  (d+2)->duration.sec = 49;

  if(!(tmp = (struct album *)realloc(d, sizeof(struct album)*4))){
    printf("No memory!\n");
    return 0;
  }

  d = tmp;
  strcpy((*(d+3)).group, "Deep Purple");
  strcpy((*(d+3)).name, "Machine Head");
  (*(d+3)).year = 1972;
  (*(d+3)).duration.min = 41;
  (*(d+3)).duration.sec = 00;

  for(i=0; i<4; i++){
    printalbum(d+i);
  }

  free(d);

  return 0;
```

```
}
```

15.2.4 Функция освобождения динамически размещенной памяти

Описание функции освобождения динамически размещённого ранее непрерывного блока памяти:

void free(**void *** block);

Функция освобождает динамически выделенный ранее непрерывный блок памяти, адрес которого задаётся значением параметра `block`. После выполнения этой функции освобождённый блок памяти становится доступным для последующих действий, связанных с распределением памяти. Примеры обращения к функции `free(...)` представлены в листингах 75, 76, 77 на страницах 45, 46, 47 соответственно.

15.3 Пример использования функций динамического распределения памяти при обработке списков

Задание: пользователем с экрана вводится последовательность целых положительных чисел. Признаком конца последовательности является ввод числа, значение которого не больше `0`. Напечатать введённые числа в обратном порядке.

Для решения задачи будет использована такая структура, как однонаправленный список (см. рисунок 25).

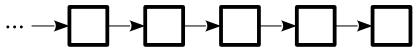


Рис. 25: Схема однонаправленного списка

Листинг 78 представляет исходный текст программы, решающей поставленную задачу.

Листинг 78: Пример обработки списка

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

struct s {
  int info;
  struct s * ptr;
};

int main() {
  struct s * tmp1, * tmp2;
  int k;

  tmp1 = tmp2 = NULL;

  /* Ввод последовательности */
  printf("Enter sequence:\n");
  scanf("%d", &k);
  while(k>0) {
    if(!(tmp1 = (struct s *)malloc(sizeof(struct s)))) {
      printf("No memory...\n");
      break;
    }
    tmp1->info = k;
    tmp1->ptr = tmp2;
    tmp2 = tmp1;
    scanf("%d", &k);
  }

  /* Печать введённой последовательности в обратном порядке*/
  printf("\nResult:\n");
  while(tmp2) {
    printf("%d\n", tmp2->info);
    tmp1 = tmp2->ptr;
```

```
        free(tmp2);
        tmp2 = tmp1;
    }

    return 0;
}
```

На рисунке 26 схематично представлен соответствующий приведённой программе однонаправленный список для трёх введённых элементов.

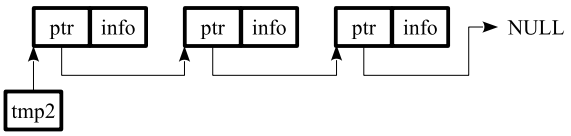


Рис. 26: Схема однонаправленного списка к листингу 78

16 Препроцессор

Препроцессор выполняет предварительную обработку исходного текста на языке программирования C, осуществляя в нем замены в соответствии со специальными директивами. После работы препроцессора начинает работу собственно компилятор, который уже переводит программу в машинный код.

Директивы препроцессора располагаются в исходном тексте программы и начинаются с символа #.

16.1 Директива препроцессора #include

Директива `#include` может быть представлена в одном из следующих форматов:

`#include <`*имя_файла*`>`

`#include "`*имя_файла*`"`

Когда в тексте программы встречается директива `#include` препроцессор ищет указанный за ней файл и включает его содержимое в исходный текст программы вместо этой директивы (см. рисунок 4 на странице 6).

Файл может указываться с помощью абсолютного или относительного пути.

Файл включаемый директивой `#include` может содержать произвольный текст на языке программирования C. Однако обычно он содержит заголовки функций, описание констант, внешних переменных и т.д.

