МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Белгородский государственный технологический университет
им. В. Г. Шухова

В. С. Брусенцева

ЯЗЫК ПРОГРАММИРОВАНИЯ СИ

Утверждено ученым советом университета в качестве учебного пособия
для студентов направлений подготовки
09.01.01 – Информатика и вычислительная техника
и 09.01.04 – Программная инженерия

Белгород
2018

УДК 004.43(07)
ББК 32.973я7
Б89

Рецензенты:

Кандидат технических наук, доцент Белгородского государственного технологического университета им. В. Г. Шухова *Н. Н. Подгорный*
Кандидат физико-математических наук, доцент Белгородского государственного национального исследовательского университета (НИУ «БелГУ») *В. В. Флоринский*

Брусенцева, В. С.
Б89 Язык программирования Си: учебное пособие /В. С. . Брусенцева. – Белгород: Изд-во БГТУ, 2018. – 58 с.

В описании языка Си автор ориентируется на знания студентов, изучивших основы алгоритмизации и программирование на языке Паскаль, опирается на понятия языков программирования, которыми уже владеют студенты, акцентирует внимание на отличиях языка Си от языка Паскаль.

Учебное пособие предназначено для студентов направлений подготовки 09.01.01 – Информатика и вычислительная техника и 09.01.04 – Программная инженерия.

Публикуется в авторской редакции

УДК 004.43(07)
ББК 32.973я7

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Язык Си был разработан и реализован начале 70-х гг. XX в. в AT&T Bell Telephone Laboratories Денисом Ритчи во время совместной работы с Кеном Томпсоном над операционной системой UNIX. Язык Си был задуман для решения задач системного программирования, и примерно 80% операционной системы UNIX написано на языке Си. Основной целью создания этого языка было максимальное приближение программиста к используемым аппаратным средствам с сохранением всех преимуществ языка высокого уровня. Это должно было обеспечить с одной стороны мобильность программного обеспечения, а с другой – его эффективность.

Прообразом языка Си был язык B, автором которого является Кен Томпсон.

В настоящее время язык Си и его библиотеки стандартизованы Американским национальным институтом стандартов (ANSI). Предпоследней является стандарт ISO/IEC 9899:1999, также известный как C99, который был принят в 2000 г. Последней на данный момент версией стандарта является стандарт C11 или ISO/IEC 9899:2011

Достоинства языка Си:

1. Позволяет разбивать большую программу на отдельно компилируемые компоненты (модули), а потом объединить их в единую программу.
2. Обеспечивает доступ к аппаратным средствам (регистрам и ячейкам памяти), что требует знания устройства компьютера. То есть Си объединяет в себе достоинства языков низкого и высокого уровня.
3. Очень гибкий язык. При этом большая ответственность возлагается на программиста, так как в Си нет такого контроля типов, как в Паскале.
4. Увеличение гибкости способствует построению компактного и эффективного машинного кода.
5. Мобильный язык.
6. Маленький язык. Мощь языка заключается в его библиотеках.
7. Поддерживает структурное программирование.

Недостатки языка Си:

1. Не для первоначального обучения.
2. За счет гибкости программы могут быть неудобочитаемыми.

Некоторые отличия языка Си от языка Паскаль:

1. В языке Си все подпрограммы называются функции.
2. Функция может не возвращать никакого значения. В этом случае она эквивалентна процедуре.
3. Даже если функция возвращает значение, его можно игнорировать.
4. Функции не могут быть вложенными.
5. Программа представляет собой набор функций, одна из которых должна иметь имя *main*.
6. В заголовке функция, даже если она без параметров, должны быть круглые скобки: Например, *int main*().

7. Точка с запятой (;) в языке Си является признаком наличия оператора, а не разделителем операторов.

8. Операторными скобками являются фигурные скобки.

9. Описания являются операторами.

10. В соответствии с последними стандартами С99 и С11 переменные могут описываться в любом месте программы, а не только в начале блока.

11. В отличие от языка Паскаль в Си прописные и строчные буквы различаются. Ключевые слова, имена переменных и функций записываются строчными буквами, а константы принято записывать прописными.

