МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет
им. Н.И. Лобачевского»

С.Н. Карпенко

Основы программирования на языке С

Учебно-методическое пособие

Рекомендовано методической комиссией института ИТММ
для студентов ННГУ, обучающихся по направлениям подготовки
09.03.04. «Программная инженерия» и 02.03.02 «Фундаментальная
информатика и информационные технологии»

Нижний Новгород
2018

УДК 004:655.4/.5(075)
ББК 681.3:Ч617(075)

К 89 Карпенко С.Н.. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С: учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2018. – 129 с.

Рецензент: д.т.н., профессор **В.Е. Турлапов**

Предлагаемое учебно-методическое пособие посвящено основам программирования на языке С. Дается представление о базовых типах данных, вычислительных структурах, принципах организации модульных (многофайловых) программ, массивах и указателях, функциях, строках, структурах, объединениях, битовых полях. Материал иллюстрируется большим количеством примеров.

Для студентов первого курса, обучающихся по направлениям подготовки 09.03.04. «Программная инженерия» и 02.03.02 «Фундаментальная информатика и информационные технологии».

УДК 004:655.4/.5(075)
ББК 681.3:Ч617(075)

ВВЕДЕНИЕ

Предлагаемое пособие предназначено для студентов первого курса ИТ специальностей и посвящено основам программирования. Материал пособия излагается на примере языка С, одного из наиболее широко распростаненных в настоящее время языков программирования. В пособии рассматриваются следующие темы: ведение и первая программа на С, базовые (фундаментальные) типы данных, вычислительные структуры, формирование модульных (многофайловых) программ, массивы и указатели, функции, работа со строками, структуры, объединения, битовые поля.

Пособие составлено на основе материалов лекций, прочитанных автором в течение ряда лет.

1. ВВЕДЕНИЕ И ПЕРВАЯ ПРОГРАММА НА С

1.1. Немного истории

Язык программирования С появился в 1970 г. Автором языка является Денис Ритчи, сотрудник компании AT&T Bell Laboratories. Группа сотрудников этой компании занималась разработкой операционной системы UNIX для компьютера PDP-11. Первоначально разработка велась в машинных кодах, затем для упрощения разработки стали применять самостоятельно созданный ассемблер. Далее стало известно, что разрабатываемую операционную системы надо будет реализовывать на компьютерах других типов. У разработчиков появилась идея не повторять реализацию на других компьютерах, а разработать переносимую операционную систему. Для этого надо было разработать переносимый алгоритмический язык, на котором будет реализована операционная система. Первый разработанный язык (ассемблер) был назван язык А, следующая версия – язык В и окончательная версия – язык С.

Первый продукт (программа), которая была написана на С был компилятор с этого языка; второй – операционная система UNIX для PDP-11. В 1976 году был успешно выполнен перенос этой операционной системы на Interdata 8/32 и впоследствии на компьютеры других типов. Переносимость (и открытость) ОС UNIX позволила ей (и языку С) достаточно быстро распространиться по всему миру и в настоящее время эта операционная система есть для всех современных компьютеров и платформ.

В 1978 году вышла первая книга по С: Brian W. Kernighan, Dennis M. Ritchie. The C programming Language, которая достаточно быстро была переведена на многие языки (первый перевод на русский язык вышел в 1985 году).

Известны следующие стандарты языка С:

- 1970 г. – C UNIX. Описанием стандарта является первая редакция книги Крернигана, Ритчи.
- 1987 г. - стандарт ANSI C
- 1989 г. C89 – ANSI/ISO C
- 1999 г. - текущий стандарт ISO 9899:1999

1.2. Общая характеристика языка С

Язык С сочетает возможности языка низкоуровневого программирования (ассемблера) и языка высокоуровневого программирования.

Как язык низкого уровня, С имеет возможность представления компьютерных (аппаратных) типов данных и набор логических операций на уровне побитового представления информации, операций сдвига кода, работы с адресами и регистрами. Уровень этих возможностей позволяет использовать С как язык системного программирования.

С другой стороны, С является современным алгоритмическим языком высокого уровня. В нем представлены все необходимые базовые типы данных и операции над ними; есть развитый механизм формирования абстрактных типов данных, работа с указателями, поддерживается парадигма процедурного и структурного программирования. Уровень этих возможностей позволяет использовать С как язык прикладного программирования.

Благодаря простоте самого языка компиляторы разрабатываются достаточно легко и существуют практически на всех современных платформах. Между тем, компиляторы языка С удачно сочетают относительно небольшое время компиляции с высокой производительностью получаемого кода: время выполнения программы, написанной на языке С мало отличается от времени выполнения аналогичной программы, написанной на ассемблере.

В языке С применяется препроцессор, директивы которого обрабатываются перед компиляцией исходного кода. Использование препроцессора позволяет существенно упростить исходный код программы за счет:

- записи необходимых объявлений в отдельный файл с автоматическим его включением в необходимые исполняемые файлы;
- механизма макросов, позволяющих настраивать программу на применение в различных аппаратных условиях;
- механизма условной компиляции, позволяющего предусматривать различные варианты фрагментов кода для возможного учета особенностей различных платформ.

1.3. Программа на языке С

Функции языка С

В отличие от языка Pascal, программа на языке С является набором функций, которые могут вызывать друг друга. Все функции программы С равноправны в том смысле, что они не могут вкладываться друг в друга как в языке Pascal.

Описание функций

Описание функции в языке С имеет вид:

<тип_возвращаемого_значения> <имя_функции>(<параметры>)

{

<тело функции>

return <возвращаемое значение>;

}

где:

- <параметры> — список исходных данных (формальных параметров) в виде:
- <тип> имя_переменной[,<тип> имя_переменной, …]
- <тело функции> — описание алгоритма функции
- <возвращаемое значение> — результат, возвращаемый из функции, типа <тип_возвращаемого_значения>

Пример.

```
double mlt(double p1, double p2)
{
  double mlt = p1 * p2;
  return mlt;
}
```

В приведенном примере функция mlt получает два параметра типа double (действительные) и возвращает их произведение.

Операторов return (вернуться в вызывающую функцию) может быть несколько.

Пример.

```
double min(double a, double b)
{
  if ( a < b ) return a;
  else         return b;
}
```

В приведенном примере функция min получает два действительных параметра и возвращает минимальное из них.

В отличие от языка Pascal, в языке C нет процедур. Если функция C ничего не возвращает, то в качестве возвращаемого типа надо указывать тип void (ничего не возвращаю). В операторе return в этом случае ничего указывать не надо.

Пример.

```
void toConsole(double a)
{
     printf("a = %4.2f", a);
     return;
}
```

В приведенном примере функция toConsole получает один действительный параметр и выводит его значение на консоль.

Вызов функций

Вызов функции в языке C имеет вид:

<имя_функции> (<передаваемые параметры>)

где:

<передаваемые параметры> — список передаваемых исходных данных (фактических параметров); элементом списка может быть константа, переменная или выражение.

Пример.

```
void foo()
{
  double a = 3.4, b = 8.6, c, d;
  c = mlt(a, b);
  d = mlt(3.7, c) + mlt(a, c - 2.9);
}
```

В приведенном примере:

- функция foo ничего не получает и ничего не возвращает;

- в функции foo объявляются четыре переменные типа double, две из которых инициируются начальным значениями;
- переменной c присваивается значение вызова функции mlt с параметрами a и b;
- переменной d присваивается значение выражения.

Функция main

Как отмечалось выше, программа на языке С является набором функций, которые могут вызывать друг друга. При этом, среди функций программы должна быть функция с именем main, с вызова которой начинает выполняться программа на языке С (головная функция программы). Описание функции main выглядит следующим образом:

```
int main()
{
     // Тело (действия) функции main
     // . . . . . .
     return 0;
}
```

Функция main имеет следующие особенности:

- функция main вызывается операционной системой, под управлением которой выполняется программа;
- по завершению выполнения функции main (т.е. завершению выполнения программы) функция main возвращает операционной системе так называемый код завершения типа int (return 0;). Код завершения сообщает операционной системе, как завершилось выполнение программы: ноль – нормальное завершение, не ноль – код аварийного завершения, по получении которого операционная система должна выполнить некоторые нестандартные действия;
- функция main может получать от операционной системы некоторые параметры, но об этом будем говорить в следующих лекциях.

Пример.

```
int main()
{
     double a, b, c;
     printf("Введите значение a: ");
     scanf("%lf", &a);
     printf("Введите значение b: ");
     scanf("%lf", &b);
     c = mlt(a, b);
```

```
    printf("a * b = %4.2f", c);
    return 0;
}
```

В приведенном примере в функции main:

- объявляются три действительные переменные;
- вводятся значения переменных a и b;
- вызывается функция mlt;
- выводится результат;
- возвращается нормальный код завершения.

1.4. Переменная, блок

Переменная

Напомним, что переменная это именованная величина, значение которой может меняться. Более точно, каждая переменная имеет четыре характеристики:

- имя (идентификатор);
- тип;
- адрес;
- значение.

Идентификатор переменной в языке С может содержать:

- строчные и прописные символы латиницы;
- цифры;
- символ «подчерк» — «_»;
- начинается не с цифры.

Внимание! В отличие от языка Pascal в языке С строчные и прописные символы различаются. Т.е. MyName и myname в языке С два разных идентификатора.

Тип переменной определяет множество значений, которое может принимать переменная и операций, которые можно выполнять с переменной.

Объявления переменных

Переменные в языке С объявляются с помощью предложения:

<тип_переменной> <список_идентификаторов>;

Где:

- тип_переменной – имя типа объявляемой переменной;
- список_идентификаторов – список идентификаторов обявляемых переменных через запятую.

Примеры:

```
int a, gamma;
double betta;
```

В приведенном примере объявлены две переменные типа int (целое) и одна переменная типа double (действительное).

При объявлении переменные могут быть инициированы начальными значениями.

Пример:

```
int ht = 3, adt, gem = -6;
double alfa = 3.98;
```

В приведенном примере объявлены три переменные типа int (целое), две из которых инициированы начальными значениями и одна переменная типа double (действительное), которая инициирована начальным значением.

Следует иметь в виду, что инициализация при объявлении переменной:

```
int ht = 3;
```

и присвоение значения после объявления:

```
int ht;
ht = 3;
```

отличаются тем, что инициализация выполняется на шаге компиляции, а присвоение значения на шаге выполнения программы (т.е. требует дополнительного времени и откомпилированного кода).

Блок

В языке С блок это группа операторов в фигурных скобках, составляющее некоторое целое (тело цикла, варианты у if, тело функции).

Примеры:

```
{
      a = b + c;
      // ...
}

if ( a > b )
{
      a = a + 1;
      b = b - 1;
}
else
{
      a = a - 1;
      b = b + 1;
}
```

Блок языка С является некоторым аналогом конструкции begin – end языка Pascal. Существенным отличием является то, что в блоке могут быть объявлены (и инициированы) переменные:

```
{
      int c = a + b;
      // ...
}
```

Объявленные в блоке переменные являются локальными — они «видны» (и существуют) только внутри блока. Блоки могут вкладываться, а объявленные переменные — перекрываться:

```
{
      int a = 3;
      double b = 2.0;
      {
            int a, b; // Эти a и b перекрывают внешние
            a = 5;    // присваивание локальной a
      }
      // Здесь опять внешние a и b, a = 3, b = 2.0
}
```

1.5. Форматный ввод / вывод

В языке С нет встроенных средств (операторов) ввода/вывода данных. Ввод/вывод в языке С выполняется с помощью специальных функций,

входящих в библиотеку функций ввода/вывода. Для подключения этой библиотеки надо в начало исходного кода вашей программы добавить команду:

```
#include <stdio.h>  // Библиотека стандартного ввода/вывода
```

Форматный вывод

Форматный вывод на консоль в языке С выполняется с помощью функции printf, вызов которой имеет вид:

```
printf(<Формат>, <Список_вывода>);
```

где:

<Формат> - форматная строка;

<Список_вывода> - список элементов вывода.

Формат преобразования вывода представляет из себя текст с включенными в него спецификаторами вывода. Текст формата выводится на консоль, а спецификаторы используются для форматных преобразований соответствующего спецификатору элементу списка вывода. Каждому спецификатору должен соответствовать свой элемент списка вывода.

Для вывода целых (int) используется спецификатор %d, для вывода действительных (double) – спецификатор %f.

Пример.

```
int i = 35;
double d = 3.14;
printf("i = %d, d = %f", i, d);
```

В этом примере на консоль выводится:

```
i = 35, d = 3.14
```

Подробнее: Б. Керниган, Д. Ритчи. Язык программирования Си. Форматный вывод (printf). http://cpp.com.ru/kr_cbook/ch7kr.html#p72

Форматный ввод

Форматный ввод с клавиатуры в языке С выполняется с помощью функции scanf, вызов которой имеет вид:

```
scanf(<Формат>, <Список_ввода>);
```

где:

<Формат> - форматная строка;

<Список_вывода> - список элементов ввода.

Формат преобразования ввода представляет из себя текстовую строку с включенными в нее спецификаторами ввода. Каждому спецификатору должен соответствовать свой элемент списка ввода.

Для ввода целых (int) используется спецификатор %d, для ввода действительных (double) – спецификатор %lf. Если спецификаторы ввода записаны через пробел, то вводимые значения надо разделять пробелом.

Элементом списка ввода может быть адрес вводимой переменной или указатель на вводимую переменную.

Пример.

```
int i ;
double d4;
printf("Введите значения переменных i и d: ");
scanf("%d %lf", &i, &d);
```

В приведенном примере на экран монитора будет выведен текст:

```
Введите значения переменных i и d:
```

в конце которого надо будет через пробел ввести целое и действительное числа.

2. БАЗОВЫЕ ТИПЫ ДАННЫХ

2.1. Целочисленные типы

2.1.1. Целочисленные типы C

В соответствии со стандартом C есть четыре знаковых целочисленных типа:

```
signed char,
short int,
int,
long int
```

и четыре беззнаковых:

```
unsigned char,
unsigned short int,
unsigned int,
unsigned long int
```

Понимать можно так: есть два целочисленных типа: `char` и `int`, которые могут использоваться с модификаторами знаковости и размера:

[`unsigned` / `signed`] [`short` / `long`] `int`

[`unsigned` / `signed`] `char`

При этом, тип по умолчанию `int` , модификатор по умолчанию - `signed` и приведенные ниже записи эквивалентны:

< модификатор > = < модификатор > `int`

`short` = `short int`

`long` = `long int`

`unsigned` = `unsigned int`

`unsigned long` = `unsigned long int`

`signed` < тип > = < тип >

`signed char` = `char`

`signed int` = `int`

`signed long` = `long` = `long int` = `signed long int`

Во избежание сложности объявления типа знаковые целые типы можно объявлять как: `char`, `short`, `int` и `long`; беззнаковые как `unsigned char`, `unsigned short`, `unsigned int` и `unsigned long`.

2.1.2. Размеры целочисленных типов

Стандарт языка С

В стандарте языка размер (количество байт) целочисленных типов не определен, т.к. конкретные размеры типов определяются в реализациях языка (компиляторах и аппаратных платформах). Размер типов и переменных типа может но может быть вычислен с помощью операции `sizeof`:

- `sizeof`(<имя_типа>) – размер типа <имя_типа>;
- `sizeof`(<имя_переменной>) – размер типа переменной <имя_переменной>.

Пример:

```
int ucLeng = sizeof(unsigned char); // Размер типа unsigned char
long a;
int aLeng = sizeof(a);         // Размер переменной a типа long
```

В стандарте языка С определены соотношения между размерами:

1 = `sizeof`(`char`) <= `sizeof`(`short`) <= `sizeof`(`int`) <= `sizeof`(`long`)

`sizeof`(<тип>) = `sizeof`(`signed` (<тип>) = `sizeof`(`unsigned` (<тип>)

Реализации языка

Как отмечалось выше, размеры целочисленных данных в разных реализациях языка могут быть разными. В таблицах ниже приведены данные по размерам и диапазонам значений целочисленных данных в нескольких реализациях языка фирмы Borland и MS Visual C 2005.

Тип	Размер, байт		
	Borland C++ 3.1	Borland C++ Builder 5.0	GCC (IPF, 64 бит)
`char`	1	1	1
`int`	2	4	4
`long`	4	4	8

Таблица 2.1. Размеры целочисленных данных реализаций языка C фирмы Borland.

Тип	Байт	Диапазон значений	
		`signed`	`unsigned`
`char`	1	[-128; 127]	[0; 255]
`short`	2	[-32 768; 32 767]	[0; 65 535]
`int`	4	[-2 147 483 648; 2 147 483 647]	[0; 4 294 967 295]
`long`	4	[-2 147 483 648; 2 147 483 647]	[0; 4 294 967 295]

Таблица 2.2. Размеры и диапазоны значений MS Visual C 2005.

2.1.3. Представления целочисленных типов

Беззнаковые целочисленные (`unsigned`) представляются в прямом коде:

$23_{10} = 17_{16} = 0001\ 0111_2$

$255_{10} = FF_{16} = 1111\ 1111_2$

$128_{10} = 80_{16} = 1000\ 0000_2$

Знаковые целочисленные (`signed`) представляются: положительные в прямом коде, отрицательные – в дополнительном коде. Дополнительный код числа (код, дополняющий до нуля) это код, сумма которого с положительным значением числа дает ноль:

$-23_{10} = E9_{16} = 1110\ 1001_2 + 00010111_2 = 1\ 0000\ 0000_2$

Соответственно:

$127_{10} = 7F_{16} = 0111\ 1111_2$

$-1_{10} = FF_{16} = 1111\ 1111_2 + 0000\ 0001_2 = 1\ 0000\ 0000_2$

$-128_{10} = 80_{16} = 1000\ 0000_2 + 1000\ 0000_2 = 1\ 0000\ 0000_2$

2.1.4. Целочисленные константы

Целочисленные константы могут быть десятичными, восьмеричными или шестнадцатеричными.

Десятичные константы записываются в виде: `[-]DDD`, где `DDD` – десятичные цифры константы.

Пример: 123, -894567.

Восьмеричные константы записываются в виде: `[-]0BBB`, где `BBB` – восьмеричные цифры константы.

Пример: 0123, -04567.

Шестнадцатеричные константы записываются в виде: `[-]0xHHH` (или `[-]0XHHH`), где `HHH` – шестнадцатеричные цифры константы.

Пример: 0xAD00, -0XFE9.

При записи целочисленных констант можно указывать спецификаторы беззнаковости (U или u) и длины (L или l). Спецификаторы указываются в конце записи констант.

Примеры:

`23L` – десятичная константа типа `signed long`

`023ul` – восьмеричная константа типа `unsigned long`

`0xDEu` – восьмеричная константа типа `unsigned`

Если при записи константы спецификаторы беззнаковости и длины не указаны, то тип константы будет определяться по умолчанию:

- для десятичных целочисленных констант: первый подходящий из `signed int` и `signed long`;
- для восьмеричных и шестнадцатеричных целочисленных констант: первый подходящий из `int` и `unsigned int`, `long`, `unsigned long`.

Спецификатор константы необходимо указывать для того, чтобы избежать лишних операций по неявному приведению типов.

Пример.

```
unsigned long a, b = 2452789;
a = 2 * b;
```

При вычислении выражения a = 2 * b; в приведенном выше фрагменте кода константа 2 (`signed int`) будет сначала неявно приведена к `unsigned long` и только после этого будет выполнено умножение. Этого приведения можно было бы избежать, если при записи константы указать спецификаторы беззнаковости и длины.

2.2. Действительные типы

Действительные типы C

В стандарте языка C определены три числовых действительных типа: `float`, `double` и `long double`.

Определены соотношения между размерами (и точностями):

`sizeof(float)` <= `sizeof(double)` <= `sizeof(long double)`

Размеры действительных типов, значность и диапазоны значений определяются конкретными реализациями. В таблице 2.3 приведены значения этих параметров для MS Visual Studio 2005.

Тип	Байт	Значность	Диапазон положительных значений
`float`	4	6	$[1{,}1755 \cdot 10^{-38}; 3{,}4028 \cdot 10^{+38}]$
`double`	8	15	$[2{,}2250 \cdot 10^{-308}; 1{,}7977 \cdot 10^{+308}]$
`long double`	8	15	$[2{,}2250 \cdot 10^{-308}; 1{,}7977 \cdot 10^{+308}]$

Таблица 2.3. Размеры, значность и диапазоны значений действительных типов в MS Visual Studio 2005.

Действительные константы

Общий вид записи действительной константы: `[-][nn][.][mmm][E/e[+/-]pp]` , где:

- `nn` – целая часть константы;

- `mmm` – дробная часть константы;
- `E/e` – признак порядка константы;
- `pp` – значение порядка константы.

Примеры: `23.56, -3.45e-6`

При записи действительных констант действуют правила:

- `nn` или `mmm` (но не обе сразу) можно опустить: `1.  .23`
- десятичную точку или признак порядка (но не обе сразу) можно опустить: `1e0  245e-7`

По умолчанию десятичная действительная константа имеет первый подходящий тип из `double` и `long double`. Для явного указания того, что действительная константа должна иметь тип можно использовать спецификатор длины L или l.