По этой причине этот файл называют заголовочным и дают ему расширение `.h` (от слова `header`).

Листинг 79 представляет пример использования директивы `#include` в операционной системе Windows.

Листинг 79: Пример директивы #include в ОС Windows

```
#include "mydir\inc\const.h"
#include "d:\other\proto.h"
#include <sys\stat.h>
```

Листинг 80 представляет пример использования директивы `#include` в UNIX-системах.

Листинг 80: Пример директивы #include в UNIX-системах

```
#include "mydir/inc/const.h"
#include "/other/proto.h"
#include <sys/stat.h>
```

Если имя файла заключено в угловые скобки, то считается, что файл расположен в одном из стандартных каталогов системы программирования C. Обычно в этих каталогах хранятся заголовочные файлы, поставляемые с системой программирования.

Если имя файла указано в кавычках, то поиск файла начинается с текущего каталога, а затем продолжается в стандартных каталогах.

16.2 Директива препроцессора #define

Формат директивы `#define`:

`#define` *макроопределение строка_замещения*

- *макроопределение* — имя, составленное по правилам построения имен языка программирования C;
- *строка_замещения* — произвольная текстовая строка.

Директива `#define` указывает препроцессору заменять все вхождения *макроопределения* в исходном тексте программы на его *строку замещения*. Процесс замещения называется **макрорасширением**.

Макроопределение действует с момента появления директивы `#define` и до конца файла. Макрорасширения не происходит если макроопределение встречается в строковой константе. В качестве примера см. листинг 81.

Листинг 81: Использование директивы #define

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

#define GLEN 33
#define NLEN 128
#define ALBCOUNT 3

struct time{
  int min, sec;
};

struct album{
  char group[GLEN];
  char name[NLEN];
  int year;
  struct time duration;
};

void printalbum(struct album * alb){
  printf("Group   : %s\n", alb->group);
  printf("Album   : %s\n", alb->name);
  printf("Year    : %d\n", alb->year);
  printf("Duration: %02d:%02d\n\n", alb->duration.min, (*alb).duration.sec);
}

int main(){
  struct album d[ALBCOUNT];
  int i;

  strcpy(d[0].group, "The Beatles");
  strcpy(d[0].name, "Sgt. Pepper's Lonely Hearts Club Band");
  d[0].year = 1967;
  d[0].duration.min = 40;
  d[0].duration.sec = 37;

  strcpy(d[1].group, "Led Zeppelin");
  strcpy(d[1].name, "Presence");
  d[1].year = 1976;
  d[1].duration.min = d[0].duration.min+4;
  d[1].duration.sec = 3;

  strcpy(d[ALBCOUNT-1].group, "Pink Floyd");
  strcpy(d[ALBCOUNT-1].name, "The Dark Side Of The Moon");
  d[ALBCOUNT-1].year = 1973;
  d[ALBCOUNT-1].duration.min = 43;
  d[ALBCOUNT-1].duration.sec = 49;
```

```
    for(i=0; i<ALBCOUNT; i++){
      printalbum(d+i);
    }

    return 0;
}
```

Макроопределение может содержать параметры, и тогда оно имеет вид:

`#define` *макроопределение*([*параметр1*, *параметр2*, ...]) *строка_замещения*

Препроцессор сначала заменяет в строке замещения все вхождения формальных *параметров* на соответствующие фактические параметры, а затем заменяет макроопределение на полученную строку замещения (см. листинг 82).

Листинг 82: Использование макроопределений с параметрами