Комментарии

Многострочный комментарий начинается парой символов **/***, и заканчивается парой символов ***/**.

Для коротких комментариев можно использовать однострочный комментарий, который начинается парой символов **//**. При этом комментарием является последовательность символов до конца строки.

ПРОСТЫЕ ТИПЫ ДАННЫХ

В языке Си 5 стандарта С99 базовых типов:

char, int – целые;

float, double – вещественные;

void – пустой.

В стандарт С11 включены типы char16_t и char32_t для поддержки кодировки Unicode.

Значением типа *char* является однобайтовое целое число – знаковое или беззнаковое, в зависимости от реализации. Каждое такое число является кодом соответствующего символа таблицы ASCII.

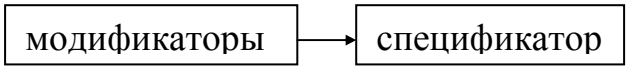
int – целый тип, знаковый, размер которого определяется размером машинного слова

Тип *float* занимает 4 байта, Модуль его значения принадлежит интервалу $(3{,}4*10^{-38}; 3{,}4*10^{38})$. Мантисса содержит 6-7 правильных значащих цифр.

Тип *double* занимает 8 байтов, Модуль его значения принадлежит интервалу $(1{,}7*10^{-308}; 1{,}7*10^{308})$. Мантисса содержит 14-15 правильных значащих цифр.

Типу *void* не выделяется память, он может использоваться при описании функции для указания того, что функция не возвращает никакого значения или для явного указания отсутствия параметров в списке.

С помощью идентификаторов *signed, unsigned , short, long* (модификаторы) базовые типы (спецификаторы) могут быть модифицированы.



Модификаторы *short*, *long* изменяют размер типа.

short и *long* могут использоваться со спецификатором *int*. При этом *int* можно опустить, и тогда *short* и *long* рассматриваются как спецификаторы.

Обычно из трех разновидностей целого типа *short*, *int* и *long* реализованы два. При этом стандарт требует, чтобы выполнялось условие *sizeof*() ≤ *sizeof*(*int*) ≤ *sizeof*(*long*)

Унарная операция *sizeof*(<тип>) определяет размер типа.

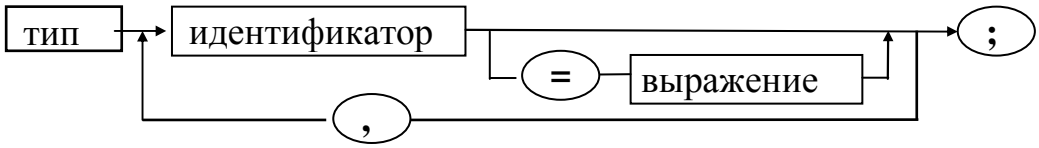
Модификатор *long* можно использовать со спецификатором *double*. *sizeof*(*long double*) = 10 байтам. Модуль значения типа *long double* принадлежит интервалу ($3{,}4*10^{-4932}$; $1{,}7*10^{4932}$). Мантисса содержит 18-19 правильных значащих цифр.

Модификаторы *signed* (знаковый), *unsigned* (беззнаковый) используют с целыми типами.

Описания *signed char* и *unsigned char* гарантируют, что тип *char* не будет зависеть от реализации.

unsigned short, *unsigned int* ≡ *unsigned* и *unsigned long* – беззнаковые целые.

Описание скалярных переменных



Локальные переменные (в языке Си называются автоматическими) при описании их можно инициализировать выражениями, значения которых к этому моменту определены.

Для улучшения читабельности переменные или группы логически связанных переменных рекомендуется описывать в разных строках и комментировать их назначение.

Примеры:

unsigned i, j; // параметры циклов

long n=10, *m*=2**n*; // длины последовательностей

signed char ch; // наиболее часто встречающийся символ.