Пример. `2.25e2L` – десятичная константа типа `long double`.

2.3. Типы данных и переменные

2.3.1. Объявления переменных

Объявления переменных в языке С имеет вид:

<Тип_переменной> <Список_идентификаторов>;

При объявлении переменных действуют следующие правила:

- идентификаторы объявляемых переменных перечисляются в списке через запятую;
- идентификатор переменной может содержать латинские буквы (строчные и прописные), цифры и знак подчеркивания;
- идентификатор не должен начинаться с цифры;
- длина идентификатора определяется реализацией компилятора;
- при объявлении переменные могут быть инициированы начальными значениями (см. примеры ниже).

Примеры.

```
char c1, c2, c3;
```

Объявление трех переменных типа `char`.

```
int i = 18;
```

Объявление переменной типа `int` и инициализаця начальным значением 18.

```
unsigned u = 045u, u1 = 0xADAu;
```

Объявление двух переменных типа `unsigned` и инициализаця их начальными значениями, заданными в виде восьмеричной и шестнадцатеричной констант.

```
long a = 5672L;
```

Объявление переменной типа `long` и инициализаця начальным значением со спецификатором длины.

```
unsigned long ul = 0XFFFUL;
```

Объявление переменной типа `unsigned long` и инициализаця начальным значением шестнадцатеричной константы со спецификаторами беззнаковости и длины.

```
char c = -127, c1 = 's';
```

Переменные типа `char` могут инициализироваться целочисленной константой или символьной константой, заключенной в одинарные кавычки.

2.3.2. Объявление синонима типа

В языке С можно объялять новое имя типа. Синтаксис объявления нового имени типа:

```
typedef <имя_старого_типа> <имя_нового_типа>;
```

С помощью приведенного выражения создается синоним <имя_нового_типа>для типа <имя_старого_типа>. Старый и новый тип равнозначны.

Примеры:

```
typedef unsigned int uint32;
typedef unsigned short uint16;
typedef unsigned char uint8;
```

Типичным применением объявления синонимов типов являются:

- введение сокращенного имени типа;
- сведение зависимостей в одну точку.

Под сведением зависимостей в одну точку понимается ситуация, когда возникает необходмость переноса кода на компьютер с менее «емкими» типами данных. В этом случае будет достаточно переопределить типы тольк в одном месте кода (в одной точке):

```
typedef unsigned long uint32;
typedef unsigned int uint16;
```

2.3.3. Некоторые особенности работы с переменными типа char

Тип данных `char` в языке С (в отличие от Pascal) является не символьным, а целочисленным. Переменные этого типа могут быть операгдами в арифметических выражениях и одинаково «охотно» работают как с целочисленными, так и с символьными данными:

- Переменные типа `char` могут инициализироваться как целочисленной константой, так и символьной константой, заключенной в одинарные кавычки.

```
char c = -127, c1 = 's';
```

- Переменным типа `char` могут присваиваться как целочисленные, так и символьные значения.

```
char c1, c2;
c1 = 'a';      // c1 = 'a' (код = 97)
c2 = 98;       // i1 = 'b'
```

- Переменным типа `char` могут присваиваться целочисленные выражения, а целочисленным переменным - символьные.

```
char c1, c2, c3;
int  i1, i2, i3;
c1 = 'a';      // c1 = 'a'
i1 = c1;       // i1 = 97
i2 = 'b';      // i2 = 98
c2 = c1 + 1;   // c1 = 'b'
c3 = i1 + 1;   // c3 = 'b'
i3 = 'a' + 1; // i3 = 98
```

- При выводе в поток переменных типа `char` выводится символьное, а не целочисленное значение.

```
char c1;
int  i1 = 30;
c1 = 3 * i1 + 7;
cout << "c1 = " << c1; // Результат: c1 = a
```

2.4. Операции и выражения

2.4.1. Перечень операций языка С

В языке С все операции разделены на 15 приоритетных групп. Такое «богатство» приоритетов позволяет не писать лишних скобок при записи выражений.

В таблице 2.4 приведен полный перечень операций языка С, разделенных на группы приоритетов.

Приоритет	Операция	Т	Название	Пример
1	->		косвенный выбор члена структуры	указатель->член_структуры
	.		прямой выбор члена структуры	структура.член_структуры
	[ ]		индексация	int array[30];
	( )		вызов функции	выражение(список_аргументов)
2	++	i	приращение на 1	++a; b++;
	--	i	уменьшение на 1	--aa b--;
	~	i	поразрядное не	char c1=0xF0, c2=~c1; // c1=0X0F
	!	i	логическое не	c1 = !c2; // c1 = 0 (0x00)
	-		унарный минус	i = -a * b / c;
	&		адрес объекта	int a = 3; int* p = &a;
	*	p	разыменование	int b = *p;
	sizeof		размер объекта	int s = sizeof(d * 3.14); // s=8
	(type)		приведение типа	int a =(int)3.14*8.29; // a = 24
3	*		умножение	выр * выр
	/		деление	выр / выр
	%	i	взятие по модулю (остаток)	выр % выр
4	+		сложение (плюс)	выр + выр
	-		вычитание (минус)	выр - выр
5	<<	i	сдвиг влево	int j, k, i = 4; k = i << 2; // 16
	>>	i	сдвиг вправо	j = k >> 2; // 1
6	<		меньше	i = f + 7 < d * p; // 0 или 1
	<=		меньше или равно	выр <= выр
	>		больше	выр > выр
	>=		больше или равно	выр >= выр

7	==		равно	выр == выр
	!=		не равно	выр != выр
8	&	i	побитовое И	unsigned i, j = 0535, k = 01234; i = k & 07 << 6; // i = 0200
9	^	i	побитовое исключающее ИЛИ	i = k ^ 07 << 3; // i = 1244
10	|	i	побитовое включающее ИЛИ	i = k | j; // i = 1735
11	&&		логическое И	i = a+3 && b/7. // 0 или 1
12	||		логическое включающее ИЛИ	выр || выр

13	? :		арифметический if	выр ? выр : выр
				i = a < b ? a + b : a - b;
				j = p / a < b ? a : b;

14	=		простое присваивание	a = b;
	*=		умножить и присвоить	a *= b; // a = a * b;
	/=		разделить и присвоить	a /= b; // a = a / b;
	%=	i	взять по модулю и присвоить	a %= b; // a = a % b;
	+=		сложить и присвоить	a += b; // a = a + b;
	-=		вычесть и присвоить	a *= b; // a = a - b;
	<<=	i	сдвинуть влево и присвоить	a <<= k; // a = a << k;
	>>=	i	сдвинуть вправо и присвоить	a >>= k; // a = a >> k;
	&=	i	И и присвоить	a &= b; // a = a & b;
	|=	i	включающее ИЛИ и присвоить	a |= b; // a = a | b;
	^=	i	исключающее ИЛИ и присвоить	a ^= b; // a = a ^ b;
15	,		запятая (последование)	выр , выр

Таблица 2.4. Операции языка С.

2.4.2. Выражения

При вычислении выражений применяется обычное правило приоритетов: сначала вычисляются выражения в скобках, затем вычисляются функции, затем операции в соответствии с приоритетами.

В языке С действует правило ассоциативности операций. Все операции делятся на правоассоциативные (выполняются слева направо) и левоассоциативные (выполняются спава налево). В таблице 2.5 приведена сводка операций языка С с указанием их приоритетов и ассоциативности.

Приор.	Операции	Ассоц.
1	`() [] -> .`	прав
2	`~ ++ -- ! - (type) * & sizeof`	лев
3	`* / %`	прав
4	`+ -`	прав
5	`<< >>`	прав
6	`< <= > >=`	прав
7	`== !=`	прав
8	`&`	прав
9	`^`	прав
10	`|`	прав
11	`&&`	прав
12	`||`	прав
13	`?:`	лев
14	`= += ...`	лев
15	`,`	прав

Таблица 2.5. Ассоциативность операций языка С.

2.4.3. Особенности выполнения некоторых операций

Операция присваивания

В языке С присваивание не оператор, а операция, результатом выполнения которой является присваиваемое значение.

Примеры:

```
int a, b, c, d;
```

```
a = b = c = 3;
```

В приведенном примере переменная c получит значение 3, затем результат этой операции (3) будет присвоен переменной b, затем результат будет присвоен переменной a. Напомним, что операция присваивания левоассоциативна.

Операция присваивания может участвовать и в более сложных выражениях:

```
d = 4 * (c = 8);
```

Операции отношения и логические операции

Операции отношения (== != < > <= >=) и логические операции (&& || ~) возвращают целочисленный 0 или 1 и могут участвовать в арифметическом выражении.

Пример. Выражение:

```
a = ( b == 7 )*d + ( c != 3 )*f;
```

эквивалентно следующему:

```
if ( b == 7 && c != 3 ) a = d + f;
else
{
  if ( b == 7 ) a = d;
  if ( c != 3 ) a = f;
}
```

Арифметический if

Синтаксис

<условие> ? <значение1> : <значение2>

Если значение <условия> истинно (не равно 0), то результат операции — <значение1>, иначе — <значение2>. <значение1> и <значение2> должны быть одного типа.

Пример:

```
int sign = s > 0 ? 1 : (s == 0 ? 0 : -1);
```

В приведенном примере переменная sign получит значение 1 при s > 0, 0 при s = 0 и -1 при s < 0.

Операция «запятая»

В литературе в отношении операции «запятая» говорится следующее: «…несколько выражений, разделенных запятыми, вычисляются последовательно слева направо. В качестве результата сохраняются тип и значение самого правого выражения. Таким образом, операция "запятая" группирует вычисления слева направо. Тип и значение результата определяются самым правым из разделенных запятыми операндов (выражений). Значения всех левых операндов игнорируются.» (Подбельский В.В. Язык СИ++).

Сказанное можно проиллюстрировать следующим примером:

```
int a = 1, b = 1, c = 1, e = 1, d = 2;
a = (b = d + 1), (c = b + 2), (e = c + 3);
```

Проверка этого примера в MS VisualStudio 2010 показала, что в результате выполнения этого фрагмента кода переменная `b` получает значение 3, переменная `c` – значение 5, переменная `e` – значение 8, что говорит о том, что разделенные запятой выражения выполняются слева направо. Но переменная `a` получает значение 3, а не 8.

Операция «запятая» достаточно экзотична и используется редко. Исключение составляет программирование систем реального времени, где она используется для последовательного опроса регистров. Кроме того, она используется в цикле `for` (см. лекцию 3).

2.5. Преобразования типов

Неявные преобразования типов

При вычислении выражений сначала все операнды приводятся к одному («старшему») типу и лишь потом вычисляется выражение.

Пример.

```
char        c  = 2;
short       s  = 3;
int         i  = 4;
long        l  = 5;
float       f  = 6.;
double      d  = 7.;
long double ld = 8.;
int r;
r = c + s + i + l + f + d + ld;
```

В приведенном примере все операнды будут приведены к `long double`, затем будет вычислено выражение, после чего результат (типа `long double`) будет приведен к `int`.

Описанные выше преобразования типов выполняются компилятором неявно (автоматически). Неявные преобразования типов можно разделить на две группы: безопасные и опасные. К безопасным относятся те преобразования, которые не приводят к потере точности или значения результата. Это преобразования целого в действительное или преобразование в более «емкий» тип. Приведенные ниже цепочки преобразования являются безопасными:

целое -> `float` -> `double` -> `long double`

`char` -> `short` -> `int` -> `unsigned` -> `long` -> `unsigned long`

К опасным относятся те преобразования, которые могут привести к потере точности или значения результата. Это преобразования действительного в целое (потеря дробной части) или преобразование в менее «емкий» тип (потеря значения). Приведенные ниже примеры показывают потерю значения при преобразовании в менее «емкий» тип.

```
short s = 0x123456789ABul; // s = 0x89AB
double d = 0.7e3 * 0.11e4; // d = 770000.00
unsigned short  i = d;     // i = 49104
```

Явные преобразования типов

При необходимости в выражениях можно указывать явное преобразование типов.

В языке С явное преобразование типов указывается в виде: (<имя_типа>)<выражение>.

Пример:

```
int i, j, k = 3;
double a = 3.8;
i = k * (2 * a);
j = k * (int)(2 * a);
```

В приведенном примере переменная `i` получит значение 22, а переменная `j` значение 21.

2.6. Именованные константы

Именованные константы и макросы

В языке С именованные константы могут объявляться в начале блока. Именованные константы объявляются с указанием префикса `const` с обязательной инициализацией:

```
const <имя типа> <имя_константы> = <значение>;
```

Пример:

```
const double PI = 3.14;
double r, ld, s;
r = 8.54;
ld = 2. * PI * r;
s  = PI * r * r;
```

Очень похожий результат можно получить с помощью директивы препроцессора `#define` (объявление макроса):

```
#define <Имя> <Текст>
```

Эта директива работает так: препроцессор просматривает код и все вхождения <Имя> заменяет на <Текст>, после чего выполняется компиляция кода

Приведенный выше пример с использованием директивы `#define` будет выглядеть так:

```
#define PI 3.14
double r, ld, s;
r = 8.54;
ld = 2. * PI * r;
s  = PI * r * r;
```

Стандартные библиотеки макросов

В языке С есть стандартные библиотеки макросов:

1. Библиотека `limits.h`:

- `SHRT_MAX` – максимальное значение данных типа `short`;
- `SHRT_MIN` – минимальное значение данных типа `short`;
- `INT_MAX` – максимальное значение данных типа `int`;
- `INT_MIN` – минимальное значение данных типа `int`;
- `LONG_MAX` – максимальное значение данных типа `long`.
- `LONG_MIN` – минимальное значение данных типа `long`.

2. Библиотека `float.h`:

- `FLT_MAX` – максимальное значение данных типа `float`;
- `FLT_MIN` – минимальное значение данных типа `float`;

- FLT_EPSILON – точность представления данных типа float;
- DBL_MAX – максимальное значение данных типа double;
- DBL_MIN – минимальное значение данных типа double;
- DBL_EPSILON – точность представления данных типа double.

2.7. Другие фундаментальные типы данных

2.7.1. Использование булевских величин

В языке С нет встроенного типа данных для представления булевских величин. Но в языке С есть стандартная библиотека stdbool.h, в которой объявлены следующие макросы:

```
#define bool unsigned
#define false 0
#define true 1
```

2.7.2. Тип void

Тип void (отсутствие типа) – специальный тип, не имеющий значений и размера. Переменные типа void объявлять нельзя. Тип void может использоваться в следующих случаях:

1. Объявление функции, которая не возвращает значения:

```
void f(…);
```

2. Объявление указателя на void (так называемого родового указателя), которому может быть присвоено значение указателя на любой тип и который можно явно приводить к указателю на любой тип.

2.7.3. Тип enum

Синтаксис объявления типов данных enum (перечисленных):

```
enum [<имя_типа>] {<значение_1>,... ,<значение_n>} [<пер_1, пер_2, ...>];
```

Такое объявление создает новый тип данных с именем <имя_типа> , значениями которого являются <значение_1>,... ,<значение_n>.

Пример:

```
enum DayOfWeek {Sunday, Monday,..., Saturday};
DayOfWeek today = Sunday;
today = Monday;
```

В приведенном примере объявлен тип данных DayOfWeek, принимающий значения Sunday, Monday,..., Saturday. Объявлена переменная today типа DayOfWeek, которой присваиваются значения этого типа.

Объявление типов данных enum эквивалентно объявлению констант типа int:

```
const int Sunday   = 0;
const int Monday   = 1;
...
const int Saturday = 6;
```

Откуда следует, что значения типов данных могут участвовать в арифметических выражениях:

```
int k = Sunday * 3 + Saturday;
```

Объявление типа данных enum может включать объявления переменных этого типа:

```
enum DayOfWeek {Sunday, Monday,..., Saturday} today1, today2;
```

Объявлять переменные типа enum можно без объявления имени типа данных:

```
enum {Up, Down} direction;
```

При объявлении типа данных enum можно задавать числовые эквиваленты значений:

```
enum Boys {John, Paul, George = 8, Ringo};
```

В этом случае:

```
const int John = 0;
const int Paul = 1;
const int George = 8;
const int Ringo = 9;
```

Преобразование значений типов данных enum к целочисленным значениям выполняется компилятором неявно. Обратного неявного преобразования нет, но явное обратное преобразование возможно:

```
Boys Pit2 = (Boys)1;
Boys Pit3 = Boys(8);
Boys Pit4 = static_cast<Boys>(9);
```

При выполнении приведенных преобразований переменная `Pit2` получит значение `Paul`, переменная `Pit3` получит значение George, переменная `Pit4` получит значение Ringo.

3. ВЫЧИСЛИТЕЛЬНЫЕ СТРУКТУРЫ

3.1. Оператор ветвления if

Общие правила

Синтаксис:

```
if(<условие>) <S1> [else <S2>]
```

где: `<S1>`, `<S2>` — выражение или группы выражений, заключенные в фигурные скобки (блоки). Правило выполнения оператора обычное: если `<условие>` истинно (см. ниже), то выполняется `<S1>`, иначе `<S2>`. Часть `else` может быть опущена, если не нужна.

Примеры:

```
if ( a > b )
    max = a;
else
    max = b;
```

В приведенном примере если a > b, то переменной max присвоится значение переменной a или значение переменной b в противном случае.

```
if ( a < 0 )
    a = -a;
```

В приведенном примере если a < 0, то переменная a изменит знак на противоположный.

```
if ( a != b ) a = a + 1; b = b - 1;
```

В приведенном примере значение переменной b всегда будет уменьшаться на единицу. Хотя такая форма записи является синтаксически корректной, она «провоцирует» неправильное понимание кода. Видимо правильнее было бы записать так:

```
if ( a != b ) { a = a - 1; b = b + 1; }
```

```
if ( a != b )
{
    a = a + 1;
    b = b - 1;
}
else
```

```
{
    a = a - 1;
    b = b + 1;
}
```

Надеюсь, что этот пример в комментариях не нуждается.

```
if ( a < b )
    a = a + 1;
    b = b - 1;
```

Это еще одна «провокация» в записи кода. Скорее всего, программист имел в виду так:

```
if ( a < b )
    a = a + 1;
b = b - 1;
```

Хорошим стилем считается обязательное использование фигурных скобок, даже если в них стоит одно выражение:

```
if ( a < b )
{
    a = a + 1;
}
b = b - 1;
```

Поговорим об условии

Особенностью языка C является то, что в качестве условия может использоваться выражение любого типа, приводимого к целочисленному. Истинность условия определяется так:

1. Вычисляется выражение условия.
2. Результат приводится к целому типу.
3. Если в приведенном результате есть хотя бы один ненулевой бит, то условие считается истинным, если все биты нулевые – ложным.

Примеры.

```
if ( a != 0 ) ...
if ( a > 0 & a < 10 ) ...
```

Надеюсь, что эти примеры в комментариях не нуждаются.

```
if ( a ) ...
if ( a = b*(d - 8.0) ) ...
```

Это менее очевидно, но все правильно. В первом случае значение переменной a будет приведено к целому и если есть хоть один ненулевой бит,

… Во втором случае результатом выражения будет значение, присвоенное переменной a, которое будет приведено к целому …

```
if ( a = 0 ) ...
if ( k = 17 ) ...
```

Синтаксически все корректно, но видимо, ошибка, т.к. в первом случае условие всегда ложно, а во втором случае всегда истинно. Скорее всего, имелось в виду следующее:

```
if ( a == 0 ) ...
if ( k == 17 ) ...
```

Вложенные if

Как обычно, конструкции `if – else` могут вкладываться друг в друга:

```
if ( a != b )
{
    if ( a < b )
        a = a + 1;
    else
        b = b - 1;
    c = a + b;
}
else
{
    c = a – b;
    if ( c < a )
        a = a - 1;
    else
        b = b + 1;
}
```

Менее очевидной является следующая конструкция:

```
int a, b, c;
a = 1;
b = 2;
c = 3;
if ( a > b )
    if ( b > c )
        c = c + 2;
else
    b = b + 2;
```

Здесь значение переменной b останется 2, т.к. несмотря на «провокационную» запись `else` относится ко второму, а не к первому .`if`

И вообще действует правило:

При вложенных `if`, `if-else` каждый `else` соответствует ближайшему «необ`else`еному» `if`у.

Примеры

Пример 1. Ниже приведен пример фрагмента кода решения квадратного уравнения с коэффициентами a, b и c.

```
double a, b, c;
double d, x1, x2, re, im;
d = b * b - 4.0 * a * c;
if ( d > 0.0 )
{
    d = sqrt(d);
    x1 = (-b + d) / (2.0 * a);
    x2 = (-b - d) / (2.0 * a);
}
else if ( d < 0.0 )
{
    d = sqrt(-d);
    re = -b / (2.0 * a);
    im =  d / (2.0 * a);
}
else
    x1 = x2 = -b / (2.0 * a);
```

Пример 2. Написать программу, которая определяет сумму страхового взноса при страховании детей при условии, что:

- страховой взнос определяется как процент от суммы страховки;
- при страховании девочек до 5 лет этот процент равен 4%, а для девочек старше 5 лет при сумме страховки до 15 000 он равен 5%, при сумме свыше 15 000 — 3%;
- при страховании мальчиков до 8 лет страховой процент равен 10%, а для мальчиков старше 8 лет он составляет 1,5% при сумме страховки до 20 000 и 2,5% при сумме страховки свыше 20 000.