```
#include<stdio.h>

#define PRTSQR(VALUE, TEXT) printf("%s value: %d\n", TEXT, VALUE)
#define SQUAREBAD(X) (X*X)
#define SQUARE(X) ((X)*(X))

int main(){
  int m;
  long int n;

  m = 10;
  n = SQUAREBAD(m+5); // => m+5*m+5
  PRTSQR(n, "SQUAREBAD(m+5)");

  n = SQUARE(m+5);    // => (m+5)*(m+5)
  PRTSQR(n, "SQUARE(m+5)");

  return 0;
}
```

16.3 Директива препроцессора #undef

Директива `#undef` обеспечивает отмену директивы `#define` для какого-либо макроопределения (см. листинг 83):

`#undef` *макроопределение*

Листинг 83: Использование директивы #undef

```
#include<stdio.h>

int main(){
  #define MODE    17
  printf("MODE: %d\n", MODE);

  #undef MODE
  // printf("MODE: %d\n", MODE);
  /* MODE не определено */

  #define MODE    33
  printf("MODE: %d\n", MODE);

  #define LEVEL   "LOW"
  printf("LEVEL: %s\n", LEVEL);

  #undef LEVEL
  // printf("LEVEL: %s\n", LEVEL);
```

```
    /* LEVEL не определено */

    #define LEVEL    "HIGH"
    printf("LEVEL: %s\n", LEVEL);

    return 0;
}
```

16.4 Директивы условной компиляции

Директивы условной компиляции позволяют изменить естественный порядок компиляции.

Директива #ifdef. Формат директивы `#ifdef`:

`#ifdef` *макроопределение*

Директива `#ifdef` проверяет, определено ли *макроопределение* в препроцессоре в данный момент.

Директива #ifndef. Формат директивы `#ifndef`:

`#ifndef` *макроопределение*

Директива `#ifndef` проверяет, является ли *макроопределение* неопределенным в препроцессоре в данный момент.

Директивы #else и #endif Директивы `#else` и `#endif` используются совместно с директивами `#ifdef` и `#ifndef`.

После любой из двух директив `#ifdef` и `#ifndef` может стоять произвольное количество строк, возможно содержащих директиву `#else` и далее до директивы `#endif`.

Если проверенное условие **истинно**, то все строки между директивами `#else` и `#endif` игнорируются.

Если проверенное условие **ложно**, то все строки между проверкой и директивой `#else` или, в случае отсутствия таковой, между проверкой и директивой `#endif` игнорируются (см. листинг 84).

Листинг 84: Использование директив условной компиляции

```
#include<stdio.h>

#define SIMPLE

int main(){
    int i;

    #define MODE     17
    #ifdef MODE
    printf("1 - MODE: %d\n", MODE);
    #endif

    #undef MODE

    #ifdef MODE
    printf("2 - MODE: %d\n", MODE);
    #endif

    #ifndef MODE
    printf("3 - MODE undefined\n");
    #endif

    #define LEVEL    "LOW"
    #ifdef LEVEL
    printf("4 - LEVEL: %s\n", LEVEL);
    #else
    printf("5 - LEVEL undefined\n");
    #endif
```

```
    #undef LEVEL
    #ifdef LEVEL
    printf("6 - LEVEL: %s\n", LEVEL);
    #else
    printf("7 - LEVEL undefined\n");
    #endif

    #define LEVEL    "HIGH"
    #ifdef LEVEL
    printf("6 - LEVEL: %s\n", LEVEL);
    #else
    printf("7 - LEVEL undefined\n");
    #endif

    #ifdef SIMPLE
    printf("8 - SIMPLE defined\n");

    i = 123;
    printf("i = %d\n", i);
    #else
    printf("9 - SIMPLE undefined\n");