Перед описанием переменной с инициализацией можно поместить модификатор ***const.*** Например, ***const float pi*=3.1415;.**

Значение такой переменной присваивается на этапе компиляции, и оно не должно изменяться во время выполнения программы, хотя полноценной константой эта переменная не является, ее нельзя использовать в константных выражениях

КОНСТАНТЫ-ЛИТЕРАЛЫ

Числовые константы всегда неотрицательны. Знак минус рассматривается как знак унарной операции.

Целые константы

По умолчанию, если целая константа умещается в машинном слове, то она относится к типу *int*, иначе – к *long*. Типы целых констант можно указать явно с помощью суффиксов. Суффикс *l* (*L*) относит целую константу к типу *long*. Например, 34*L*. Суффикс *u* (*U*) относит константу к беззнаковому типу. Например, 176*U,* 90*ul*.

Целые константы могут быть представлены в десятичной, восьмеричной и шестнадцатеричной системе счисления.

Целая десятичная константа, отличная от 0, начинается цифрой отличной от 0. Восьмеричная константа имеет префикс 0. Пара символов 0*x* или 0*X* является префиксом шестнадцатеричной константы.

Примеры.

$020_8 = 16_{10}$

$0x20_{16} = 32_{10}$

$0XFF_{16} = 255_{10}$

Вещественные константы

Вещественные константы по умолчанию имеют тип *double*. Суффикс *f* позволяет отнести константу к типу ***float.***

Вещественная константа может быть записана в форме с фиксированной или с плавающей точкой.

1*E*2*f* = 100.

1.3*e*–5 = 0.00013

Символьные константы

Символьные константы, имеющие графические изображения, (за исключением некоторых) заключаются в апострофы: ‘*a*’, ‘1’.

Для записи некоторых управляющих символов используются, так называемые, esc*ape*-последовательности:

‘*n*’ (10) – символ перехода на новую строку,

‘*t*’ (9) – символ табуляции,

‘*b*’ (8) – символ удаления предшествующего символа,

‘*a*’ (7) – символ подачи звука,

‘\0’ (0) – символ признака конца строки.

Escape-последовательности используются для записи следующих символов, имеющих графические изображения: ‘\’’ (апостроф), ‘\”’ (кавычки), ‘\\’ (обратный слеш), ‘\?’ (знак вопроса).

Существует также альтернативная форма записи любого символа с помощью *escape*-последовательности вида: ‘\<код>’. Код может быть задан восьмеричным значением без лидирующего нуля или шестнадцатеричным с префиксом *x или X*. Например, ‘\101’≡‘*x*41’ ≡ ‘*a*’.

Строковые константы

Строковые константы – последовательности символов, заключенные в кавычки. Например: “Это строка”.

Строка занимает на один байт больше, чем количество символов в ней, так как в конец строки компилятор автоматически добавляет ноль-символ, символ с кодом 0 (‘\0’), который является признаком конца строки.

В строках можно использовать *escape*-последовательности, но при этом они в апострофы не заключаются. Если длинная строковая константа не помещается в одной строке экрана, то её можно перенести на новую строку, используя в качестве символа переноса обратный слеш (\).

СТАНДАРТНЫЙ ВВОД И ВЫВОД

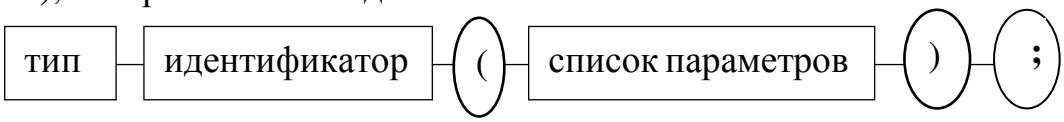
В само́м языке Си нет средств ввода и вывода. Ввод и вывод выполняется с помощью библиотечных функций, интерфейс с которыми находится в стандартном заголовочном файле *stdio.h* (расширение *h* от слова *head*). В этом файле содержится всё, что необходимо для ввода и вывода. Текст заголовочного файла должен быть включен в программу с помощью директивы препроцессора

#include <stdio.h>

В системах программирования эта директива обычно уже подключена в окне редактирования.

В файле *stdio.h* описаны не только функции, но другие программные объекты, которые могут быть использованы, например, константа *EOF*, значение которой равно –1.