Решение поставленной задачи может выглядеть так:

```
int age;int sum;int sex;double st_sum;double pro;cout  << "Пол
ребенка: ";cin >> sex;cout  << "Возраст ребенка: ";
cin>>age;cout<<"Сумма страховки: ";cin >> sum;if ( sex == 0 )if (
age <= 5 )pro = 0.04;else if (sum<15000)pro = 0.05;else
pro=0.03;else if(age>8)if ( sum > 20000 )pro = 0.015;else pro =
0.025;else pro = 0.1;st_sum = sum * pro;
cout<<"Вы должны заплатить "<<st_sum <<" рублей"<< endl;
```

Это решение синтаксически корректно, но совершенно не читаемо. Читаемый и понятный вариант выглядит так:

```
// Объявление переменных ---------------------------------------
// Исходные данные: ----------------------
int age;   // Возраст ребенка
int sum;   // Сумма страховки
int sex;   // Пол ребенка: 0 - девочка, 1 - мальчик
// Требуемый результат: ------------------
double st_sum; // Сумма страховки
// Рабочие переменные --------------------
double pro;    // Страховой процент

// Ввод исходных данных ----------------------------------------
cout  << "Пол ребенка: ";
cin >> sex;
cout  << "Возраст ребенка: ";
cin >> age;
cout  << "Сумма страховки: ";
cin >> sum;

// Выполнение вычислений ---------------------------------------
if ( sex == 0 )
    if ( age <= 5 )
        pro = 0.04;
    else
        if ( sum < 15000  )
            pro = 0.05;
        else
            pro = 0.03;
else
    if ( age > 8 )
        if ( sum > 20000 )
            pro = 0.015;
        else
            pro = 0.025;
    else
        pro = 0.1;

st_sum = sum * pro;

// Вывод результатов -------------------------------------------

cout << "Вы должны заплатить " << st_sum << " рублей" << endl;
```

3.2. Переключатель switch

Синтаксис:

```
switch(<выражение>)
{
     case <константное выражение1>: <S1>; [break;]
     case <константное выражение2>: <S2>; [break;]
     ...
     [default: <Sdef>]; [break;]
}
```

<S1>, <S2>, . . ., <Sdef> — выражения или группы выражений (не в фигурных скобках). Выполняются все <Si>, начиная с того, <константное выражение> которого совпало со значением <выражения>. Выполняются до ближайшего `break` (или до конца тела `switch`). Если ничего не подошло, то выполняется <Sdef> (при наличии `default`)

Примеры

Пример 1.

```
switch( n )
{
    case 1: cout << "Понедельник";
    case 2: cout << "Вторник";
    case 3: cout << "Среда";
    case 4: cout << "Четверг";
    case 5: cout << "Пятница";
    case 6: cout << "Суббота";
    case 7: cout << "Воскресенье";
    default: cout << "Неверный номер";
}
```

В этом примере при n = 1 на консоль будет выведено:

```
ПонедельникВторникСредаЧетвергПятницаСубботаВоскресеньеНеверный
номер
```

Название дня недели по номеру дня будет выдавать такой код:

```
switch( n )
{
    case 1: cout << "Понедельник"; break;
    case 2: cout << "Вторник";     break;
    case 3: cout << "Среда";       break;
    case 4: cout << "Четверг";     break;
    case 5: cout << "Пятница";     break;
    case 6: cout << "Суббота";     break;
    case 7: cout << "Воскресенье"; break;
    default: cout << "Неверный номер";
```

```
}
```

Пример 2. Выдать на консоль количество дней в месяце по его номеру:

```
int month;
cout << "Введите номер месяца: ";
cin >> month;

switch(month)
{
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        cout << "31" << endl;
        break;
    case 4: case 6: case 9: case 11:
        cout << "30" << endl;
        break;
    case 2:
        cout << "28" << endl;
        break;
    default:
        cout << "Неверный номер месяца" << endl;
}
```

3.3. Цикл с предусловием while

Общее описание

Синтаксис:

```
while (<условие>) <S>
```

где <S> - тело цикла - выражение или группа выражений, заключенная в фигурные скобки (блок) — тело цикла. Цикл выполняется пока <условие> истинно (см. подраздел «Поговорим об условии» раздела 3.1). Условие проверяется перед выполнением тела цикла.

Примеры

Задача 1. Написать функцию, подсчитывающую сумму цифр целого числа.

Вариант 1:

```
int sumOfDigits1(int n)
{
    int sum = 0;
    while ( n )
    {
        int c = n % 10;
        sum = sum + c;
        n = n / 10;
    }
```

```
    return sum;
}
```

Надеюсь, что вариант 1 в комментариях не нуждается. Вариант, использующий специфические операции языка С выглядит более компактно:

Вариант 2:

```
int sumOfDigits2(int n)
{
    int sum = 0;
    while ( n )
    {
        sum += n % 10;
        n /= 10;
    }
    return sum;
}
```

Еще более компактно выглядит следующий вариант:

Вариант 3:

```
int sumOfDigits3(int n)
{
    int sum = 0;
    n *= 10;
    while ( n /= 10 )
        sum += n % 10;
    return sum;
}
```

Совершенно экзотично (но корректно) выглядит следующий вариант:

Вариант 4:

```
int sumOfDigits4(int n)
{
    int sum = 0;
    n *= 10;
    while ( (n /= 10) && (sum += n % 10) );
    return sum;
}
```

Задача 2. Написать функцию, вычисляющую точность представления чисел типа `double`.

```
double getEpsilon()
{
    double epsilon = 1.0;
    while(epsilon / 2.0 + 1 > 1)
        epsilon /= 2.0;
    return epsilon;
}
```

3.4. Цикл с постусловием do-while

Общее описание

Синтаксис:

```
do <S> while(<условие>);
```

где <S> - тело цикла - выражение или группа выражений, заключенная в фигурные скобки (блок). Цикл выполняется пока <условие> истинно (см. подраздел «Поговорим об условии» раздела 3.1). Условие проверяется после выполнения тела цикла.

В отличие от цикла `while`, тело цикла `do – while` выполнится по крайней мере один раз

Примеры

Задача 1. Написать функцию, которая вводит число типа `double` с контролем принадлежности введенного значения диапазону [a1, a2].

```
double inputDouble(const char* comment, double a1, double a2)
{
    double d;
    do
    {
        cout << comment;
        cin >> d;
        if ( d < a1 )
            cout << "Значение должно быть >= " << a1 <<endl;
        if ( d > a1 )
            cout << "Значение должно быть <= " << a2 <<endl;
    } while ( (d < a1) || (d > a2) );
    return d;
}
```

Задача 2. Написать функцию, вычисляющую значение n!

Вариант 1:

```
int Factorial1(int n)
{
    int fac = 1;
    do
    {
        fac = fac * n;
        n = n - 1;
    } while ( n );
    return fac;
}
```

Надеюсь, что вариант 1 в комментариях не нуждается. Вариант, использующий специфические операции языка С выглядит более компактно:

Вариант 2:

```
int Factorial2(int n)
{
    int fac = 1;
    do
        fac *= n--;
    while ( n );
    return fac;
}
```

3.5. Цикл for

Общее описание

Синтаксис:

```
for([<выражение1>]; [<условие>]; [<выражение2>])<S>
```

где <S> - тело цикла - выражение или группа выражений, заключенная в фигурные скобки (блок).

Особенность цикла `for` языка С состоит в том, что цикл `for` эквивалентен конструкции:

```
[<выражение1>];
while ( <условие> )
{
    <S>;
    [<выражение2>];
}
```

Что и объясняет все особенности работы цикла `for`.

Примеры

Задача 1. Написать функцию, вычисляющую значение n!

```
int Factorial(int n)
{
    int fac = 1;
    for ( int i = 2; i <= n; i++ )
        fac *= i;
    return fac;
}
```

Задача 2. Суммирование квадратов первых n натуральных чисел может быть записано в любом из следующих вариантов:

```
int s = 0;
for( i = 1; i <= n; i++ ) s += i * i;
for( i = 0; i <= n; s += ++i * i );
for( i = 0; i <= n; ) s += ++i * i;
```

Задача 3. Написать функцию, выполняющую суммирование отрезка ряда: $1+a-a^2+a^3+\ldots+(-1)^{n-1}a^n$:

```
double setSegmentSum( double a, int n)
{
    double da = 1.0;
    double sum = 1.0;
    for ( int i = 1; i <= n; i++ )
    {
        da = da * a;
        if ( i % 2 )
            sum -= da;
        else
            sum += da;
    }
    return sum;
}
```

В этом примере есть две особенности:

- счетчик цикла `i` объявляется в заголовке цикла `for`. При таком варианте переменная `i` является локальной переменной блока цикла;
- введена рабочая переменная `da`, в которой «накапливается» степень `a`. Такой прием позволяет избежать использования операции возведения в степень.

Задача 4. Написать функцию, определяющую, является ли фраза палиндромом (читается одинаково слева направо и справа налево).

Вариант 1:

```
bool isPolindrom1( char str[] )
{
    DelSymbol(str, ' ');
    int n = StrLen(str);
    int i, j = n - 1;

    bool res = true;
    for ( i = 0; i < n; i++ )
    {
      if ( str[i] != str[j] )
        {
```

```
            res = false;
            break;
          }
        else j--;
    }
    return res;
}
```

Комментарии к этому варианту:

- строка в языке С представляется массивом элементов типа `char`;
- в примере используются две функции: `DtlSymbol` - удаление пробелов в строке `str` и `StrLen` - определение длины строки `str`.

Следует отметить, что приведенный вариант весьма неэффективен. Во-первых, просмотр сравниваемых символов выполняется от начала до конца (и от конца до начала), тогда как этот просмотр достаточно выполнять до середины строки. Во-вторых, сравнение достаточно проводить до первого не совпавшего символа. Эти недостатки устранены с следующем варианте решения задачи:

Вариант 2:

```
bool isPolindrom2( char str[] )
{
    DelSymbol(str, ' ');
    int n = StrLen(str);
    int i, j = n - 1;

    bool res = true;
    for ( i = 0; res && i < j; i++ )
    {
      if ( str[i] != str[j] )
        {
          res = false;
          break;
        }
        else j--;
    }
    return res;
}
```

И наконец, наиболее компактно выглядит вариант, использующий специфические операции языка С:

Вариант 3:

```
bool isPolindrom3( char str[] )
{
    DelSymbol(str, ' ');
    int n = StrLen(str);
```

```
    int i, j;

    bool res = true;
    for (i = 0, j = n - 1; res && i < j; i++, j--)
        res = (str[i] == str[j]);
    return res;
}
```

Обратите внимание, что в записи заголовка цикла for дважды используется операция «запятая».

3.6. Операторы goto, break; continue

Оператор goto

Оператор безусловного перехода. Синтаксис:

```
goto <идентификатор>;
...
<идентификатор>: <инструкция>
```

Осуществляет переход к выполнению программы начиная с <инструкции>.

Использовать оператор goto не рекомендуется, т.к. его применение усложняет понимание кода. Например:

```
int a, b, c, m;
if ( a > b ) goto met3;
if ( b > c ) goto met2;
met1: m = c;
     goto met4;
met2: m = b;
     goto met4;
met3: if ( a < c ) goto met1;
     m = a;
met4:;
```

Не сразу можно догадаться, что приведенный фрагмент кода выполняет выбор максимального числа из трех чисел. Приведенный ниже вариант решения той же задачи более понятен:

```
if ( a > b )
     if ( a > c )
          m = a;
     else
          m = c;
else
     if ( b > c )
          m = b;
     else
          m = c;
```

Операторы break и continue

Оператор `break` выполняет безусловный выход из ближайшего включающего этот оператор блока.

Оператор `continue` выполняет безусловный переход на проверку условия выхода из ближайшего включающего этот оператор цикла.

Использование операторов `break` и иллюстрирует `continue` следующий

Пример.

```
int x, sum = 0;
int all_count = 0;
int pos_count = 0;
cout << "Вводите числа:" << endl;
while( 1 )
{
    cin >> x;
     all_count++;
    if( x < 0 )
        continue;
    if( x == 0 )
        break;
    sum += x;
    pos_count++;
}
cout << "Введено " << all_count << " чисел" << endl;
cout << "из них " << pos_count << " положительных, ";
cout << "сумма положительных = " << sum << endl;
```

Приведенный фрагмент кода выполняет ввод с консоли последовательность целых чисел, подсчет количества введенных чисел, количества и суммы положительных чисел. Признаком конца последовательности чисел является ввод нуля.

4. МНОГОФАЙЛОВАЯ ПРОГРАММА

4.1. Схема построения многофайловой программы

4.1.1. Однофайловая программа

На рис. 4.1 представлена схема формирования исполняемого (exe) файла для программы, исходный код которой размещен в одном файле. Если исходный код невелик, то вполне можно обойтись представленным на схеме вариантом программы. Если же исходный код начинает разрастаться, то накладные расходы на его компиляцию становятся неоправданно высоки и возникает необходимость перехода к многофайловой организации программы.

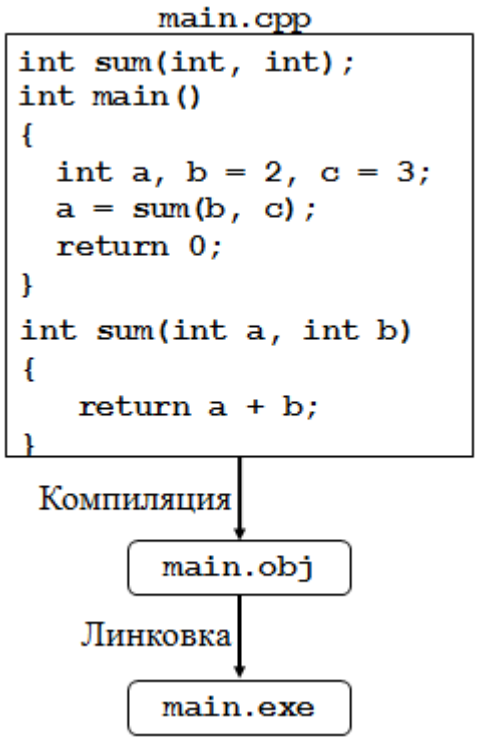


Рис. 4.1. Схема формирования исполняемого файла однофайловой программы.

4.1.2. Многофайловая программа. Директива препроцессора #include

Схема формирования многофайловой программы представлена на рис. 4.2.

В этой программе в файле main.cpp оставлена функция main; функция sum вынесена в отдельный исходный файл module.cpp. Если сделано только то, что описано в предыдущем пункте, то программа компилироваться не будет, т.к. перед функцией main в файле main.cpp нет объявления прототипа функции sum.

Для «исправления» этой ситуации в многофайловых программах языка С добавляют еще один файл module.h, в который вносят объявления всех функций файла module.cpp, а в файл main.cpp добавляют директиву препроцессора #include “module.h”. Перед компиляцией файл main.cpp будет обрабатываться препроцессором, который вместо директивы #include “module.h” вставит в файл main.cpp содержимое файла module.h и компиляция пройдет успешно.

Файлы с расширением cpp принято называть исходными файлами модуля, а соответствующие файлы с расширением h – заголовочными файлами модуля.

В заголовочные файла выносят объявления не только прототипов функций модуля, но и типов данных, глобальных переменных, констант, …

Заголовочные файлы модулей принято подключать с помощью директивы include не только к исходным файлам, использующим модули, но и к исполняемым файлам модулей.

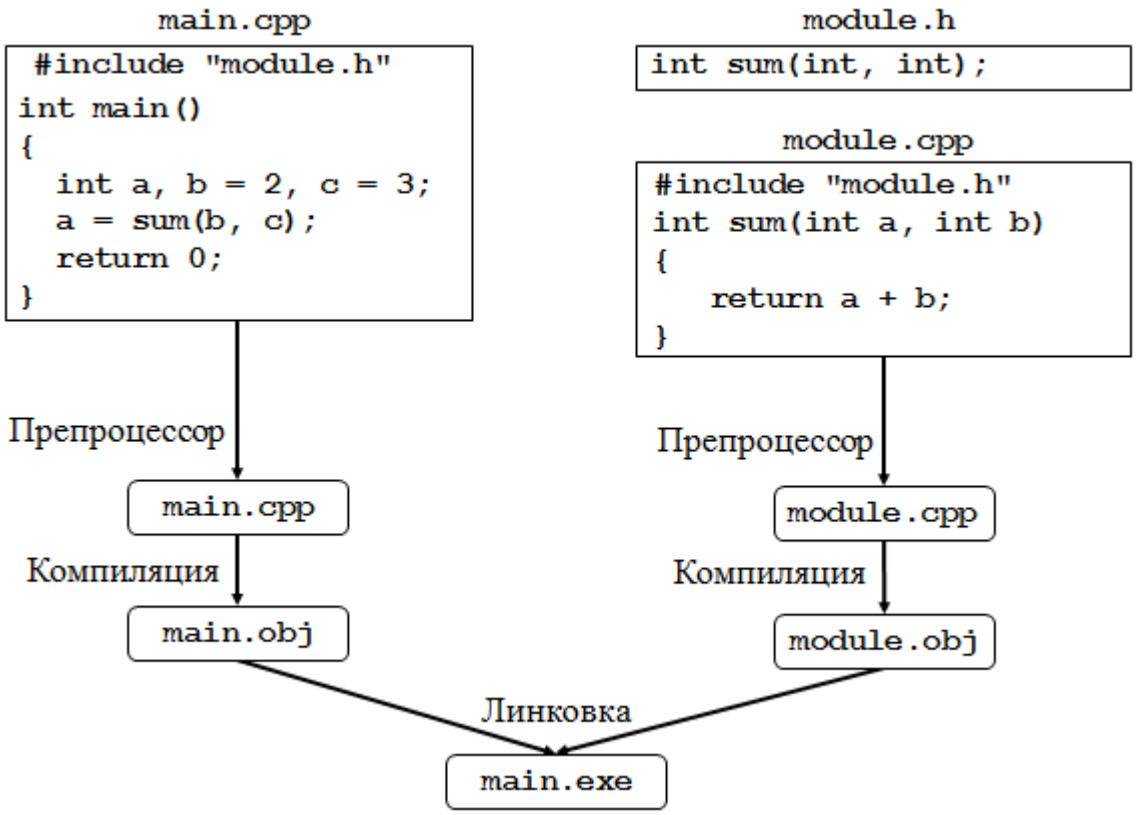


Рис. 4.2. Схема формирования исполняемого файла многофайловой программы.

4.2. Директивы препроцессора #ifndef, #define, #endif

На рис. 4.3 схема организации трехфайловой программы, включающей исходный файл main.cpp и два модуля: person и phone. Для упрощения на схеме представлены только заголовочные файлы модулей с объявлением структур Person и PhoneBook. В полной мере заголовочные файлы должны включать объявления прототипов функций работы с этими структурами.

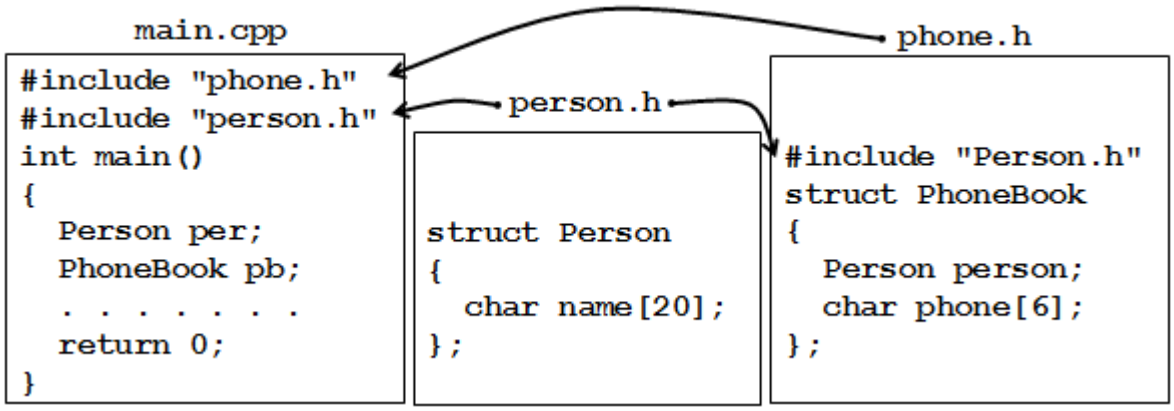


Рис.4.3. Схема организации трехфайловой программы.

После обработки препроцессором файла main.cpp, этот файл примет вид, представленный на рис. 4.3.1. При компиляции этого файла будет зафиксирована ошибка двойного объявления структуры Person.