    i = 321;
    printf("i = %d\n", i);
    #endif

    return 0;
  }
```

17 Операция определения типа typedef

Операция определения типа **typedef** позволяет давать новые имена типам данных, допустимых в языке программирования C.

Общий вид операции **typedef**:

typedef *тип имя*;

- *тип* — любой допустимый языком программирования C тип данных;
- *имя* — новое имя, назначаемое соответствующему типу данных.

Операция **typedef** вводит не новые типы, а только синонимы для типов, которые могли бы быть определены другим путем.

Операция typedef напоминает по действию директиву препроцессора `#define`, но имеет следующие отличия:

- в отличие от директивы `#define` операция **typedef** дает символические имена, но ограничивается только типами данных;
- операция **typedef** выполняется компилятором, а не препроцессором.

Область действия определения типа с использованием операции **typedef** зависит от расположения ключевого слова **typedef**.

Если определение находится внутри функции, то область действия локальна и ограничена этой функцией.

Если определение расположено вне функции, то область действия глобальна по отношению к функциям, содержащимся в соответствующем исходном файле и расположенным после данного определения (см. листинг 85).

Листинг 85: Использование операции typedef

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

```
3   #include <stdlib.h>
4
5   #define STRLEN 81
6   #define NEWPTR int *
7
8   typedef float Real;
9   typedef int Integer, *IntPtr;
10  typedef char Stroka[STRLEN];
11
12  typedef struct song_list {
13    Stroka song_name;
14    struct song_list * next ;
15  } SongInfo, *SongPtr;
16
17  SongPtr CreateAlbum() {
18    NEWPTR a, b, c;
19    IntPtr d, e, f;
20    Stroka buffer;
21    SongPtr ret, tmpnew, tmpprev;
22
23    ret = tmpnew = tmpprev = NULL;
24
25    while(1) {
26      fgets(buffer, STRLEN, stdin);
27      if(buffer[0] == '.') break;
28      if(ret == NULL) {
29        tmpnew = (SongPtr)calloc(1, sizeof(SongInfo));
30        if(!tmpnew) {
31          printf("No memory...\n");
32          return ret;
33        }
34        strcpy(tmpnew->song_name, buffer);
35        tmpnew->next = NULL;
36        ret = tmpnew;
37      }
38      else {
39        tmpprev = tmpnew;
40        tmpnew = (SongPtr)calloc(1, sizeof(SongInfo));
41        if(!tmpnew) {
42          printf("No memory...\n");
43          rturn ret;
44        }
45        tmpprev->next = tmpnew;
46        strcpy(tmpnew->song_name, buffer);
47        tmpnew->next = NULL;
48      }
49    }
50
51    return ret;
52  }
53
54  int main() {
55    SongPtr BeatAlbum;
56    SongPtr tmp;
57    printf("Enter the song names...\n");
58    BeatAlbum = CreateAlbum();
59
60    while(BeatAlbum) {
61      fputs(BeatAlbum->song_name, stdout);
62      tmp = BeatAlbum;
63      BeatAlbum = BeatAlbum->next;
64      free(tmp);
```