Функции в заголовочных файлах описаны своими заголовками (прототипами), которые имеют вид



Здесь тип – тип возвращаемого значения, идентификатор – имя функции, в списке параметров перечисляются через запятую или только типы параметров, или типы и имена параметров.

Ввод и вывод символов

int getchar(*void*) – функция без параметров считывает вводимый с клавиатуры символ (из стандартного потока ввода) и возвращает его код как целое. В случае ошибки функция возвращает значение *EOF*.

int putchar(*int c*) – выводит на экран (в стандартный поток вывода) символ *c*, возвращает код выводимого символа как целое. В случае ошибки функция возвращает значение *EOF*.

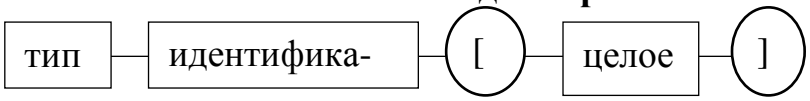
Описание программы, которая считывает символ и выводит его.

```
#include <stdio.h>
int main(void)
{
  int c=getchar();
  putchar(c);
}
```

Тело этой функции можно заменить одним оператором:

```
putchar(getchar());
```

Описание одномерных массивов



Тип – базовый тип, идентификатор – имя массива, целое определяет размер массива. Индексация элементов массива начинается с нуля.

int a[*n*];

a – массив из *n* целых чисел: *a*[0], *a*[1], … , *a*[*n*–1].

В Си имя массива является указателем на базовый тип (переменная *a* имеет тип – указатель на целое). Имя массива – константа-указатель, его значение равно адресу начального элемента массива, и изменить его нельзя.

Тип «типизированный указатель» имеет вид: <базовый тип> *.

*int *p=a*; // *p* – переменная-указатель на целое, содержит адрес элемента *a*[0]. Типы переменной *p* и константы *a* одинаковы.

Строки хранятся в памяти ЭВМ в символьных массивах. В соответствии с описанием *char s*[200]; в массиве *s* могут храниться строка, длина которой не превышает 199, так как последним символом должен быть признак конца строки ‘\0’.

Ввод и вывод строк

Прототип функции ввода строки:

*char * gets* (*char s*[]) или *char * gets*(*char * s*)

Функция *gets* считывает символы, вводимые с клавиатуры до нажатия клавиши *Enter* (до символа ‘*n*’), записывает их строку *s*. Вместо символа новой строки ‘*n*’ в строку *s* записывает ноль-символ. Возвращает указатель на начальный символ строки *s* или пустой указатель *NULL* (аналог *nil* в Паскале) в случае ошибки.

Функция *gets* удалена из стандарта C11 и заменена новой безопасной *char * gets_s*(*char *s*, *size_t size*), где *size* – размер массива *s*. В отличие от функции *gets* в строке сохраняется символ ‘*n*’, и после него в *s* записывается ноль-символ. Функция при успешном выполнении возвращает указатель на начальный символ строки *s*. В случае ошибки или конца файла функция возвращает пустой указатель *NULL*.

Прототип функции вывода строки:

int puts(*char s*[]); или *int puts*(*char *s*);

Функция *puts* выводит на экран строку *s*, заменяя ноль-символ символом ‘*n*’. Возвращает неотрицательное число или *EOF* в случае ошибки.

Форматный вывод

int printf(*const char* *<управляющая строка>[,<список параметров>]);

Список параметров – список выражений

Управляющая строка содержит символы трёх типов: обычные символы, которые выводятся на экран; управляющие символы; и спецификации преобразования.

printf (“Введите число”); //аналог *write*(‘Введите число’);

printf (“Введите число *n*”); //аналог *writeln*(‘Введите число’);

Каждая спецификация преобразования определяет, в каком виде будет выведено значение очередного параметра из списка параметров.

Спецификация преобразования имеет вид:

%[<модификаторы>]<символ преобразования>

Каждому символу преобразования, за исключением символа %, соответствует параметр определенного типа

Символы преобразования представлены в таблице.