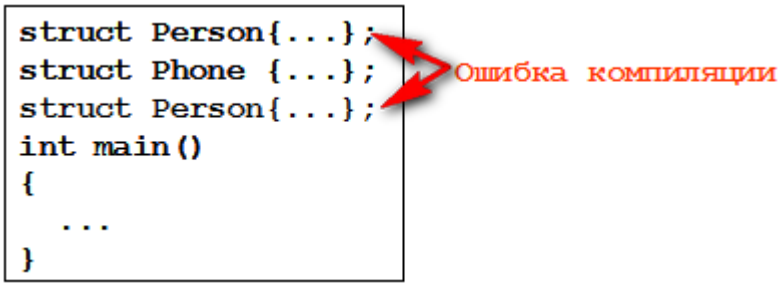


Рис. 4.3.1. Результат обработки препроцессором файла main.cpp рис. 4.3.

Для того избежать повторного срабатывания директивы include при вложенных связях между модулями программы, описание заголовочных файлов модулей должно включать использование директив препроцессора #ifndef, #define, #endif так, как это показано на рис. 4.4.

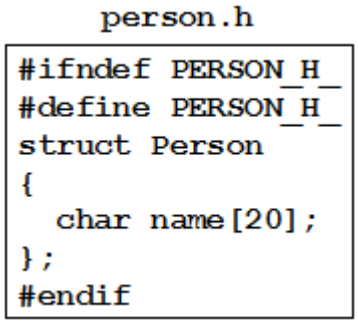


Рис. 4.4. Использование директив препроцессора #ifndef, #define, #endif в файле person.h.

Комментарии к рис.4.4 начнем с понятия макроса. Макрос это специальный объект препроцессора, который объявляется с помощью директивы #difine, которая в общем случае имеет вид:

```
#define <имя_макроса>[<параметры_макроса> <тело_макроса>]
```

В объявлении макроса обязательным является только имя макроса. Имя макроса это идентификатор, в записи которого принято использовать только строчные буквы, подчерк и цифры.

Параметры макроса и тело могут быть опущены; в нашем случае они не потребуются, и говорить о них пока не будем.

Для нас сейчас важно то, что каждый раз, встретив объявление макроса, препроцессор запоминает его имя и всегда помнит, был ли объявлен макрос с указанным именем.

Директива #ifndef (директива условной компиляции) имеет вид:

```
#ifndefi <имя_макроса>
```

и обрабатывается препроцессором так:

- если макрос <имя_макроса> не был объявлен, то весь код, следующий за директивой до директивы #endif переносится в код программы;
- иначе выполняется переход на директиву #endif.

Итак, в случае заголовочный файл, представленных на рис. 4.4 будет обрабатываться так:

- если макрос PERSON_H_ еще не был объявлен, то он будет объявлен и объявление структуры Person будет добавлено в формируемый препроцессором код;
- иначе этого сделано не будет.

Полная схема организации трехфайловой программы с использованием директив #ifndef, #define, #endif представлена на рис. 4.5.

main.cpp

```
#include "phone.h"
#include "person.h"
int main()
{
  Person per;
  PhoneBook pb;
  . . . . . .
  return 0;
}
```

person.h

```
#ifndef PERSON_H_
#define PERSON_H_
struct Person
{
  char name[20];
};
#endif
```

phone.h

```
#ifndef PHONE_H_
#define PHONE_H_
#include "Person.h"
struct PhoneBook
{
  Person person;
  char phone[6];
};
#endif
```

Рис.4.5. Схема организации трехфайловой программы с использованием директив #ifndef, #define, #endif.

4.3. Общие правила формирования заголовочных файлов

4.3.1. Общая схема заголовочного файла

Общая схема заголовочного файла имеет вид:

```
#ifndef <имя_модуля>
#define <имя_модуля>

#include "<необходимый_модуль1>" // Если необходимо
#include "<необходимый_модуль2>" // Если необходимо
...
#include "<необходимый_модульn>" // Если необходимо

// Интерфейсная часть модуля
...
#endif
```

В заголовочном файле модуля выполняется подключение заголовочных файлов необходимых модулей. В интерфейсную часть модуля включают объявления типов данных, глобальных переменных, констант и прототипов функций модуля.

4.3.2. Куда подключаются заголовочные файлы?

1. Заголовочный файл (module.h) целесообразно подключать к «своему» исполняемому файлу (module.cpp).
2. Если module1 использует объекты module2, то:

- если (и только если) объекты module2 используются в заголовочном файле module1, то заголовочный файл module1.h подключается к заголовочному файлу module2.h и не подключается (при соблюдении правила 1) к исполняемому файлу module2.cpp.
- иначе заголовочный файл module1.h подключается к исполняемому файлу module2.cpp и не подключается к заголовочному файлу module2.h.

5. МАССИВЫ И УКАЗАТЕЛИ

5.1. Массивы (введение)

5.1.1. Массив, его объявление и инициализация

Объявление массива

Массив это набор пронумерованных (индексированных) однотипных элементов.

В общем виде объявление массива выглядит так:

<тип> <идентификатор>[<константа>]

где: <константа> — количество элементов массива

Примеры:

```
int array[20];
double delta, beta[30], alpha;
```

В этом примере delta, alpha — простые переменные, beta[30] – массив из 30 double

В языке С при объявлении массива в качестве его длины может быть указана константа или константное выражение.

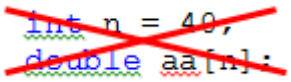
Пример:

```
double mas[100];
```

Предпочтительнее длину массива задавать с помощью макроса #define (о макросах будем говорить ниже), как это показано в следующем примере:

```
#define MAX 20
int array[MAX];
int ar[MAX+10];
```

Ни в коем случае нельзя в качестве длины массива указывать переменную (как это, к сожалению часто пытаются делать начинающие программисты):



Объявление (заказ) массива переменной длины (длины, определяемой в процесса выполнения программы) может быть сделано с помощью динамической памяти, к которой тоже будем говорить ниже.

Инициализация массива

При объявлении массива его элементы могут быть инициированы начальными значениями. Список инициализаторов элементов массива записывается в фигурных скобках через запятую.

Пример:

```
int days[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

В примере объявлен массив days, инициированный количествами дней в каждом из 12 месяцев года.

При инициализации массива его длину можно не указывать. Компилятор сам подсчитает длину массива по количеству инициализаторов. Приведенный выше пример можно было записать и так:

```
int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Длина массива days в этом случае тоже будет 12.

Если при объявлении массива с инициализатором объявленная длина массива больше количества инициализаторов, то проинициированы будут только первые элементы массива.

Пример:

```
int days[12] = {31, 28, 31, 30};
```

В массиве days длиной 12 элементов инициированы будут только первые четыре элемента.

Ошибкой компиляции является случай, когда количество инициаторов массива больше объявленной длины.

5.1.2. Использование массивов

После того, как массив объявлен, можно обращаться к его элементам. В общем виде обращение к элементу массива выглядит так:

<идентификатор_массива>[<индексное_выражение>];

где: < индексное_выражение > — выражение типа «целое», определяющее порядковый номер элемента массива, к которому ведется обращение.

Пример:

```
#define MAX 20
int array[MAX];
array[3] = 8;
int i = 4, j = 7;
int a = array[i] * array[j];
int d = array[i + j];
```

Элементы массива в языке С всегда нумеруются с 0. Иными словами, элементы массива длиной 20 будут иметь номера: 0, 1, 2, …, 19. Или, номер последнего элемента массива длиной m будет m-1.

5.1.3. Массив как параметр функции

Массив можно передавать в функцию как один из ее параметров. Передачу массива функции проиллюстрируем следующим примером.

```
int SumArray( int[], int );  // Прототип функции
int main()
{
  int a[20], b[10];
  // Ввод a, b
  int sa = SumArray(a, 20);
  int sb = SumArray(b, 10);

  return 0;
}
int SumArray(int ms[], int n)
{
  int sum = 0, i;
  for ( i = 0; i < n; i++ ) sum += ms[i];
  return sum;
}
```

Комментарии:

1. Объявлен протопоп функции SumArray, получающей массив целых (`int[]`) и целое.
2. В функции main объявлены:

- массив a длиной 20 элементов;
- массив b длиной 10;
- переменная sa, которой присваивается значение функции SumArray с передачей ей массива a и константы 20.

3. Функция SumArray:

- получает массив ms и переменную n (при первом обращении к функции это будет массив a и 20);
- вычисляет сумму элементов полученного массива и возвращает ее в качестве результата.

4. При повторном обращении к функции SumArray ей будет передан массив b и его длина 10.

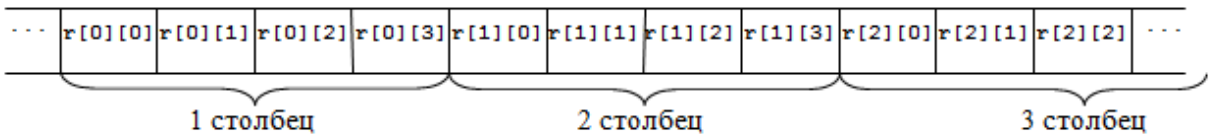
5.1.4. Многомерные массивы

В языке С могут использоваться и многомерные массивы. При объявлении многомерных массивов длина каждой размерности указывается в отдельных квадратных собках.

Пример:

```
int r[3][4];
```

В памяти двумерные массивы размещаются «по столбцам»: сначала меняется второй индекс, потом – первый:



Инициализация многомерных массивов имеет «вложенный» вид:

```
int r[3][4] = {{5, 7 , 45, 10},
               {6, 71, 48, 11},
               {8, 75, 55, 12} };
```

5.1.5. Примеры

Ввод элементов массива

Задача. Написать функцию ввода элементов массива и функцию main, иллюстрирующую применение этой функции.

Решение:

```
void intArrayInput(int ar[], int nar)
{
     int i;
     for ( i = 0; i < nar; i++ )
     {
       printf("[%3d] = ", i + 1);
       scanf("%d", &ar[i]);
```

```
      }
}
#define MAX = 200
int main()
{
  int a[MAX];
  int na;
  do
  {
       printf("Количество элементов массива = ");
       scanf("%d", &na);
       if ( na <= 0 || na >= MAX )
       {
            printf("Количество элементов массива должно быть > 0 и
< %d.", MAX);
       }
  } while ( na <= 0 || na >= MAX );
  intArrayInput(a, na);

  return 0;
}
```

Комментарии.

1. Функция `intArrayInput` получает массив целых и его длину. Ввод значений выполняется с помощью цикла for. Каждое значение массива вводится с предварительной подсказкой, какой элемент надо вводить. Организованный таким образом ввод будет выглядеть так:

[1] = 18

[2] = -16

.....

2. В функции `main` объявляется массив a длиной MAX и переменная na, в которую будет вводиться реальная длина массива. Далее идет цикл do, в котором организуется ввод длины массива с подсказкой и контролем значения na, которое должно быть не менее 1 и не более MAX.

Задача. Написать функцию поиска максимального элемента массива целых.

Решение 1:

```
int maxInt1(int Arr[], int nArr)
{
     int max;
     int i;
     max  = Arr[0];
     for ( i = 1; i < nArr; i++ )
       if ( Arr[i] > max )
```

```
        {
           max = Arr[i];
        }
      return max;
}
```

Надеемся, что представленное решение в комментариях не нуждается.

Решение 2:

```
int maxInt2(int Arr[], int nArr)
{
      int nmax;
      int i;
      nmax = 0;
      for ( i = 1; i < nArr; i++ )
        if ( Arr[i] > Arr[nmax] )
        {
           nmax = i;
        }
      return Arr[nmax];
}
```

Комментарий:

В решении 2 в отличие от решения 1 отслеживается не максимальный элемент, а его номер. Преимущество решения 2 состоит в том, что определяется не только максимальный элемент массива, но и его номер, что может оказаться полезным при сортировке массива.

5.2. Указатели

5.2.1. Понятие указателя

Указатель на тип вводится как некоторый новый тип данных.

Определение. Переменная типа «указатель на <тип>» есть переменная, значениями которой является множество адресов переменных типа <тип>.

Иными словами, значением переменной типа «указатель на <тип>» является адрес переменной с типом < тип >.

Синтаксис объявления переменной типа «указатель на <тип> имеет вид:

<тип>* <идентификатор>;

Пример:

```
int *pi;        // pi - указатель на int
double* pd;     // pd - указатель на double
char *pc;       // pc - указатель на char
```

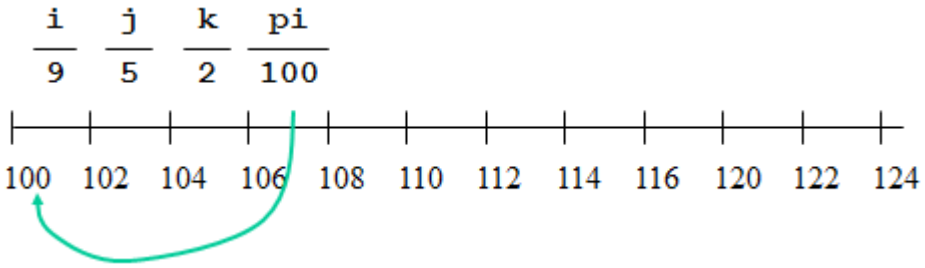
Как показано в примере, местоположение символа «*» роли не играет, но при этом надо быть внимательным, «действие» звездочки распространяется только на один объявляемый идентификатор списка:

```
int* a, b;          // a – int*, b – int
int c, *d;          // c – int, d – int*
```

В этом фрагменте объявлены два указателя на int: (a и d) и две переменные типа int (b и c).

Пример:

```
int i = 9, j = 5, k = 2, *pi;
pi = &i;      // pi указывает на i
```



Комментарии:

В примере объявлены и инициированы три переменные целого типа и указатель на целое (pi). Затем pi присвоено значение адреса i, после чего pi стал указывать на i.

После выполнения операции:

```
pi = &j;      // pi указывает на j
```

pi станет указывать на j.

Нулевой указатель. Нулевой указатель – указатель, который ни на что не указывает. Присвоить нулевое значение указателю можно так:

```
pi = 0;
```

или так:

```
pi = NULL;
```

5.2.2. Операции с указателем

Операция разыменовывания

Операция разыменовывания указателя – обращение к переменной, на которую указывает указатель. Синтаксис операции:

*<указатель>

Тип значения - <тип> указателя (и переменной, на которую он указывает).

Пример:

```
int i = 9, j = 5, k = 2;
int* pi = &i;
*pi = 3;          // i = 3
pi = &j;
k = i + *pi;      // k = 8
```

Комментарии:

1. В примере (первая строка) объявлены три переменные целого типа, которые инициированы начальными значениями.
2. Во второй строке объявлен указатель на целое (pi), который инициирован адресом переменной a (pi указывает на a).
3. В третьей строке примера разыменованному pi (т.е. переменной a) присваивается значение 3.
4. В четвертой строке pi присваивается значение адреса переменной j (т.е. pi начинает указывать на j).
5. В пятой строке переменной k присваивается значение суммы i и разыменнованного pi (т.е. j).

Операции сравнения

Значения указателей можно сравнивать. При выполнении операций сравнения указателей сравниваются их значения, т.е. значения адресов, на которые они указывают.

Пример:

```
int ms[20], *p1, *p2;
p1 = &ms[3];
p2 = &ms[5];
if ( p1 == p2 )
     *p1 = 3;
if ( p1 != p2 )
     *p2 = 8;
if ( p1 <= p2 )
```

```
    *p1 = *p2;
```

Комментарии:

1. В примере (первая строка) объявлены целочисленный массив и два указателя на целое.
2. Далее указателю p1 присваивается значение адреса третьего элемента массива, а указателю p2 – значение адреса пятого элемента массива.
3. Затем сравниваются значения указателей p1 и p2, и если бы они были равны, то третьему элемента массива было бы присвоено значение 3.
4. Остальное аналогично.

Обратите внимание на различие в сравнении указателей и разыменованных указателей. В первом случае:

```
if ( p1 < p2 )
     . . . . . .
```

сравниваются указатели (адреса, на которые они указывают), а во втором:

```
if ( *p1 < *p2 )
     . . . . . .
```

сравниваются значения, на которые указывают указатели.

Операции сложения, вычитания

Указатели можно складывать с целым и вычитать из него целое. Результат выполнения этих операций – указатель. При сложении указателя с целым значение указателя (адрес) увеличивается на слагаемое целое, умноженное на длину типа указателя. При вычитании значение указателя уменьшается на аналогичную величину:

```
<тип> var, *p1 = &var, p2, p3;
int i = 3;
p2 = p1 + i;         // p = &var + i*sizeof(<тип>)
p3 = p1 - i;         // p = &var - i*sizeof(<тип>)
```

Пример:

```
int ms[20], *p;
p = &ms[0];
*(p + 3) = 8;  // *(p + 3) ≡ ms[3]
```

Операции инкремент, декремент

К указателям применимы операции инкремент и декремент:

```
<тип> var, *p = &var;
p++;       // p = p + sizeof(<тип>)
p--;       // p = p - sizeof(<тип>)
```

5.3. Массивы и указатели

5.3.1. Массив = указатель

В языке С идентификатор массива является указателем на первый элемент массива. Иными словами, по определению (операции индексации):

ms[i] есть *(ms + i)

Обращение к i-тому элементу массива может выглядеть привычно:

```
int ma[5], i;
for ( i = 0; i < 5; i++ )
   ma[i] = i;
```

менее привычно:

```
for ( i = 0; i < 5; i++ )
   *(ma + i) = i;
```

и совсем не привычно:

```
for ( i = 0; i < 5; i++ )
   i[ma] = i;
```

5.3.2. Массив – константный указатель

Идентификатор массива является константным указателем.

Пример.

```
int main()
{
  int ma[5], i, s;
  intArrayInput(ma, 5);
  s = 0;
  for ( i = 0; i < 5; i++ )
     s += ma[i];
  return 0;
}
```

В приведенной функции main объявлен массив, выполнено обращение к функции ввода его значений и подсчитана сумма его элементов. Здесь все в порядке. Но если этот пример записать так:

```
int main()
{
```

```
  int ma[5], i, s;
  intArrayInput(ma, 5);
  s = 0;
  for ( i = 0; i < 5; i++ )
     s += *ma++; // Ошибка: lvalue requite
  return 0;
}
```

то в строке, отмеченной комментарием будет зафиксирована ошибка: значение указателя, являющегося идентификатором массива менять нельзя.

Исключение составляет случай, когда массив является параметром функции.

Пример.

```
int sum(int *ma , int n)
{
   int i, s = 0;
   for ( i = 0; i < n; i++ )
      s += arr[i];
   return s;
}

int main()
{
  int ma[5], i, s;
  intArrayInput(ma, 5);
  s = sum(ma, 5);
  return 0;
}
```

В этом примере суммирование элементов массива выполняется функцией sum, в которой обращение к элементам массива выполняется с помощью операции индексации.

Функция sum могла быть описана в виде, при котором обращение к элементам массива выполняется через указатель:

```
int sum(int arr[], int n)
{
   int i, s = 0;
   for ( i = 0; i < n; i++ )
   s += arr[i];
   return s;
          for ( i = 0; i < 20; i++ )
        s += *arr++;      // Можно!!
 }
```

Обратите внимание, что в заголовке функции получение функцией массива может быть объявлено как `int arr[]` или как `int *ma`. Оба эти варианта

эквивалентны, и при каждом из них обращение к элементам массива может выполняться либо через операцию индексации, либо через указатель.

5.4. Еще об указателях

5.4.1. Указатель на void (родовой указатель)

В языке С можно объявлять указатель на void (объявлять переменные типа void нельзя!). При этом:

1. Указатель на void это указатель на «сырую память», не имеющую типа, его нельзя разыменовывать, использовать в арифметических операциях и др.
2. Указатели типа void* можно сравнивать друг с другом.
3. Указатель на void это «родовой» указатель, ему можно присвоить указатель на объект любого типа, но при этом надо «помнить», указатель на какой объект содержит указатель на void.

Пример:

```
int i, j, *pi; double* pd;
void* p = &i;      // Верно
pi = (int*)(p);    // Верно
j = *(int*)p;      // Верно
pd = (double*)(p);// Верно, но...
```

В приведенном примере:

1. Объявляется p - указатель на void, который инициируется адресом переменной i.
2. pi (указателю на int) присваивается p, приведенное к указателю на int.
3. j присваивается разименованный p, приведенный к указателю на int.
4. pd (указателю на double) присваивается p, приведенное к указателю на double. Компилятор не заметит, но на что сейчас указывает pd?

5.4.2. Указатель на константу, константный указатель

Указатель на константу не может менять значение объекта, на который указывает, но может быть переуказан на другой объект.

Синтаксис объявления указателя на константу:

```
const <тип>* <идентификатор>
```

Пример:

```
int i = 9, j = 5, k = 4;
const int* pi;
pi = &i;
*pi = 3;        // Ошибка компиляции
j = *pi + k;    // Допустимо
pi = &j;        // Допустимо
```

Константный указатель может менять значение объекта, на который указывает, но не может быть переуказан на другой объект.

Синтаксис объявления указателя на константу:

```
<тип>* const <идентификатор>
```

Пример:

```
char c = 5;
char* const win_core = &c;
char ch;
win_core = &ch;      // Ошибка компиляции
*win_core = 8; // c = 8
```

5.5. Классы и типы памяти

5.5.1. Область видимости и время жизни объектов

Область видимости

Область видимости объекта программы (переменной, константы, типа данных, функции) — часть текста программы, в которой объект доступен (может быть использован).