```
65        }
66        return 0;
67
68    }
```

Пояснения к исходному тексту, представленному листингом 85:

- строка 8 — определение нового имени `Real` для типа **float**;
- строка 9 — определение новых имён `Integer` и `IntPtr` для типов **int** и **int*** соответственно;
- строка 10 — определение нового имени `Stroka` для типа массив **char** из 81 элемента (см. также строку 5 листинга);
- строки 12–15 — определение новых имён `SongInfo` и `SongPtr` для типов **struct song_list** и **struct song_list*** соответственно;
- строка 18 — объявление переменной `a` типа **int*** и переменных `b`, `c` типа **int** (см. также строку 6 листинга);
- строка 19 — объявление переменных `d`, `e`, `f` типа **int*** (см. также строку 9 листинга);
- строка 20 — объявление переменной `buffer` как массива **char** из 81 элемента (см. также строку 10 листинга);
- строка 21 — объявление переменных `ret`, `tmpnew`, `tmpprev` типа **struct song_list*** (см. также строки 12–15 листинга).

18 Использование файлов в языке программирования C

Общая схема использования файлов в программе представлена на рисунке 27

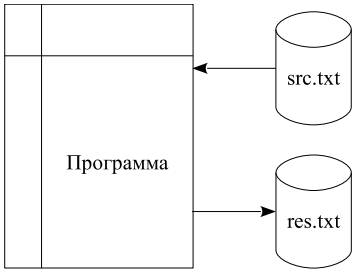


Рис. 27: Схема использования файлов

На рисунке 27 схематично представлена программа в оперативной памяти, файл с именем `src.txt`, из которого вводится исходная информация, и файл с именем `res.txt`, в который выводится результирующая информация, причём, оба этих файла расположены во внешней памяти (например, на магнитном диске).

Таким образом, необходимо связать конкретный файл во внешней памяти с программой, находящейся в оперативной памяти.

В языке программирования C файлы идентифицируются с помощью переменной-указателя на тип **FILE**. Говорится, что такого рода переменная представляет собой логический файл в программе. В каждый момент времени с логическим файлов может быть связан один единственный физический файл, размещённый во внешней памяти.

Для **установления связи логического файла** в программе **с физическим файлом**, размещённым во внешней памяти, используется функция открытия файла **fopen()**:

FILE * fopen(**char *** fname, **char *** mode);

- значение параметра `fname` — адрес строки символов, содержащей имя физического файла, размещённого во внешней памяти;
- значение параметра `mode` — адрес строки символов, содержащей описание режима открытия файла, например:
  - `"r"` — файл открывается только на чтение;

– `"w"` — файл открывается только на запись; если файл с соответствующим именем уже существует, то его содержимое будет уничтожено, то есть запись будет происходить в пустой файл, иначе будет создан новый файл;

– `"a"` — файл открывается для добавления; если соответствующий файл не существует, то он создается.

В случае успешного выполнения функция в качестве результата возвращает указатель (адрес памяти) на созданный логический файл.

Иначе функция возвращает значение `NULL`.

Для завершения связи логического файла в программе **с физическим файлом**, размещённым во внешней памяти, используется функция закрытия файла **fclose()**:

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

- `stream` — переменная-указатель, соответствующая «закрываемому» логическому файлу.

Функция возвращает значение `0` в случае успешного выполнения.

Иначе функция возвращает значение EOF.

После вызова функции `fclose()` соответствующий логический файл может быть использован для связи с другими физическими файлами (см. листинг 86).

Листинг 86: Использование функций fopen() и fclose()

```
#include <stdio.h>

int main() {
  FILE * f;
  if(f = fopen("test.txt", "r")) {
    // ...обработка файла test.txt...
    fclose(f);
  }
  else{
    printf("Ошибка открытия файла test.txt\n");
  }
  // ...
  if(f = fopen("other.txt", "w")) {
    // ...запись информации в файл other.txt...
    fclose(f);
  }
  else{
    printf("Ошибка открытия файла other.txt\n");
  }

  return 0;
}
```

В системе программирования C предопределены, в частности, следующие два логических файла:

- **stdin** — стандартный файл ввода, который, как правило, соответствует клавиатуре;
- **stdout** — стандартный файл вывода, который, как правило, соответствует терминалу.

Для этих логических файлов явное использование функций `fopen()` и `fclose()` не нужно.

19 Основные функции ввода-вывода

Наиболее часто используемыми в процессе изучения языка программирования C функциями ввода-вывода стандартной библиотеки системы программирования C являются:

- функции форматного ввода **scanf(...)**, **fscanf(...)**;
- функции форматного вывода **printf(...)**, **fprintf(...)**;
- функция ввода строки символов **fgets(...)**;
- функции вывода строк символов **puts(...)**, **fputs(...)**.

Подробное описание этих и других функций стандартной библиотеки системы программирования C можно найти в [1] и [2].

20 Указатель на функцию

В языке программирования С функция не является переменной, но можно определить переменную-указатель на функцию и работать с ней как с обычной переменной: присваивать, размещать в массиве, передавать в качестве параметра функции и т.д. Фактически, в переменной-указателе на функцию может хранится адрес области памяти, по которому может быть передано управление.

Описание указателя на функцию:

тип_результата (**имя_переменной_указателя_на_функцию*)([*список параметров*]);

Листинг 87 демонстрирует пример использования переменной-указателя на функцию.

Листинг 87: Пример использования переменной-указателя на функцию

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #define N 13
5
6  int mysort(void *array, int n, int size,
7             int (*compare)(void *a, void *b),
8             void (*swap)(void *c, void *d)){
9    int i, j;
10
11   if(n <= 0){
12     return 1;
13   }
14
15   for(i = 0; i<n; i++){
16     for(j = i; j<n; j++){
17       if((*compare)(array+(j*size), array+(i*size))<0){
18         (*swap)(array+(i*size), array+(j*size));
19       }
20     }
21   }
22   return 0;
23 }
24
25
26 int compareint(int *a, int *b){
27   if((*a) < (*b)) return -1;
28   if((*a) > (*b)) return 1;
29   return 0;
30 }
31
32 void swapint(int *a, int *b){
33   int tmp;
34   tmp = *a;
35   *a = *b;
36   *b = tmp;
37 }
38
39 int comparefloat(float *a, float *b){
40   if( (((*a)-(*b))>0.0?(*a)-(*b):(*b)-(*a)) < 0.00001) return 0;
41   if((*a) < (*b)) return -1;
42   return 1;
43 }
44
45 void swapfloat(float *a, float *b){
46   float tmp;
47
48   tmp = *a;
49   *a = *b;
50   *b = tmp;
```

```
51    }
52
53    int comparechar(char *a, char *b){
54      if((*a) < (*b)) return -1;
55      if((*a) > (*b)) return 1;
56      return 0;
57    }
58
59    void swapchar(char *a, char *b){
60      char tmp;
61
62      tmp = *a;
63      *a = *b;
64      *b = tmp;
65    }
66
67    int comparestring(char **a, char **b){
68      if(strcmp(*a, *b) < 0) return -1;
69      if(strcmp(*a, *b) > 0) return 1;
70      return 0;
71    }
72
73    void swapstring(char **a, char **b){
74      char *tmp;
75
76      tmp = *a;
77      *a = *b;
78      *b = tmp;
79    }
80
81    int main(){
82      int (*comp)();
83      void (*swp)();
84      int srcarray[N] = {1, 187, 5, 56, 9, 15, 74, 53, 42, 31, 20, 1, 13};
85      float floatarray[N] = {1.1, 187.187, 5.5, 56.56, 9.9, 15.15, 74.74,
86                             53.53, 42.42, 31.31, 20.2, 1.1, 13.13};
87      char chararray[N] = {'1', 'Q', '5', '5', '9', 'C', 'A', 'Z', 'B',
88                           'S', '4', '2', 'D'};
89      char *strarray[N] = {"One", "One hundred eighty seven", "Five",
90                           "Fifty six", "Nine", "Fifteen", "Seventy four",
91                           "Fifty three", "Forty two", "Thirty one",
92                           "Twenty", "One", "Thirteen"};
93      int i;
94
95      /* Sort INT */
96      comp = compareint;
97      swp  = swapint;
98      if(!mysort(srcarray, N, sizeof(int), comp, swp)){
99        printf("SORTED INT ARRAY:\n");
100       for(i = 0; i<N; i++){
101         printf("srcarray[%02d] = %4d\n", i, srcarray[i]);
102       }
103     }
104
105     /* Sort FLOAT */
106     comp = comparefloat;
107     swp  = swapfloat;
108     if(!mysort(floatarray, N, sizeof(float), comp, swp)){
109       printf("\nSORTED FLOAT ARRAY:\n");
110       for(i = 0; i<N; i++){
111         printf("floatarray[%02d] = %8.3f\n", i, floatarray[i]);
112       }
```

```
113     }
114
115     /* Sort CHAR */
116     comp = comparechar;
117     swp  = swapchar;
118     if(!mysort(chararray, N, sizeof(char), comp, swp)){
119       printf("\nSORTED CHAR ARRAY:\n");
120       for(i = 0; i<N; i++){
121         printf("chararray[%02d] = %c\n", i, chararray[i]);
122       }
123     }
124
125     /* Sort STRING */
126     comp = comparestring;
127     swp  = swapstring;
128     if(!mysort(strarray, N, sizeof(char*), comp, swp)){
129       printf("\nSORTED STRING ARRAY:\n");
130       for(i = 0; i<N; i++){
131         printf("strarray[%02d] = %s\n", i, strarray[i]);
132       }
133     }
134
135     return 0;
136
137   }
```