Области видимости может быть:

- локальной, когда объект объявлен в блоке и виден в этом блоке и во всех вложенных блоках;
- файловой, когда объект объявлен в файле и виден во всем файле, начиная с места его объявления;
- глобальной (межфайловой), когда объект, объявленный в одном файле при специальных объявлениях виден в другом файле.

Действует правило перекрытия видимости: при наложении областей видимости одноименных объектов внутренние видимости перекрывают внешние.

Видимость функций является файловой с момента описания функции или объявления ее прототипа. При объявлении прототипа функция может быть описана в другом файле.

Время жизни

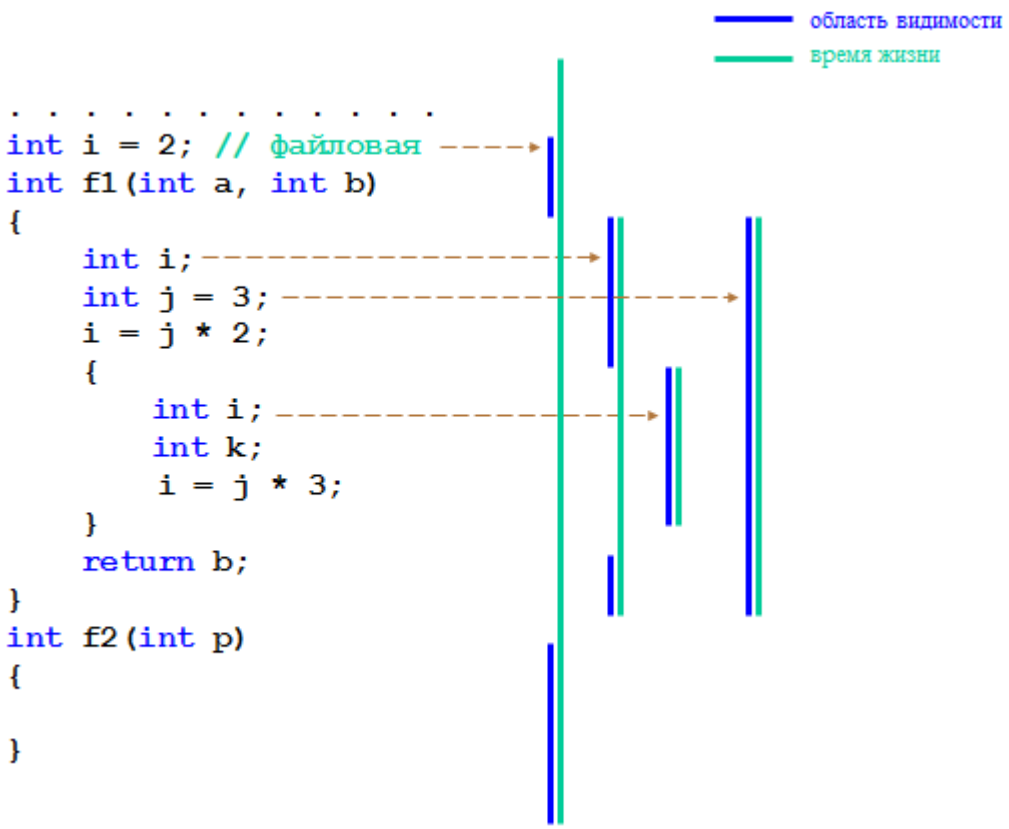
Время жизни объекта — интервал времени выполнения программы, в течение которого объект существует.

Время жизни объекта может быть:

- глобальным – время работы всей программы (при этом не обязательно объект всегда видим);
- локальным - время выполнения блока (при новом входе в блок объект обновляется);

У функций время жизни глобальное.

Пример.



В приведенном примере:

- файловая переменная i имеет глобальное время жизни и файловую область видимости, которая перекрывается локальной переменной i функции f1;
- все переменные функции f1 являются блочными, имеют отмеченные на схеме примера локальные области видимости и локальные времена жизни.

5.5.2. Классы памяти и механизмы управления памятью

Классы памяти

Время жизни и область видимости переменных определяется не только местом их объявления, но и спецификатором класса памяти, указанным при объявлении переменной. В языке С используются четыре спецификатора класса памяти: `register`, `auto`, `static`, `extern`.

Спецификатор класса памяти объявляемой переменной подразумевается всегда; при его отсутствии работает принцип умолчания (см. ниже).

В таблице 5.1 приведены характеристики областей видимости и времен жизни коассов памяти языка С.

	register	auto	static	extern
В блоке	✓	✓*	✓	✓
В файле	×	×	✓*	✓
Время жизни	локальное	локальное	глобальное	глобальное
Видимость	локальная	локальная	лок./файл.	лок./файл.

Таблица 5.1. Время жизни и область видимости спецификаторов классов памяти.

В первых двух строках таблицы отмечено, где могут быть объявлены переменные с соответствующими спецификаторами классов памяти:

- при объявлении в блоке могут использоваться все четыре спецификатора класса памяти; при отсутствии явного объявления спецификатора класса памяти по умолчанию – auto;
- при объявлении в файле могут использоваться только спецификаторы static или extern; при отсутствии явного объявления спецификатора класса памяти по умолчанию – static.

Время жизни и область видимости переменных классов памяти register и auto локальные.

Время жизни переменных классов памяти static и extern глобальное. Область видимости этих переменных локальная или файловая в зависимости от места объявления (блок или файл).

Пример.

```
[static] int alpha;  // ВЖ – глоб., ОВ - файловая
void fun(int beta)   // ВЖ – лок., ОВ – лок.
{
  [auto] int gamma;  // ВЖ – лок., ОВ – лок.
  //...
  {
    static int delta;     // ВЖ – глоб., ОВ – лок.
    [auto] int epsilon;   // ВЖ – лок.,  ОВ – лок.
    extern int zeta;      // ВЖ – глоб., ОВ – лок.
    register int eta;     // ВЖ – лок.,  ОВ – лок.
    //...
```

```
    }
    //...
}
```

Время жизни и область видимости переменных примера отмечены в комментариях.

Типы (сегменты) памяти

Для размещения выполняемой программы операционная система предоставляет несколько сегментов памяти. Каждый сегмент является непрерывным по адресации фрагментом физической памяти компьютера.

Выделяются сегменты следующих типов:

- программный сегмент служит для размещения программного кода (функций);
- статический сегмент служит для размещения статических (static) переменных;
- автоматический (стековый) сегмент служит для размещения переменных класса памяти auto;
- динамический (куча) сегмент служит для размещения переменных, память под которые запрашивается и освобождается в процессе выполнения программы.

В процессе подготовки и выполнения программы работают различные механизмы управления различными типами памяти.

Программный код (функции) размещается в программном сегменте на шаге загрузки программы до начала ее выполнения и остается неизменным все время ее выполнения.

Память под статические объекты программы выделяется в статическом сегменте также на шаге загрузки программы и остается неизменной (не пере выделяется) все время выполнения программы.

Память под объекты класса auto выделяется и освобождается в стековом сегменте автоматически в процессе выполнения программы. При этом действует следующий механизм управления стековой памятью:

- стековый сегмент разбит на две половины: одна половина (левая) содержит локальные переменные активных блоков, вторая (правая) содержит свободную память;
- начало свободной памяти стека хранится в специальном указателе;

- в начале выполнения очередного блока в стеке справа от указателя выделяется память под локальные переменные этого блока; указатель стека смещается вправо; эти переменные становятся видимыми и начинают жить;

- при завершении блока указатель стека перемещается влево не все переменные завершаемого блока; переменные блока становятся невидимыми, время жизни их прекращается.

Память в куче выделяется и освобождается в процессе выполнения программы не автоматически, а по явному запросу на выделение и освобождении памяти (см. раздел Динамические массивы).

5.6. Динамические массивы

5.6.1. Динамические массивы в C

В языке C нет встроенных операций запроса и освобождения памяти в куче. Эти действия в C выполняются с помощью специальных функций. Наиболее часто используются функции malloc и free, которые находятся в библиотеке stdlib.

Синтаксис у этих функций следующий :

```
void* malloc(size_t size);
void free(void*);
```

Функция запроса памяти в куче malloc:

- получает целое size – количество запрашиваемых байт памяти;

- выделяет в куче фрагмент памяти запрашиваемой длины;

- возвращает значение указателя на эту память как указатель на void.

Функция освобождения памяти в куче free:

- получает значение указателя на ранее запрошенную в куче память;

- освобождает эту память;

В общем виде запрос памяти под массив из n элементов типа <тип> будет выглядеть:

```
<тип>* <указатель> = (<тип*>)malloc(n*sizeof(<тип>));
```

Освобождение памяти:

```
free(<указатель>);
```

Пример.

```
#include <stdlib.h>
int* createAndInputIntArray(int arraySize)
{
   int i;
   int* ar = (int*)malloc(arraySize * sizeof(int));
   for ( i = 0; i < arraySize; i++ )
   {
      printf("[%3d] = ", i);
      scanf("%d", &ar[i]);
   }
   return ar;
}
int main()
{
   int n, *mas;
   do
   {
     printf("Введите количество элементов массива:");
     scanf("%d", &n);
   } while ( n <= 0 );
   mas = createAndInputIntArray(n);
   // Использование массива mas
   free(mas);
   return 0;
}
```

В приведенном примере:

1. Описана функция `createAndInputIntArray`, создающая в куче целочисленный массив длиной `arraySize` и выполняющая ввод значений этого массива. Функция возвращает указатель на созданный массив.
2. В функции main вводится значение длины массива; выполняется обращение к функции `createAndInputIntArray` и после использования сформированного массива mas выполняется освобождение занятой им в куче памяти.

5.6.2. Динамические массивы в C++

В языке C++ есть встроенные операции запроса и освобождения памяти в куче: new и delete.

В общем виде обращение к этим операциям выглядит так.

- запрос памяти под один экземпляр переменной типа <тип>:

```
<тип>* <указатель> = new <тип>;
```

- запрос памяти под массив из n элементов типа <тип>:

```
<тип>* <указатель> = new <тип>[n];
```

- освобождение памяти, занятой одним элементом:

```
delete <указатель>;
```

- освобождение памяти, занятой массивом:

```
delete[] <указатель>;
```

Пример.

Приведенный выше пример в C++ будет выглядеть так:

```
int* createAndInputIntArray(int arraySize)
{
   int* ar = new int[arraySize];
   for ( int i = 0; i < arraySize; i++ )
   {
      cout << "[" << i << "] = ";
      cin >> ar[i];
   }
   return ar;
}
int main()
{
   int n;
   do
   {
     cout << "Введите количество элементов массива:";
     cin >> n;
   } while ( n <= 0 );
   int* mas = createAndInputIntArray(n);
   // Использование массива mas
   delete[] mas;
   return 0;
}
```

5.6.3. Многомерные динамические массивы

Массивы указателей, указатели на указатели

Многомерные динамические массивы формируются с помощью указателей на указатели и массивов указателей.

Принцип формирования многомерных массивов выглядит следующим образом.

Если мы объявим в программе массив указателей:

```
int* array_of_pointers[5];
```

то получим массив (рис. 5.1), под каждый элемент которого можно заказывать память в куче:

```
array_of_pointers[0] = new int[10];
array_of_pointers[1] = new int[8];
. . . . . .
```

и запись:

```
array_of_pointers[1][3] = 12;
```

будет означать присваивание значения 12 третьему элементу первого массива.

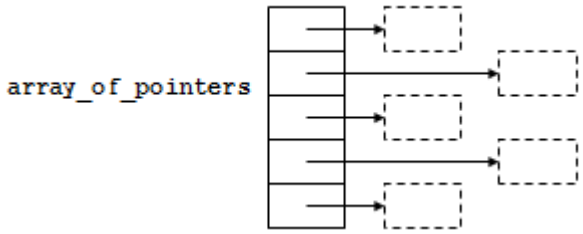


Рис. 5.1. Массив указателей.

Если мы теперь объявим указатель на указатель (рис. 5.2):

```
int** ppi;
```

затем под этот указатель на указатель закажем в куче память под массив из m указателей:

```
ppi = new int*[m];
```

и, наконец, под каждый элемент этого массива закажем в куче массивы длиной n:

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

В результате нас будет сформирован двумерный массив.

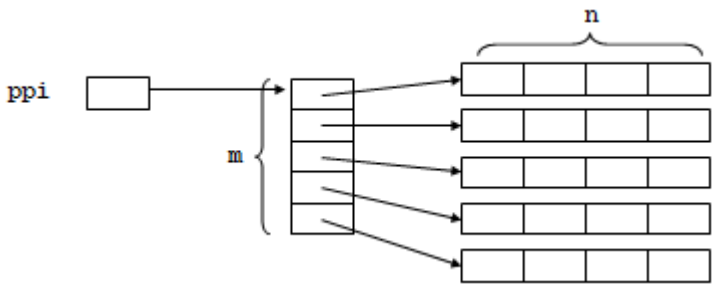


Рис. 5.2. Указатель на указатель.

Двумерный динамический массив

Пример. Написать функции создания и освобождения двумерного динамического массива

```
// Создание двумерного динамического массива -----------
double** crtA2d(int n, int m)
{
   double **ms = new double*[n];
   for ( int i = 0; i < n; i++ )
      ms[i] = new double[m];
   return ms;
}
// Освобождение двумерного динамического массива -------
void delA2d(double **ms, int n)
{
   for ( int i = 0; i < n; i++ )
      delete[] ms[i];
   delete[] ms;
}

int main()
{
   int n1, n2;
   cout << "n1 = "; cin >> n1;
   cout << "n2 = "; cin >> n2;
   double **mat = crtA2d(n1, n2);  // Создание массива
   for (int i = 0; i < n1; ++i)
      for (int j = 0; j < n2; ++j)
         mat[i][j] = i*100 + j;

  // . . . . . . . . . . . . . . . . . . . . . . . .
   delA2d(mat, n1); // Освобождение массива
   return 0;
}
```

В приведенном примере:

1. Создание двумерного динамического массива описано выше.
2. Освобождение массива выполняется в обратном порядке: сначала освобождается память, занятая строками массива (массивами целых); замем освобождается память, занятая массивом указателей.
3. Функция main иллюстрирует использование функций примера.

Трехмерный динамический массив

Пример. Написать функции создания и освобождения трехмерного динамического массива.

```
// Создание трехмерного динамического массива -----------
int*** crtA3I(int n1, int n2, int n3)
{
   int i1, i2;
   int ***ms;
   ms = new int**[n1];
   for ( i1 = 0; i1 < n1; i1++ )
   {
      ms[i1] = new int*[n2];
      for ( i2 = 0; i2 < n2; i2++ )
         ms[i1][i2] = new int[n3];
   }
   return ms;
}
// Освобождение трехмерного динамического массива -------
void delA3I(int ***ms, int n1, int n2)
{
   int i1, i2;
   for ( i1 = 0; i1 < n1; i1++ )
   {
      for ( i2 = 0; i2 < n2; i2++ )
         delete[] ms[i1][i2];
      delete[] ms[i1];
   }
   delete[] ms;
}
```

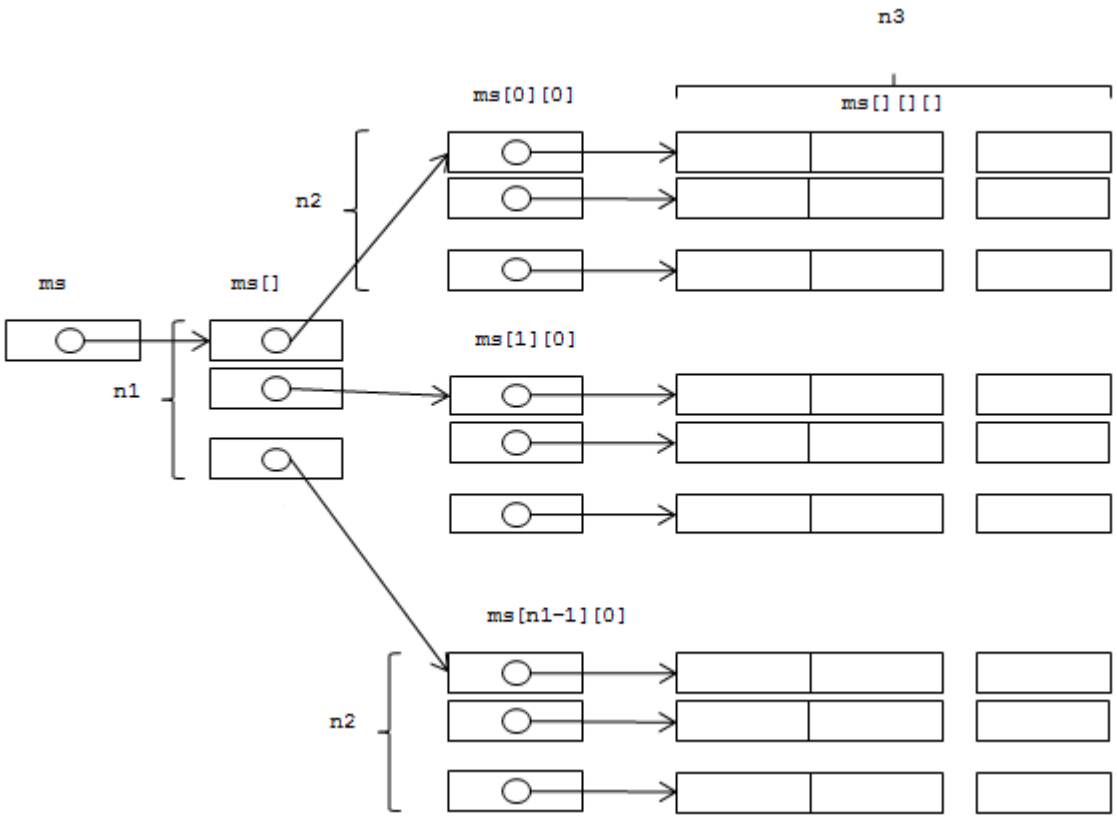


Рис. 5.3. Схема формирования трехмерного динамического массива..

6. ФУНКЦИИ

6.1. Механизмы передачи параметров функции

6.1.1. Введение

В языке С (как и большинстве алгоритмических языков) используется три способа (механизма) передачи параметров вызываемой функции:

- передача параметра по значению;
- передача параметра по указателю;
- передача параметра по ссылке (только в С++).

Разберемся с этими механизмами подробнее.

6.1.2. Передача параметра по значению

При передаче параметра по значению в качестве фактического параметра вызова функции указывается выражение (в частном случае - имя передаваемой переменной); в качестве формального параметра функции тоже имя переменной. При этом:

- каждый формальный параметр является локальной переменной функции;
- при вызове функции значения фактических параметров заносятся в соответствующие им формальные параметры;
- после выполнения функции значения (возможно измененных) формальных параметров назад не возвращаются.

Пример.

```
void Sum(int c, int d, int s)
(2)
{
     // Присваиваем переменной
(3)  s = c + d;
}

int main()
{
     int a = 1;
```

```
(1)   Sum(3, 4, a); // Передаем переменную
(4)   return 0;
}
```

В приведенном примере функция Sum получает три целых параметра и сумму первых двух присваивает третьему. В функции main обвялена переменная a, которая инициирована значением 1 и выполняется вызов функции Sum, которой передается две константы и переменная a. Что произойдет с переменной a в результате этих действий?

Рассмотрим ситуацию по шагам:

1. Шаг (1). Перед вызовом функции Sum состояние памяти с переменными программы представлено на рис.6.1: выделена память под переменную a, ей присвоено значение 1.

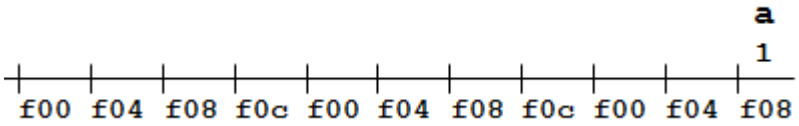


Рис. 6.1.

2. Шаг (2). Перед выполнением вызванной функции Sum состояние памяти с переменными программы представлено на рис.6.2: выделена память под переменные c, d и s, им присвоены переданные при вызове значения.

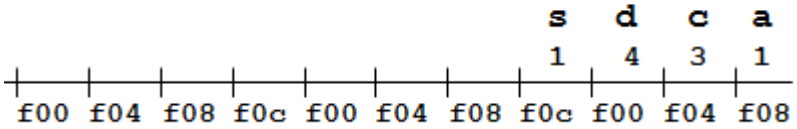


Рис. 6.2.

3. Шаг (3). При выполнении функции Sum переменная s получает новое значение (рис.6.3).

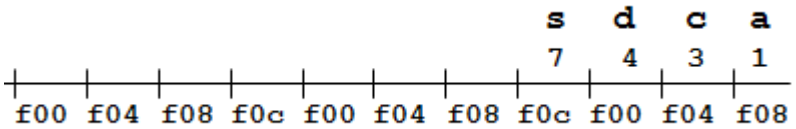


Рис. 6.3.