Пояснения к листингу 87:

- строки 7, 8 — формальные параметры функции `mysort(...)`, являющиеся параметрами-указателями на функцию;
- строки 17, 18 — обращение к функциям посредством параметров-указателей на функцию;
- строки 82, 83 — объявление переменных-указателей на функцию;
- строки 96, 97, 106, 107, 116, 117, 126, 127 — присваивание переменным-указателям на функцию значений, в качестве которых выступают имена функций, реализация которых представлена в строках 26–79;
- строки 98, 108, 118, 128 — вызовы функции `mysort(...)` и передача в качестве фактических параметров переменных-указателей на функцию.

21 Аргументы функции main()

В общем случае функция `main()` может принимать от нуля до трёх аргументов:

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

- значением аргумента `argc` является количество элементов массива `argv`;
- массив `argv` содержит указатели (адреса) на строки символов, набранные в командной строке при запуске программы на выполнение (параметры командной строки) и отделённые друг от друга одним или несколькими пробелами;
- массив `env` содержит указатели (адреса) на строки символов, в которых содержатся имена заданных переменных окружения сеанса пользователя операционной системы и их значения; последний элемент массива `env` содержит значение `NULL`.

Альтернативное описание функции `main()` с аргументами:

int main(**int** argc, **char** ****argv**, **char** ****env**);

Листинг 88 представляет собой пример программы, обеспечивающей печать информации, переданной в аргументах функции `main()` после запуска этой программы на исполнение.

Листинг 88: Аргументы функции main()

```
#include <stdio.h>
int main(int argc, char *argv[], char *env[]) {
    int i;

    printf("Number of command line arguments: %d \n\n", argc);

    printf("Command line arguments:\n");
    for (i = 0; i < argc; i++)
        printf("Argument %2d: %s\n", i, argv[i]);

    printf("\n\nEnvironment variables:\n");
    for (i = 0; env[i] != NULL; i++)
        printf("%s\n", env[i]);

    return 0;
}
```

Операционная система сама по себе является программой. Соответственно, когда выполняется запуск программы в рамках операционной системы, операционная система выполняет сначала размещение всех необходимых областей памяти, заполнение их информацией, а затем передаёт управление и соответствующие значения аргументов функции `main()` (см. рисунок 28).

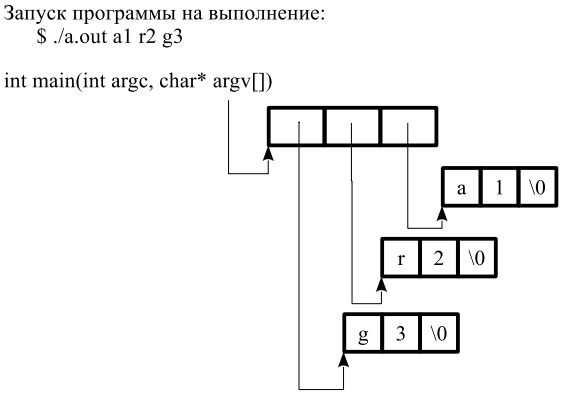


Рис. 28: Схема размещения аргументов функции main()

Список литературы

[1] Б. Керниган, Д. Ритчи, Язык программирования C *(любое издание, кроме первого)*

[2] The GNU C Library Reference Manual