4. Шаг (4). При возврате в функцию main (рис. 6.4) переменные c, d и s пропадают, а переменная a остается неизменной.

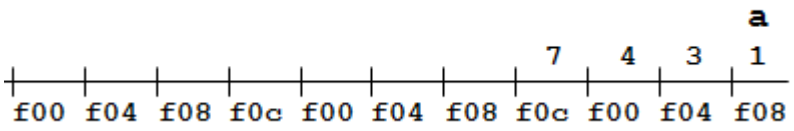


Рис. 6.4.

Часто возникает вопрос: как быть если измененное значение формального параметра надо вернуть вызывающей функции? Для этого надо использовать передачу параметра по указателю или по ссылке.

6.1.3. Передача параметра по указателю

При передаче параметра по указателю:

- формальный параметр, через который будет возвращаться значение должен быть объявлен как указатель;
- соответствующий фактический параметр не может быть константой или выражением, а должен быть: указателем на объект (переменную) или адресом объекта (переменной), значение которой будет изменено после возврата из функции;
- в функции возвращаемое значение должно быть присвоено разименованному значению формального параметра.

Пример.

```
// Получаем указатель на переменную
void Sum(int c, int d, int* s)
(2)(5)
{
     // Присваиваем разыменованному указателю
(3)(6)   *s = c + d;
}

int main()
{
     int a = 1, b = 3;
     int *pb = &b;
(1)  Sum(3, 4, &a); // Передаем адрес переменной
(4)
     Sum(5, 6, pb); // Передаем указатель на переменную
     return 0;
}
```

В приведенном примере функция Sum получает два целых параметра и указатель на целое. Сумму первых двух присваивает разименованному

третьему. В функции main обвялены переменные a и b, которые инициированы значениями 1 и 3; указатель на целое, инициированный адресом переменной b. Выполняется вызов функции Sum, которой передается две константы и адрес переменной a и вызов функции Sum с передачей двух констант и указателя pb. Что произойдет с переменными a и b в результате этих действий?

Рассмотрим ситуацию по шагам:

1. Шаг (1). Перед первым вызовом функции Sum состояние памяти с переменными программы представлено на рис.6.5. Выделена память под переменные a и b, им присвоены значения 1 и 2. Выделена память под указатель pb и ему присвоено значение адреса переменной b.

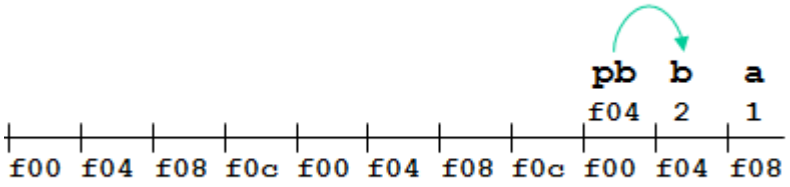


Рис.6.5.

2. Шаг (2). Перед выполнением вызванной функции Sum состояние памяти с переменными программы представлено на рис.6.6: выделена память под переменные c, d и s, им присвоены переданные при вызове значения (s присвоено значение адреса a).

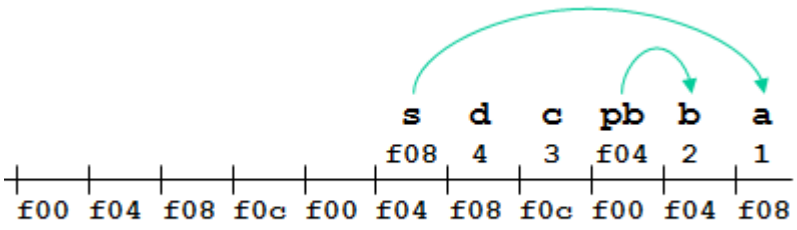


Рис. 6.6.

3. Шаг (3). При выполнении функции Sum разименованная s (т.е. a) получает новое значение (рис.6.7).

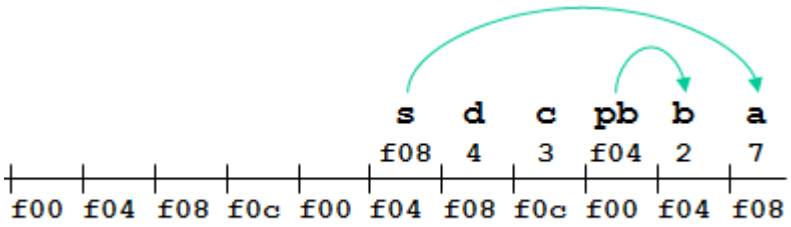


Рис. 6.7.

4. Шаг (4). При возврате в функцию main (рис.6.8) переменные c, d и s пропадают, а переменная a остается с полученным новым значением.

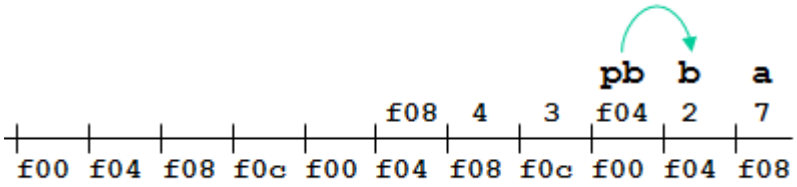


Рис. 6.8.

5. Шаг (5). Состояние памяти при повторном вызове функции Sum представлено на рис. 6.9. Вновь выделена память под переменные c, d и s, им присвоены переданные при вызове значения. Переменной s присвоено значение переменной pb, т.е. s как и pb указывает на b

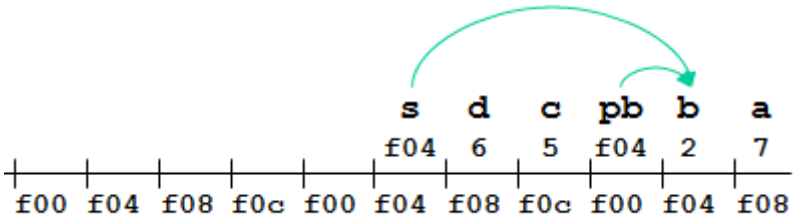


Рис. 6.9.

6. Шаг (6). При выполнении функции Sum разименованная s (т.е. b) получает новое значение (рис. 6.10).

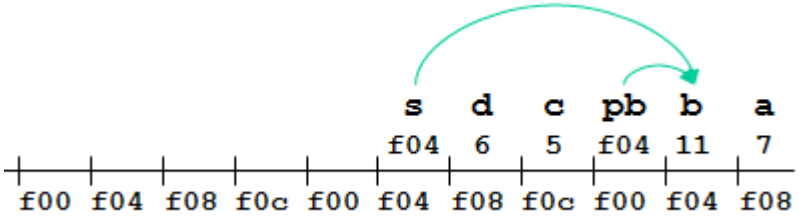


Рис. 6.10.

6.1.4. Передача параметра по ссылке (только в C++)

Ссылка на переменную

В C++ помимо указателей есть еще один тип данных — «ссылка на переменную»

Синтаксис объявления ссылки имеет вид:

<тип>& <идентификатор> = <инициализатор>

где:

<идентификатор> — идентификатор ссылки

<инициализатор> — идентификатор переменной, на которую ссылка ссылается

Ссылка на переменную — новая переменная, являющаяся псевдонимом переменной, на которую «ссылается ссылка». В отличие от указателя, который содержит адрес переменной, на которую он указывает, переменная типа ссылка имеет тот же адрес, что и переменная, на которую она ссылается.

Пример.

```
int i = 2, j = 3;
int& ri = i;    // ri – ссылка на i
ri = 8;         // i = 8
```

В приведенном примере фрагмента кода объявлены две переменные целого типа, инициированные начальными значениями и ссылка на целое ri, инициированная переменной i. Схема размещения объявленных переменных в памяти представлена на рис.11. Видно, что переменные i и ri «лежат» в одном месте и при присвоении переменной ri нового значения то же значение получает переменная i.

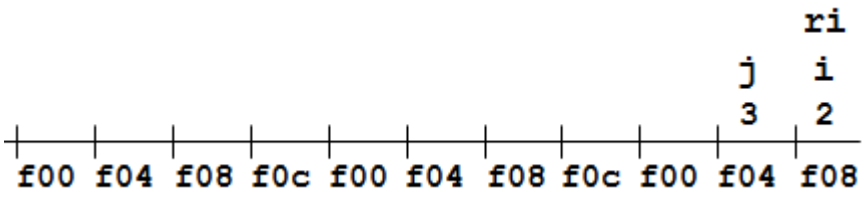


Рис. 6.11.

Важно! В отличие от указателя ссылка должна быть обязательно инициирована при объявлении и не может быть переинициирована на другую переменную.

Передача параметра по ссылке

При передаче параметра по ссылке:

- формальный параметр, через который будет возвращаться значение должен быть объявлен как ссылка;
- соответствующий фактический параметр не может быть константой или выражением, а должен быть переменной, значение которой будет изменено после возврата из функции;
- в функции возвращаемое значение должно быть присвоено значению формального параметра - ссылки.

Пример.

```
void Sum(int c, int d, int& s)
(2)
{
     // Присваиваем переменной
(3)  s = c + d;
}

int main()
{
     int a = 1;
(1)  Sum(3, 4, a); // Передаем переменную
     return 0;
}
```

Рассмотрим ситуацию по шагам:

1. Шаг (1). Перед вызовом функции Sum состояние памяти с переменными программы представлено на рис.6.12: выделена память под переменную a, ей присвоено значение 1.

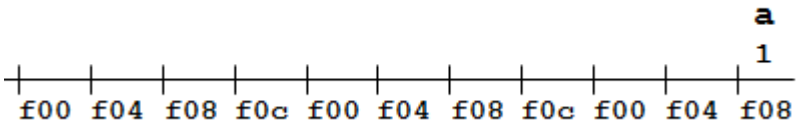


Рис. 6.12.

2. Шаг (2). Перед выполнением вызванной функции Sum состояние памяти с переменными программы представлено на рис.6.13: выделена память под переменные c и d, им присвоены переданные при вызове значения; переменная s «легла» на переменную a.

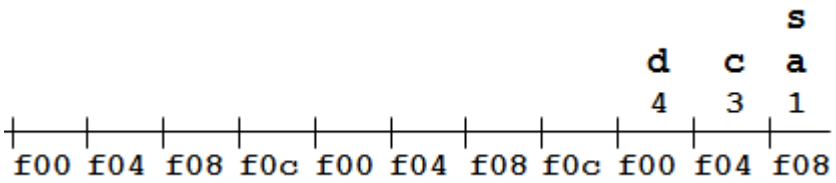


Рис. 6.13.

3. Шаг (3). При выполнении функции Sum переменная s (и переменная a) получает новое значение (рис.6.14).

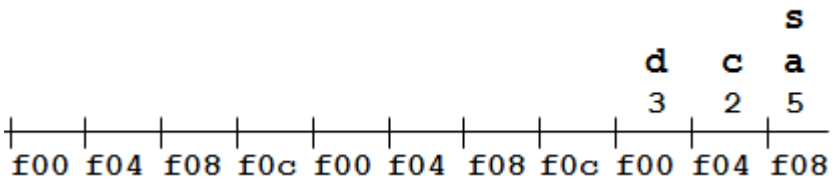


Рис. 6.14.

6.1.5. Особенности передачи параметров по указателю и ссылке

Передача параметров по указателю или ссылке применяется но только в том случае, когда через этот параметр надо вернуть рассчитанное в функции значение. Таким приемом пользуются также в случае, когда передаваемый параметр является конструкцией большого размера (структура, объединение, объект класса). При передаче такого параметра по значению:

- во-первых, тратится много памяти на соответствующий формальный параметр;
- во-вторых, тратится много времени на присвоение ему значения фактического параметра.

При передаче сложного параметра в функцию по указателю или ссылке в целях экономии памяти и времени в случае, когда он не является выходным, чревато тем, что функция, «забыв» об этом, может изменить значения полей этого параметра. Чтобы избежать такой ситуации, в функции такой формальный параметр объявляется как константный указатель или константная ссылка. В этом случае «забывчивость» функции будет диагностирована на этапе компиляции, что существенно упростит выявление такой ошибки.

6.2. Функция как параметр функции

6.2.1. Объявление функции-параметра в заголовке функции

Функция может быть формальным параметром другой функции. При этом, функция – формальный параметр может объявляться непосредственно в заголовке принимающей функции.

Пример.

```
int f2(double fun(double), double x1, double x2)
{
     if (fun(x1) > fun(x2))
          return 1;
     else
          return -1;
}
double f1(double x)
{
     return sin(x)*cos(x);
}
int main()
{
   double a = 1.1, b = 2.3;
   int rez = f2(f1, a, b);
}
```

В приведенном примере

- функция f2 в качестве первого параметра получает функцию с формальным именем fun, получающую один параметр типа double и возвращающую значение типа double;
- функция f1 получает один double и возвращает double, т.е. относится к тому типу, что и функция fun a f2;
- в функции main объявляются и инициируются начальными значениями две переменные типа double и выполняется вызов функции f2 с функцией f1 в качестве первого параметра.

При таком вызове в функции f2 в качестве функции fun будет вызываться функция f1.

6.2.2. Объявление типа функции

Наиболее часто функция – формальный параметр объявляется с использованием предварительно объявленного типа функции.

Пример.

```
typedef double tfun(double);
int f2(tfun fun, double x1, double x2)
{
     if (fun(x1) > fun(x2))
          return 1;
     else
          return -1;
}
double f1(double x)
{
     return sin(x)*cos(x);
}
int main()
{
   double a = 1.1, b = 2.3;
   int rez = f2(f1, a, b);
}
```

В первой строке приведенного примера объявляется новы тип данных с именем tfun, который является функций, принимающей один double и возвращающей double.

В функции f2 первый параметр объявлен как параметр типа tfun, т.е. функция, принимающая double и возвращающая double.

В остальном пример полностью соответствует предыдущему.

6.2.3. Пример

Написать функцию решения нелинейного алгебраического уравнения f(x) = 0 методом Ньютона.

Эта функция должна получать:

- функцию вычисления правой части уравнения f(x);
- начальное приближение решения x0;
- значение точности решения eps.

```
typedef double tfun1d(double);
```

```
double Root(tfun1d f, double x0, double eps)
{
  double dx  = eps*1.0e-4, df;
  double x = x0, fx = f(x);
  while ( fabs(fx) > eps )
  {
    df = (f(x + dx) - f(x - dx)) / (2.0 * dx);
    fx = f(x -= fx / df);
  }
  return x;
}
double f1(double x)
{
  return sin(x) * cos(x);
}
double f2(double x)
{
  return exp(x) - 1.5;
}

int main()
{
   double r1 = Root(f1, 1.0, 1.e-5);
   double r2 = Root(f2, 0.0, 1.e-7);
}
```

В приведенном примере:

1. Объявляется тип функции tfun1d, принимающей double и возвращающей double.
2. Описывается функция Root, получающая функцию типа tfun1d, начаольное приближение решения x0 и точность решения eps. В функции Root запрограммирован метод Ньютона (см. https://ru.wikipedia.org/wiki/%CC%E5%F2%EE%E4_%CD%FC%FE%F2%EE%ED%E0).
3. Далее описаны две функции типа tfun1d: f1 и f2.
4. В функции main выполняется вызов функции Root с функцией f1 (т.е. решается уравнение sin(x)*cos(x) = 0) и с функцией f2 (т.е. решается уравнение exp(x) – 1.5= 0).

7. СТРОКИ

7.1. Представление строк в языке С

7.1.1. Как представляются строки в языке С

Специального типа данных «строка» в языке С (и в языке С++) нет. Нет в смысле множества значений этого типа и допустимых операций.

Представление строк в языке С подчиняются следующим правилам:

1. Строки в языке С объявляются как массивы типа char:

```
char str[15];
```

2. Признаком конца строки является нулевой байт. При присвоении строке значения массив заполняется символами значения слева-направо, последним заносится нулевой байт (признак конца строки), далее — мусор:

В	о	в	а	+	Л	е	н	а	\0	??	??	??	??	??

3. При объявлении строка может быть инициирована начальным значением:

```
char str[15] = "Вова+Лена";
```

4. Операции (действия) над строками обеспечиваются с помощью специальных функций, которые программист может написать сам или воспользоваться стандартными библиотеками.

7.2. Библиотека string

Одной из наиболее известных стандартных библиотек работы со строками является библиотека string, которая присутствует во всех реализациях языка С.

Для подключения к библиотеке string надо выполнить команду:

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

7.2.1. Функция strcpy – копирование строки

Синтаксис:

```
char* strcpy(char d[], const char s[]);
```

Описание. Строка s копируется в строку d. Функция возвращает указатель на сороку d.

Пример.

```
char str[21];
strcpy(str, "Лена");
```

В приведенном примере строка `"Лена"` будет скопирована с строку str.

У функции strcpy есть один существенный недостаток. Если строка s окажется «длиннее» строки d, то функция strcpy этого «не замечает» и копирует в d всю строку s, затирая «чужую» память. Эта проблема называется проблемой переполнения буфера (принимающей строки) и она в той или иной мере свойственна всем С-функциям работы со строками.

7.2.2. Функция strncpy – копирование строки с контролем переполнения буфера

Синтаксис:

```
char* strncpy(char d[], const char s[], int n);
```

Описание. Из строки s в строку d копируется не более чем n символов. Функция возвращает указатель на сороку d.

Пример.

```
char str[21];
strcpy(str, "Лена", 20);
```

В приведенном примере строка `"Лена"` будет скопирована с строку str.

У функции strncpy есть тоже есть существенный недостаток. Если строка s окажется «длиннее» строки d, то функция strcpy копирует в d только n первых символов строки s, но ноль байт конца строки не записывает. Это тоже одно из проявлений проблемы переполнения буфера.

7.2.3. Функция strlen – определение длины строки

Синтаксис:

```
int strlen(const char str[]);
```

Описание. Функция возвращает количество символов в строке str.

Пример.

```
int len = strlen("Вова = ?");
```

В приведенном примере переменная len получит значение 8.

7.2.4. Функция strcat – конкатенация (объединение) двух строк

Синтаксис:

```
char* strcat(char d[], const char s[]);
```

Описание. Строка s приписывается справа к строке d. Функция возвращает указатель на сороку d.

Пример.

```
char str[21] = "Вова+";
strcat(str, "Лена");
```

В приведенном примере строка `"Лена"` будет приписана к строке str.

У функции strcat есть тот же недостаток, что и у функции strcpy. Если объединение двух строк окажется «длиннее» строки d, то функция strcat этого «не замечает» и приписывает к d всю строку s, затирая «чужую» память.

7.2.5. Функция strncat – конкатенация (объединение) двух строк с контролем переполнения буфера

Синтаксис:

```
char* strncat(char d[], const char s[], int n);
```

Описание. Из строки s в строку d приписывается справа не более чем n символов. Функция возвращает указатель на сороку d.

Пример.

```
char str[10] = "Вова+";
strcat(str, "Лена, Оля, Маша", 4);
```

В приведенном примере к строке `"Вова+"` припишется строка `"Лена"`.

У функции strncat есть тоn же недостаток, что и функции strncpy. Если строка s окажется «длиннее» остатка строки d, то функция strcat припишет к d только n первых символов строки s, но ноль байт конца строки не записывает.

7.2.6. Полная сводка функций библиотеки string

В предыдущих разделах были описаны лишь часть функций библиотеки string. Полная сводка функций этой библиотеки приведена в таблице 7.1.

Таблица 7.1. Функции библиотеки string.

Функция	***Комментарий***
char *strcpy(char *s1, char *s2)	копирует строку s2 в s1. Возвращает s1.
char *strncpy(char *s1, char *s2, size_t n)	копирует строку s2 в s1, но копируется не более n символов. Возвращает s1.
char *strcat(char *s1, char *s2)	объединяет строки s1 и s2 (дописывает s2 в s1). Возвращает s1.
char *strncat(char *s1, char *s2, size_t n)	объединяет строки s1 и s2, но дописывает не более n символов. Возвращает s1.
int strcmp(char *s1, char *s2)	сравнивает строки (содержимое, не указатели), расставляя их в лексикографическом порядке. Возвращает 0 для совпадающих строк, отрицательное значение при s1<s2 и положительное при s1>s2.
int strncmp(char *s1, char *s2, size_t n)	тоже сравнивает строки, но берет из них для сравнения не более n первых символов.
char *strchr(char *s, int c)	возвращает указатель на первый встреченный в строке символ c. Если такого символа в строке не оказалось, возвращает NULL.
char *strrchr(char *s, int c)	похожа на strchr(), но возвращает указатель на последний символ c в строке.
char *strstr(char *s1, char *s2)	возвращает указатель на первую встреченную в строке s1 подстроку s2. Если подстроки не нашлось, возвращает NULL.
size_t strlen(char *str)	возвращает длину строки.

char *strerror(size_t n)	возвращает строку сообщения, соответствующего ошибке с номером n.
void *memcpy(void *dst, void *src, size_t len)	копирует len байтов (включая нулевые) из src в dst. Возвращает dst.
void *memove(void *dst, void *src, size_t len)	делает то же, что и memcpy. Это - единственная функция, которая по стандарту обязана правильно копировать перекрывающиеся объекты.
int memcmp(void *s1, void *s2, size_t len)	аналог strcmp, но с учетом нулевых байтов.
void *memchr(void *s, int c, size_t len)	аналог strchr, но с учетом нулевых байтов.
void *memset(void *s, int c, size_t len)	заполняет первые len байтов массива s символом c.

7.3. Функции работы со строками

В этом разделе мы разберем несколько примеров функций работы со строками.

7.3.1. Определение длины строки

Итак, пусть вам надо написать «свою» функцию определения длины строки. Сделать это можно несколькими способами (вариантами).

Вариант 1

```
int StrLen1(const char* str)
{
    int length;
    for ( length = 0; str[length]; length++ ) ;
    return length;
}
```

Комментарии:

1. Заголовок функции мог быть записан так:

```
int StrLen1(char* str)
```

или так:

```
int StrLen1(char[] str)
```

Но более правильный вариант тот, что приведен в основном тексте. Объявление str как константный указатель гарантирует, что функция не будет изменять эту строку.

2. В приведенном варианте используется цикл for. Переменная length играет роль счетчика цикла; цикл выполняется до тех пор, пока очередной символ не станет нулевым (признак конца строки). Тела цикла нет – счетчик цикла (length) увеличивается по правилу выполнения цикла for.

Вариант 2

Использование цикла for в программах такого типа является корректным, но все же экзотичным. Более употребимым является использование цикла while:

```
int StrLen2(const char* str)
{
    int length = 0;
    while ( str[length++] );
    return length - 1;
}
```

Комментарий. Увеличение длина строки внесено в проверку условия выхода из цикла. В результате, при выходе из цикла переменная length показывает длину строки с учетом нулевого байта. Поэтому в качестве результата возвращается значение этой переменной, уменьшенное на единицу.

Вариант 3

При обработке строк обычно используют указатели:

```
int StrLen3(const char* str)
{
    const char* p = str;
    while ( *p++ );
    return p - str - 1;
}
```

Комментарии:

1. Объявляется рабочий указатель p, которому присваивается значение str.
2. В цикле while указатель p передвигается на конец строки (на нулевой байт).
3. Возвращается значение разности указателей p и str, уменьшенное на единицу.
4. Преимущество использования указателей состоит в том, что в цикле while здесь используется одна операция (увеличение указателя на единицу), а

цикле while предыдущего варианта – две (индексации и увеличение индекса на единицу).

7.3.2. Удаление символа в строке

Функция удаления символа в строке тоже может быть написана по-разному. Рассмотрим два «указательных» варианта.

Вариант 1

```
int DelSymbol1(char *str, char sym)
{
  char *p1 = str, *p2 = str;
  while (*p1)
    if (*p1 == sym) p1++; else *p2++ = *p1++;
  *p2 = '\0';
  return (p1 - p2);
}
```

Комментарии:

1. Функция DelSymbol1 получает строку str и символ sym, который из нее надо удалить; возвращает количество удаленных символов.
2. Объявляются два рабочих указателя p1 и p2, которые устанавливаются на начало строки str.
3. С помощью указателя p1 проходим по строке str (цикл while)и на каждом шаге проверяем, не указывает ли p1 на символ, который надо удалить

- если да, то передвигаем p1 на шаг вперед;
- если нет, то символу, на который указывает p2 присваиваем символ, на который указывает p1 и оба указателя передвигаем на шаг вперед.

4. По завершению цикла символу, на который указывает p2 присваиваем значение нуля.
5. Функция возвращает значение разности указателей p1 и p2.

Вариант 2

Более компактно (и изящно) выглядит вариант с использованием цикла do:

```
int DelSymbol2(char *str, char sym)
{
  char *p1 = str, *p2 = str;
  do
```

```
  {
     if ( *p1 != sym ) *p2++ = *p1;
  } while (*p1++);

  return (p1 - p2);
}
```

Комментарии:

6. Здесь в цикле проверяется, указывает ли p1 на неудаляемый символ и если да, то символу, на который указывает p2 присваиваем символ, на который указывает p1 и только p2 передвигаем на шаг вперед.
7. Указатель p1 передвигаем при проверке выхода из цикла.
8. Здесь по завершению цикла уже не надо присваивать ноль разименованному p2, т.к. это уже было сделано в цикле.

7.3.3. Создание копии строки

Одним из способов решения проблемы переполнения буфера при копировании строки является написание функции, создающей копию строки:

```
char* CreateACopy(const char* source)
{
  char* rez = (char*)malloc((StrLen(source) + 1)*sizeof(char));
  char* p1 = rez;
  const char* p2 = source;
  while ( *p1++ = *p2++ );
  return rez;
}
```

Комментарии:

1. Функция CreateACopy получает копируемую строку source и возвращает указатель на созданную этой функцией копию строки source.
2. Для этого:

- объявляется rez - указатель на char, под который в куче заказывается память необходимой длины (длина строки source плюс единица под нулевой байт);
- объявляются два рабочих указателя: p1 (указывает на rez) и p2 (указывает на source);
- с помощью этих указателей строка source копируется в строку rez;
- возвращается значение указателя rez, который сейчас указывает на созданную в куче копию строки source.

3. Функция CreateACopy создаст копию строки любой длины, но она имеет существенный недостаток: клиент этой функции (та функция, которая будет ее использовать несет ответственность за освобождение памяти, занимаемой созданной копией строки.

7.3.4. Копирование подстроки

Написать функцию, копирующую часть заданной строки:

```
void StrSub(char* dest, const char* source, int pos, int count)
{
     int cnt = 0;         // Количество скопированных символов
     while ( cnt < count )
     {
          dest[cnt] = source[pos + cnt++];
     }
    dest[cnt] = '\0';
}
```

Комментарии:

1. Функция `StrSub` получает:

- копируемую строку source;
- строку dest, в которую надо скопировать часть строки source;
- pos - номер позиции строки source, начиная с которой надо выполнять копирование;
- count – количество копируемых символов.

2. Хочется надеяться, что описанный в функции алгоритм решения задачи в комментариях не нуждается.
3. Описанный алгоритм имеет ряд существенных недостатков:

- что будет, если значение pos превышает длину строки source?
- что будет, если значение pos+count превышает длину строки source?
- что будет, если значение pos или count будет отрицательным?

В качестве самостоятельной работы Вам предлагается устранить эти недостатки.

7.3.5. Поиск подстроки

Написать функцию, выполняющую поиск заданной подстроки в заданной строке.

```
// Поиск подстроки substr в строке str
int StrPos(const char* str, const char* substr)
{
     int rez = -1;
     const char *p1 = str, *p2;
     const char *pf = str + StrLen(str) - StrLen(substr) + 1,
*pss;
     while ( p1 < pf && rez == -1 ) {
          p2 = p1;
          rez = p1 - str;
          pss = substr;
          while ( *pss && rez != -1 )
               if ( *pss++ != *p2++ ) rez = -1;
          p1++;
     }
    return rez;
}
```

Комментарии:

1. Функция `StrPos` получает строку str, в которой надо найти строку strsub. В качестве результата возвращает номер позиции строки str, начиная с которой в str содержится strsub или -1, если strsub в str нет.
2. Объявляется целая переменная rez – возвращаемый результат, которому по умолчанию присваивается значение -1 (не нашли).
3. Объявляются указатели на char:
    a. p1 – указывает на str; с его помощью будм просматривать строку str;
    b. p2 – рабочий;
    c. pf – указывает на символ строки str. дальше которого можно не искать;
    d. pss – рабочий.
4. Далее в цикле while пока p1 не дошел до конца просмотра строки str и вхождение не найдено (rez равно -1):
    a. p2 устанавливается на p1; rez присваивается значение возможного ответа; pss устанавливается на substr;
    b. в цикле while пока pss не дошел до конца подстроки и rez не -1: если очередной символ pss не равен очередному символу p2, то rez присвоить -1

7.4. Ввод/вывод строк в С

7.4.1. Вывод на консоль. Функция printf

Функция printf

Для вывода на консоль в языке С надо:

1. Подключить библиотеку стандартного ввода/вывода:

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

2. Использовать для вывода функцию форматного вывода на консоль:

```
int printf(const char* format [, arg1, agr2,..]);
```

где:

- `format` – формат преобразования вывода;
- `- [, arg1, agr2,..]` – список вывода;

Функция `printf` возвращает количество выведенных символов.

Элементом списка вывода является выражение (константа, переменная, элемент массива, …).

Формат преобразования вывода представляет из себя текст с включенными в него спецификаторами вывода. Текст формата выводится на консоль, а спецификаторы используются для форматных преобразований соответствующего спецификатору элементу списка вывода. Каждому спецификатору должен соответствовать свой элемент списка вывода.

Запись спецификатора вывода начинается с символа ‘%’ и имеет вид:

```
%[flg][wid][.pre][F|N|h|l|L]type_char
```

где:

- `flg` – флаг:

  - (минус) – разместить в левой части поля

  + (плюс) – знаковое выводить со знаком (+ или -)

- `wid` – ширина (к-во символов) поля, выделяемого для элемента вывода;
- `pre` – точность

- `F|N|h|l|L` – модификаторы :

  F – дальний (far) указатель;

  N – близкий (near) указатель;

  h – число типа short int

  l, L – число типа long

- `type_char` – один символ, определяющий тип преобразования:

	Символ	G	F или E, что короче
	целое десятичное число	o	восьмеричное число
	целое десятичное число	s	строка символов
	десятичное в виде x.xx e+xx	u	беззнаковое целое число
	десятичное в виде x.xx E+xx	x	шестнадцатеричное число
	десятичное в виде xx.xxxx	X	шестнадцатеричное число
	десятичное в виде xx.xxxx	%	символ %
	f или e, что короче	p(n)	указатель

Примеры

Пример 1

Вариант 1:

```
int a = 8;
double r = 1.1;
printf("%d", 10);
printf("%s", "Hello!");
printf("a = %d", a);
printf("%f", 2 * 3.14 * r);
```

Результат выполнения приведенного фрагмента кода будет выглядеть так:

```
10Hello!a = 86.908000
```

Для перехода на новую строку в формате надо указывать “\n”:

Вариант 2:

```
int a = 8;
double r = 1.1;
printf("%4d\n", 10);
printf("%s", "Hello!\n");
printf("a = %3d\n", a);
printf("%5.3f\n", 2 * 3.14 * r);
```

Результат выполнения приведенного фрагмента кода будет выглядеть уже так:

```
  10
Hello!
a =   8
6.908
```

Пример 2

Для использования кириллиц в выводимой на консоль информации надо:

1. Подключить специальную библиотеку:

   ```
   #include <locale.h>  // подключение библиотеки использования
   ```

2. В начале функции main вызвать функцию разрешающую использование кириллиц:

   ```
   setlocale(LC_CTYPE, "Russian"); // использование кириллиц
   ```

Если этого не сделать, то на консоль будут выводиться «зюковки».

```
#include <stdio.h>
#include <string.h>
#include <locale.h> // подключение библиотеки использования кирил-
лиц

int main()
{
   double b, c;

   setlocale(LC_CTYPE, "Russian"); // использование кириллиц

   b = 12.89;
   c = 4.12;
   printf("b = %6.2f, c = %6.2f\nИх сумма = %6.2f, их произведение
= %6.2f\n", b, c, b + c, b * c);
   return 0;
```

```
}
```

Результат выполнения приведенного фрагмента кода будет выглядеть так:

```
b = 12.89, c = 4.12
Их сумма =  17.01, их произведение = 53.11
```

7.4.2. Ввод с консоли. Функция scanf

Функция scanf

Для ввода с консоли в языке С надо:

1. Подключить библиотеку стандартного ввода/вывода:

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

2. Использовать для ввода функцию форматного ввода с консоли:

```
int scanf(const char* format [, arg1, agr2,..]);
```

где:

- `format` – формат преобразования ввода;
- `- [, arg1, agr2,..]` – список ввода;

Функция `scanf` возвращает количество введенных символов.

Элементом списка вывода является адрес переменной или указатель на переменную, в которую выполняется ввод значения.

Как и в случае функции `printf`, формат преобразования вывода представляет из себя текст с включенными в него спецификаторами ввода. Каждому спецификатору должен соответствовать свой элемент списка ввода.

Запись спецификатора ввода начинается с символа ‘%’ и имеет вид:

```
%[*][wid][F|N|h|l|L]type_char
```

где:

- `*` – пропуск поля ввода;
- `wid` – ширина (к-во символов), занимаемых элементом ввода;
- `F|N|h|l|L` – модификаторы :

  F – дальний (far) указатель;

  N – близкий (near) указатель;

h – число типа short int

l, L – число типа long

- `type_char` – один символ, определяющий тип преобразования:

c	символ	h	short int
d	целое десятичное число	o	восьмеричное число
i	целое десятичное число	s	строка символов
e,f,g	десятичное float	x	шестнадцатеричное число
le, lf, lg	десятичное double	p(n)	указатель

Каждому элементу списка ввода должно соответствовать свое поле ввода в вводимой с консоли строки. Указывать поля ввода можно различными способами.

3. С помощью параметра wid спецификатора ввода можно указать, сколько очередных символов вводимой строки занимает поле ввода соответствующего элемента списка ввода.

Пример

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

int main()
{
   int a, b, c;

   setlocale(LC_CTYPE, "Russian"); // использование кириллиц

   printf("Введите значения a, b и c:");
   scanf("%1d%2d%2d", &a, &b, &c);
   printf("a = %d, b = %d, c = %d", a, b, c);

   return 0;
}
```

Результат выполнения приведенного фрагмента кода будет выглядеть так:

```
Введите значения a, b и c: 12345
a = 1, b = 23, c = 45
```

4. Между спецификаторами ввода может быть указан символ, являющийся разделителем полей ввода.

Пример

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

int main()
{
   int a, b, c;

   setlocale(LC_CTYPE, "Russian"); // использование кириллиц

   printf("Введите значения a, b и c:");
   scanf("%d %d %d", &a, &b, &c);
   printf("a = %d, b = %d, c = %d", a, b, c);

   return 0;
}
```

Результат выполнения приведенного фрагмента кода будет выглядеть так:

```
Введите значения a, b и c: 1 23 45
a = 1, b = 23, c = 45
```

В качестве разделителя полей можно использовать любой символ.

Пример

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

int main()
{
   int day, year, month;

   setlocale(LC_CTYPE, "Russian"); // использование кириллиц

   printf("Введите дату: ");
   scanf("%d.%d.%d", &day, &month, &year);
   printf("Дата: %4d/%2d/%2d", year, month, day);

   return 0;
}
```

Результат выполнения приведенного фрагмента кода будет выглядеть так:

```
Введите дату: 1.12.2014
Дата: 2014/12/ 1
```

Если в спецификаторе ввода после символа ‘%’ поставить символ ‘*’, то соответствующее поле будет пропущено.

Пример

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

int main()
{
   int a, b, c;

   setlocale(LC_CTYPE, "Russian"); // использование кириллиц

   printf("Введите значения a, b и c:");
   scanf("%d %*d %d %d", &a, &b, &c);
   printf("a = %d, b = %d, c = %d", a, b, c);

   return 0;
}
```

Результат выполнения приведенного фрагмента кода будет выглядеть так:

```
Введите значения a, b и c: 1 23 45 67
a = 1, b = 45, c = 67
```

Ввод строк

Функция scanf вводит строку до первого пробела.

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

int main()
{
   char str[100];

   setlocale(LC_CTYPE, "Russian"); // использование кириллиц

   printf("Введите строку:");
   scanf("%s", str);
   printf("Строка: %s", str);

   return 0;
}
```

Результат выполнения приведенного фрагмента кода будет выглядеть так:

```
Введите строку:Вова + Лена
```

```
Строка: Вова
```

Для правильного ввода строки надо использовать функцию gets:

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

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

int main()
{
   char str[100];

   setlocale(LC_CTYPE, "Russian"); // использование кириллиц

   printf("Введите строку:");
   gets(str);
   printf("Строка: %s", str);

   return 0;
}
```

Результат выполнения приведенного фрагмента кода будет выглядеть уже так:

```
Введите строку:Вова + Лена
Строка: Вова + Лена
```

Внимание! Функция gets не контролирует переполнение буфера.

7.5. Текстовые файлы в С

7.5.1. Структура текстового файла

Текстовый файл является последовательностью строк, разделенных признаком (символом) <конец_строки>. Каждая строка является последовательностью слов, разделенных пробелами. Заканчивается текстовый файл специальным признаком (символом) EOF (конец файла).

7.5.2. Вывод/ввод в текстовый файл

Общие правила

Для использования функций работы с файлами необходимо подключить библиотеку:

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

Для записи данных в текстовый файл и чтения данных из текстового файла надо выполнить следующие действия:

1. Объявить файловую переменную:

```
FILE* f;
```

2. Открыть файл:

```
f = fopen(const char* filename, const char* mode);
```

3. Выполнить запись/чтение данных с помощью функций fprintf и fscanf (см. ниже).
4. Закрыть файловую переменную:

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

Открытие файла

Функция открытия файла fopen получает два параметра:

- `filename` - имя файла;
- `mode` - режим открытия файла.

В качестве режима открытия файла можно указывать:

- rt – открыть файл на чтение;
- wt – открыть файл на запись с начала файла;
- at – открыть файл на добавление в конец файла;

Функция open возвращает указатель на открытый файл или 0 (нулевой указатель) при невозможности открыть файл. Контроль правильности открытия файла может быть выполнен так

```
FILE* f;
if( !(f = fopen("filename.txt", "rt")) )
{
   printf("Can't open file 'filename.txt'\n");
   return 1;
}
```

Запись/чтение данных

Запись данных в файл выполняется с помощью функции fprintf:

```
int fprintf(FILE* f, const char* format [, arg1, agr2,..]);
```

Чтение данных из файла выполняется с помощью функции fscanf:

```
int fscanf(FILE* f, const char* format [, arg1, agr2,..]);
```

Работа с этими функциями полностью аналогична работе с функциями printf и scanf.

Закрытие файла

Функция закрытия файла fclose получает закрываемую файловую переменную и возвращает 0, если файл был успешно закрыт и EOF (-1) в противном случае. Контроль правильности открытия файла может быть выполнен так

```
if(!fclose(f))
{
   printf("Can't close file\n");
   return 3;
}
```

Другие полезные функции

1. Проверка конца файла

```
int feof(FILE* file);
```

Функция feof возвращает 0, если в файле есть еще записи или EOF в противном случае.

2. Запись/чтение символа

```
int fputc(int ch, FILE* file);
int fgetc(FILE* file);
```

Функции fputc и fgetc возвращают EOF при ошибке.

3. Чтение строки

Неприятность чтения строки с помощью функции fscanf состоит в том, что строка читается не до конца строки, а до первого разделителя слов строки (пробела, табуляции, …).

```
char* fgets(char* buf, int n, FILE* file);
```

где:

buf – строка (буфер), в которую будет выполняться чтение;

n – длина буфера buf;

file – файловая переменная

Функция fgets:

- читает из file в буфер buf очередную строку до признака конца строки или до переполнения буфера buf;
- возвращает указатель на прочитанную строку (buf) или 0 в случае ошибки.

7.5.3. Примеры

Пример 1. Запись и чтение числовых данных

```
int main()
{
   int a, b, c;
   int n, i;
   double element;
   FILE* f;

   setlocale(LC_CTYPE, "Russian");

   f = fopen("D:\\TMP\\data.txt", "wt");

   printf("a, d, c = ");
   scanf("%d %d %d", &a, &b, &c);
   fprintf(f, "%d %d %d\n", a, b, c);

   printf("Количество элементов массива = ");
   scanf("%d", &n);
   fprintf(f, "%d\n", n);
   printf("Введите элементы массива: ");
   for (i = 0; i < n; ++i)
   {
     b = printf("%d) ", i);
     a = scanf("%le", &element);
     c = fprintf(f, "%f\n", element);
   }

   fclose(f); // закрытие файла

   return 0;
}
```

В приведенном примере:

1. Объявляется файловая переменная f, в которой на запись открывается `D:\\TMP\\data.txt`.
2. С консоли вводятся значения ранее объявленных переменные целого типа a,b и c. Значения этих переменных записываются в файл.

3. С консоли вводится значение переменной n –количество элементов массива, которое также записывается в файл.
4. Затем в цикле с консоли вводится очередной элемент массива и записывается в файл.
5. В конце выполняется закрытие файла.

Приведенный ниже пример читает из текстового файла данные, записанные программой, приведенной выше.

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

int main()
{

  int e, f, g;
  int m, i;
  double* ar;
  FILE* file;

  setlocale(LC_CTYPE, "Russian");

  file = fopen("D:\\TMP\\data.txt", "rt");

  fscanf(file, "%d %d %d", &e, &f, &g);
  printf("e, f, g = %d, %d, %d\n", e, f, g);

  fscanf(file, "%d", &m);
  printf("Количество элементов массива = %d\n", m);
  ar = (double*)malloc(sizeof(double)*m);
  for (i = 0; i < m; ++i)
  {
    fscanf(file, "%le", &ar[i]);
  }
  for (i = 0; i < m; ++i)
  {
    printf("%d) %f\n", i, ar[i]);
  }

  fclose(file);  // закрытие файла

  return 0;
}
```

В приведенном примере:

1. Объявляется файловая переменная file, в которой на чтение открывается файл `D:\\TMP\\data.txt`.
2. Из файла читаются значения объявленных ранее переменных целого типа: e, f и g, значения которых далее выводятся на консоль.
3. Из файла читается значение переменной m –количество элементов массива, которое выводится на консоль.
4. Объявляется ar – указатель на double, под который заказывается память длинной m элементов типа double.
5. Затем в цикле из файла вводятся элемента массива.
6. Затем в цикле элементы введенного массива выводятся на консоль.
7. В конце выполняется закрытие файла.

Пример 2

Написать функцию записи в текстовый файл массива типа double. Функция должна принимать:

- имя файла;
- указатель на массив;
- длину массива.

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

void writeDArrayToFile(const char* fileName, const double* arr,
const int arrCount)
{
  int i;
  FILE* f;

  f = fopen(fileName, "wt");
  fprintf(f, "%d\n", arrCount);

  for (i = 0; i < arrCount; ++i)
  {
    fprintf(f, "%e\n", arr[i]);
  }

  fclose(f);  // закрытие файла
}
```

```
int main()
{
  int n, i;
  double* mas;

  setlocale(LC_CTYPE, "Russian");

  printf("Количество элементов массива = ");
  scanf("%d", &n);
  mas = (double*)malloc(n*sizeof(double));
  printf("Введите элементы массива: ");
  for (i = 0; i < n; ++i)
  {
    printf("%d) ", i);
    scanf("%le", &mas[i]);
  }
  // Запись элементов массива в файл
  writeDArrayToFile("D:\\TMP\\data.txt", mas, n);

  return 0;
}
```

В приведенном примере:

1. В функции `writeDArrayToFile` :
    a. Объявляется файловая переменная f , в которой на запись открывается файл, в который надо записать массив.
    b. В файл записывается длина массива.
    c. В цикле в файл записываются элементы массива.
    d. Выполняется закрытие файла.
2. В функции `main` :
    a. Объявляется переменная n – длина массива, с консоли вводится ее значение.
    b. Объявляется mas - указатель на double, под который заказывается память из n элементов.
    c. С консоли вводятся значения элементов массива.
    d. Выполняется вызов функции `writeDArrayToFile`.

Написать функцию чтения из текстового файла массива типа double. Функция должна принимать имя файла; возвращать указатель на прочитанный массив и длину прочитанного массива.

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

double* readDArrayFromFile(const char* fileName, int* arrCount)
{
  int i;
  double* arr;

  FILE* f = fopen(fileName, "rt");
  fscanf(f, "%d", arrCount);
  arr = (double*)malloc(*arrCount*sizeof(double));
  for (i = 0; i < *arrCount; ++i)
  {
    fscanf(f, "%le", &arr[i]);
  }
  fclose(f);  // закрытие файла

  return arr;
}

int main()
{
  int n, i;
  double* mas;
  setlocale(LC_CTYPE, "Russian");

  // Чтение элементов массива
  mas = readDArrayFromFile("D:\\TMP\\data.txt", &n);

  printf("Элементы массива: \n");
  for (i = 0; i < n; ++i)
  {
    printf("%d) %f\n", i, mas[i]);
  }

  free(mas);

  return 0;
}
```

В приведенном примере:

1. Функция `readDArrayFromFile` получает имя файла, из которого надо прочитать массив, указатель на длину этого массива, в качестве результата возвращает указатель на double, который будет указывать на прочитанный массив.

2. В функции `readDArrayFromFile`:
    a. Объявляется файловая переменная f, в которой на чтение открывается файл, из которого надо прочитать массив.
    b. Из файла читается значение длины массива.
    c. Объявляется arr - указатель на double, под который заказывается память из arrCount элементов.
    d. В цикле из файла читаются значения элементов массива.
    e. Выполняется закрытие файла.
3. В функции `main` :
    a. Объявляется переменная n – длина массива и переменная mas – указатель на double.
    b. Выполняется вызов функции `readDArrayFromFile`.
    c. На консоль вводятся значения элементов массива.
    d. Освобождается память, занятая массивом mas.

Пример 3

В текстовом файле записана книга – набор неизвестного количества строк текста. Написать функцию удаления из этой книги лишних пробелов, т.е. замены двух и более подряд стоящих пробелов одним пробелом. Функция должна принимать имя исходного файла и имя файла, в который должен быть записан результат.

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

// Удаление лишних пробелов
void delBlanks(const char* inFileName, const char* outFileName)
{
   char s1[5000], s2[5000];   // буферы чтения и записи строк кни-
ги
   char *p1, *p2;
   int bCount;
   int rez;

   FILE *fIn, *fOut;
   fIn = fopen(inFileName, "rt"); // открытие файла для чтения
   fOut = fopen(outFileName, "wt"); // открытие файла для записи

   while ( !feof(fIn) ) {   // пока не дошли до конца файла
```

```
        rez = fgets(s1, 200, fIn); // читаем очередную строку
         if ( rez == 0 )        // А было ли что-то прочитано?
              continue;          // нет
        // Удаление лишних пробелов в строке
        p1 = s1; p2 = s2;
        bCount = 0;
        while ( *p1 )
        {
           if ( *p1 == ' ' ) bCount++; else bCount = 0;
           if ( bCount <= 1 )*p2++ = *p1;
           p1++;
        }
         *p2 = '\0';
        fprintf(fOut, "%s", s2);    // запись очередной строки
    }

    fclose(fIn);    // закрытие файла чтения
    fclose(fOut);    // закрытие файла записи
}
```

В приведенном примере:

1. Функция `delBlanks` получает имя файла, из которого надо прочитать исходную книгу и имя файла, в который надо записать результат
2. Объявляются две строки: s1, в которую будут последовательно читаться строки исходной книги, и s2, в которую будет переписываться s1 с удаленными лишними пробелами.
3. Объявляются два рабочих указателя на char: p1 и p2, с помощью которых будут удаляться лишние пробелы.
4. Объявляются файловая переменная fIn, в которой открывается на чтение исходная книга и файловый поток fOut, в которой открывается результирующий файл.
5. Далее в цикле while до тех пор пока во входном потоке есть строки:
    a. В s1 читается очередная строка.
    b. Выполняется удаление лишних пробелов с записью результата в s2.
    c. Строка s2 записывается в выходной файл.
6. Выполняется закрытие файлов.

8. СТРУКТУРЫ

8.1. Структуры

8.1.1. Определение структуры

В отличие от массива, который является набором элементов одного типа, каждый из которых имеет свой порядковый номер (индекс), структура это:

- набор элементов различных типов (членов или полей структуры);
- каждый член структуры имеет свое имя.

При работе со структурами мы объявляем тип структуры и переменные (экземпляры) структур этого типа. В объявлении типа структуры объявляются:

- имя типа структуры;
- ее члены – их имена и типы.

Синтаксис объявления типа структуры имеет вид :

```
struct [<имя_типа>]
{
  <описание_членов>
} [<список_переменных>];
```

где:

- `<имя_типа>` - имя типа структуры;
- `<описание_членов>` - описание членов структуры;
- `<список_переменных>` - необязательный при объявлении типа список переменных структуры.

Пример:

```
struct Friend // Friend – имя типа структуры
{
  char name[20];
  int age;
} Sam, July;  // Sam, July - имена переменных типа Friend
```

В приведенном примере:

1. Объявляется тип структуры `Friend`.

2. Объявляется, что этот тип структуры содержит два члена:
    a. name – массив типа char (имя товарища);
    b. age – переменная типа int – возраст товарища.
3. Объявляются две переменные типа Friend: Sam и July.

После объявления типа структуры могут быть объявлены переменные этого типа. Синтаксис объявления переменной типа структуры имеет вид:

```
struct <имя_типа> name1, name2, …;
```

где:

- <имя_типа> - имя типа структуры;
- name1, name2, … - имена объявляемых переменных типа структуры.

Пример:

```
struct Friend Bob, Mike;
```

При объявлении переменной типа структуры ее поля можно инициировать:

```
struct Friend Bob = {“Bob”, 19};
```

В языке C++ при объявлении переменной типа структуры ключевое слово `struct` можно опускать. В языке C можно использовать синоним типа, который позволяет не указывать ключевое слово `struct`:

```
struct Friend { char name[20]; int  age; };
typedef struct Friend MyFriend; // MyFriend – синоним struct
Friend
struct Friend Sam, Bob; // Объявление через структуру
MyFriend Jane;          // Объявление через синоним
MyFriend Mike = {"Mike", 23}; // Объявление через синоним с иници-
ализацией
```

8.1.2. Операции со структурами

Операция присваивания

Для переменных типа структура определена операция присваивания:

```
struct Friend Mike = {"Mike", 23};
struct Friend Bob, Mike = {"Mike", 23};
Bob = Mike;
```

Операция присваивания переменных структур имеет следующие особенности:

1. Выполняется последовательное присваивание значений полей.
2. Если какое-то поле является указателем на массив, то после присваивания обе структуры будут указывать на один и тот же массив.

Работа с членами структур

Обращение к членам структуры выполняется с помощью операции «.» прямой выбор члена: <имя_переменной>.<член_структуры>

Пример:

```
struct Friend Sam, Bob;
Bob.age = 19;
StrCopy(Bob.name, 19, "Bob");
```

В приведенном примере полю age структуры Bob присваивается значение 19, а в поле name копируется строка “Bob”.

Обращение к членам структуры через указатель на структуру выполняется с помощью операции: «->» (косвенный выбор члена): <имя_указателя_на_структуру>-><член_структуры>

Пример:

```
MyFriend *Jane;          // Объявление через синоним
struct Friend *Mike;     // Объявление через структуру
Jane = (MyFriend*)malloc(sizeof(struct Friend));
Mike = (MyFriend*)malloc(sizeof(struct Friend));
Jane->age = 17;
(*Mike).age = 20;
StrCopy((*Jane).name, 19, "Jane");
StrCopy(Mike->name, 19, "Mike");
```

В приведенном примере:

1. Объявляются два указателя на структуру Jane и Mike.
2. В куче под них запрашивается память.
3. Полю age структуры Jane присваивается значение 17, а полю age разименованного Mike – значение 20.
4. Аналогичным образом копируются поля name этих структур.

Массивы структур

Аналогично массивам простых типов могут быть объявлены массивы структур.

Пример.

```
struct Friend { char name[20]; int age; };
typedef struct Friend MyFriend; // MyFriend - синоним
int main()
{
  struct Friend myFriends[20];
  int i;
  myFriends[3].age = 19;
  for ( i = 0; i < 20; ++i )
    myFriends[i].age = 10 + i;
  return 0;
}
```

В приведенном примере объявлен myFriends – массив структур Friends и проиллюстрирована работа с элементами этого массива.

Члены (поля) структур

Членами (полями) структуры могут быть:

1. Элементы «простых» типов: int, double, char, …
2. Массивы
    a. «простых» элементов;
    b. массивов;
    c. структур.
3. Другие (вложенные) структуры

Такое многообразие возможностей позволяет строить сложно структурированные конструкции данных и появлению выражений типа:

```
phonebook.records.pages[i].item[0].name
```

которое означает:

- phonebook это структура, одним из полей которой является поле records;
- records это структура, одним из полей которой является поле pages;
- pages это массив структур, одним из полей который является поле item;
- item это массив структур, одним из полей который является поле name.

Структура не может быть членом самой себя:

```
struct Friend
{
  char name[20];
  int  age;
  Friend FriendOfMyFriend; // ошибка компиляции!!!!
};
```

Но в качестве члена структуры может выступать указатель на себя:

```
struct Friend
{
  char name[20];
  int  age;
  Friend* FriendOfMyFriend; // ошибки нет
};
```

Если две структуры должны содержать указатели друг на друга, то вторая структура должна быть объявлена перед первой в виде прототипа:

```
struct S2;      // Объявление прототипа
struct S1
{
 struct S2* link;
};
struct S2
{
 struct S1* link;
};
```

8.1.3. Структуры и функции

Структура как параметр функции

Структура может передаваться функции в качестве формального параметра:

- по значению;
- по указателю;
- по ссылке (только в C++!!).

Передача функции структуры в качестве параметра имеет следующие особенности:

1. При передаче по значению под структуру-формальный параметр выделяется память, в которую копируется передаваемая структура.

2. При передаче по указателю (и по ссылке) выделение памяти под формальный параметр и копирование не выполняются, что дает существенную экономию рабочей памяти и времени выполнения программы.
3. Если принимаемая функцией структура является входным параметром, то при передаче по указателю (и по ссылке) возникает опасность, что функция может «забыть» об этом и изменить поля полученной структуры.

Отсюда вытекают следующие правила передачи функции структуры:

4. Структуру надо передавать функции по указателю (или по ссылке).
5. Если передаваемая функции структура является входным параметром, то передавать ее надо по константному указателю (или константной ссылке).

Структура как возвращаемое функцией значение

Функция может возвращать структуру:

- как значение;
- как указатель;
- как ссылку (только в C++!!).

Возвращение функцией структуры имеет следующие особенности:

1. При возврате структуры как значения возвращаемая структура объявляется, как правило, как локальная переменная функции. При ее возврате создается временная копия возвращаемой структуры, которая и возвращается функций. После завершения выполнения функции и использования возвращенного результата временная копия уничтожается.
2. При возврате структуры как указателя возвращаемая структура создается, как правило, в куче. При ее возврате возвращается указатель на эту структуру. Ответственность за ее освобождение лежит на вызывающей функции.
3. При возврате структуры как ссылки возвращаемая структура объявляется, как правило, как локальная переменная функции. При ее возврате возвращается ссылка на эту локальную переменную.

8.2. Объединения

8.2.1. Определение объединения

Объединение — это переменная, которая в различное время может содержать значения различных типов и размеров.

Синтаксис объявления объединения имеет вид:

```
union [<имя_типа>]
{
    <описание_членов>
}[<список_переменных>];
```

где:

- `<имя_типа>` - имя типа объединения;
- `<описание_членов>` - описание членов объединения (вариантов значений и типов, которые может принимать объединение);
- `<список_переменных>` - необязательный при объявлении типа список переменных объединения.

Пример:

```
union PhoneNumber
{
  long number;
  char coded[12];
} phone1, phone2;
phone1.number = 4352217;
StrCopy(phone1.coded, 12, "+79020000000");
```

Расположение полей number и coded в памяти представлено на рис. 8.1. При выполнении первого присваивания приведенного примера объединение phone1 получит значение 4352217. При выполнении второго присваивания – значение +79020000000, которое затрет ранее присвоенное значение.

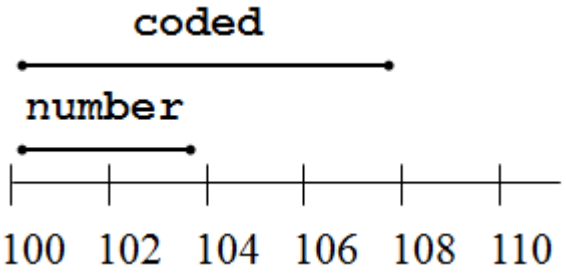


Рис. 8.1. Расположение полей объединения `PhoneNumber`.

8.2.2. Представление прямоугольников

Прямоугольники могут быть представлены несколькими способами:

- координатами левого верхнего и правого нижнего угла;
- координатами центра и размерами (ширина, высота);
- координатами левого верхнего угла и размерами;
- ……..

При этом, возникает необходимость в одном объявлении прямоугольника использовать несколько вариантов его представления.

Пример.

Описать объявление прямоугольника, сочетающего следующие варианты его представления:

- координатами левого верхнего и правого нижнего угла;
- координатами центра и размерами (ширина, высота).

Решение:

```
union Rectangle
{
  struct { int left, top, right, bottom;         };
  struct { int x_centr, y_centr, width, height; };
} r1, r2;
// Координаты левого верхнего и правого нижнего угла
r1.left = 4;
r1.top = 10;
r1.right = 25;
r1.bottom = 30;
// Координатами центра и размеры
r2.x_centr = 20;
r2.y_centr = 30;
r2.width = 10;
r2.height = 30;
```

Расположение в памяти полей объединения `Rectangle` представлено на рис. 8.2. Переменная r1 примера используется в представлении «координаты левого верхнего и правого нижнего угла», а r2 – в представлении «координаты центра и размеры».

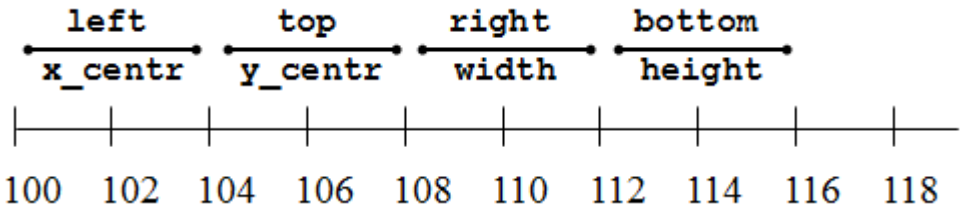


Рис.8.2. Расположение полей объединения `Rectangle`.

Пример.

Описать объявление прямоугольника, сочетающего следующие варианты его представления:

- координатами левого верхнего и правого нижнего угла;
- координатами левого верхнего угла и размеры;

Решение:

```
struct Rectangle1
{
  int left, top;
  union { int right, width; };
  union { int bottom, height; };
}r1, r2;
// Координаты левого верхнего и правого нижнего угла
r1.left = 4;
r1.top = 10;
r1.right = 25;
r1.bottom = 30;
// Координатами центра и размеры
r2.left = 20;
r2.top = 30;
r2.width = 10;
r2.height = 30;
```

Расположение в памяти полей объединения `Rectangle1` представлено на рис. 8.3. Переменная r1 примера используется в представлении «координаты левого верхнего и правого нижнего угла», а r2 – в представлении «координаты левого верхнего угла и размеры».

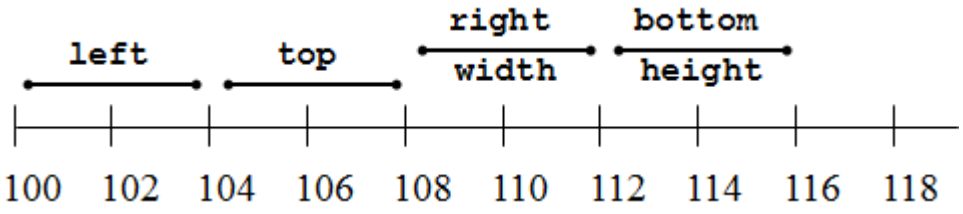


Рис. 8.3. Расположение полей объединения `Rectangle1`.

8.3. Битовые поля

8.3.1. Что такое битовые поля?

Битовые поля — способ указать количество бит, необходимых для хранения члена структуры или объединения.

Объявление битовых полей:

```
{struct|union} [<имя_типа>]
{
   <тип> [<имя_поля>] : <количество_бит>;
   ...
} [<список_переменных>];
```

где:

- <имя_типа> - имя типа структуры или объединения;
- <тип> - тип битового поля;
- <имя_поля> - имя битового поля;
- <количество_бит> - количество бит, которое должно занимать битовое поле;
- <список_переменных> - необязательный при объявлении типа список переменных объединения.

В языке С могут использоваться следующие типы битовых полей:

- int;
- signed [int];
- unsigned [int].

Битовые поля имеют следующие свойства:

1. Количество бит не должно превышать размера типа поля.
2. Битовые поля могут использоваться наряду с обычными полями.
3. Битовые поля могут не иметь имени, что будет означать пропуск указанного количества бит.

8.3.2. Пример

```
struct Date
{
  unsigned year  : 11;
  unsigned month : 4;
  unsigned       : 1;
  unsigned day   : 5;
  unsigned week  : 3;
};
```

Расположение в памяти битовых полей примера представлено на рис. 8.4.

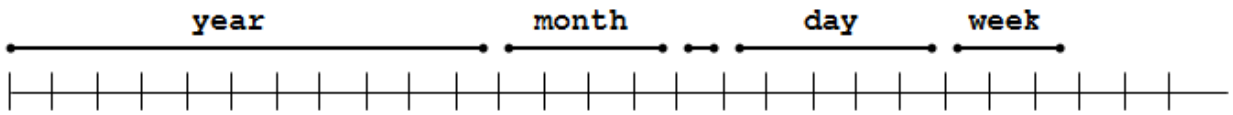


Рис. 8.4. Расположение битовых полей структуры `Date`.

СПИСОК ЛИТЕРАТУРЫ

1. Подбельский В.В. Программирование на языке СИ./М. Финансы и статистика, 2004 г., С. 598
2. Брайан У. Керниган, Деннис М. Ритчи. Язык программирования С./ М. Вильямс, 2017г., С. 288.
3. Герберт Шилдт. Полный справочник по С./ М. Вильямс, 2009г., С. 704.

СОДЕРЖАНИЕ

Сергей Николаевич Карпенко

Основы программирования на языке C

Учебно-методическое пособие

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
603950, Нижний Новгород, пр. Гагарина, 23.

Подписано в печать _____. Формат _____.

Электронная версия