Таблица

Символ	Назначение
d	вывод целого знакового десятичного числа
i	как *d*, но аргумент может быть задан 8-ричной, 16-ричной константой или целым выражением.
u	вывод целого неотрицательного числа. Содержимое памяти рассматривается как беззнаковое целое.
o	вывод целого неотрицательного числа в восьмеричном виде без лидирующего нуля
x	вывод целого неотрицательного числа в 16-ричном виде без префикса и со строчными буквами
X	как x, но буквы прописные
f	вывод вещественного числа с фиксированной точкой
e, E	вывод вещественного числа с плавающей точкой
g, G	вывод в наиболее компактном представлении формата *f* или *e*, *E*
c	вывод символа
s	вывод строки
p	вывод указателя
%	вывод символа ‘%’
n	ничего не выводит, но этому символу должен соответствовать параметр типа *int**, и по этому адресу будет сохранено количество символов, которое было выведено до встречи спецификации %*n*

Примеры.

printf(“*i* = %*i, j* = %*i*”, 025, 25); // *i* = 21, *j* = 25

printf(“*i* = %*i*”, 0*x*25); // *i* = 37

printf(“*i* = %*o*”, 28); // *i* = 34

printf(“*i* = %*x*”, 29); // *i*= 1*d*

printf(“*i* = %*X*”, 29); // *i*= 1*D*

printf(“*p*=%*i*%%”,75); //*p*=75%

Модификаторы **::**=[<флаги>][<ширина>][.<точность>][*F*|*N*|*h*|*l*|*L*]

F и *N* используются с символом преобразования *p*, *F* – для вывода дальних указателей в формате *XXXX*:*УУУУ* (*XXXX* и *УУУУ* 16-ричные сегмент и смещение). *N* – для вывода ближних указателей, выводится только смещение (*УУУУ*).

h используется с символами преобразования для вывода целых, если соответствующий параметр имеет тип *short*.

l – как *h*, но с параметром типа *long*.

L и *l* используются с символами преобразования для вывода вещественных значений, *l* должен соответствовать параметр типа *double*, *L – long double*.

Ширина – ширина поля вывода, это минимальное количество знакомест, выделяемых выводимому значению. Если для выводимого значения требуется больше знакомест, то поле вывода расширяется до минимально необходимого, Иначе – значение прижимается к правому краю.

Точность определяет:

a) максимальное количество символов при выводе строки;

б) максимальное количество цифр после точки для символов преобразования *f*, *e*, *E*

в) количество значащих цифр для *g*, *G*, не искажая выводимого значения;

г) минимальное количество цифр при выводе целого числа. Если количество цифр меньше точности, то число дополняется ведущими нулями.

Вывод вещественного числа с указанной точностью осуществляется с округлением.

Ширина и/или точность может быть заменена *, в этом случае в списке аргументов каждому символу * должен соответствовать параметр, значением которого заменяется ширина и/или точность.

По умолчанию с символами преобразования *f*, *e*, *E* после точки выводится 6 знаков. Если точность равна 0, то точка не выводится.

При использовании символа преобразования *g* или *G*, если порядок меньше –4 или порядок больше точности, используется вывод с плавающей точкой, иначе – с фиксированной. Хвостовые нули не выводятся. Если вещественное число является целым по величине, то точка не выводится.

Флаги

1. *Знак минус*. По умолчанию выводимое значение, если ширина поля вывода больше, чем количество символов в выводимом значении, значение выравнивается по правому краю. Для выравнивания значения по левому краю используется флаг «–»

2. *Знак плюс* используется для вывода числа со знаком. *printf*(“%+*i*”, 29); // Выводится +29

3. *Знак пробел* предписывает выводить пробел перед выводом неотрицательного значения.

printf(“(“%*i*%%*i*% *i* ”12, 34, 34); // Выводится 1234 34

4. 0 используется при выводе числовых значений. Если ширина поля вывода больше длины выводимого значения, пробелы слева заменяются нулями.

printf(“%05*i*”,17); / Выводится / 00017

5. *Знак решетка* (#) предписывает использовать альтернативную форму при выводе числовых значений.

а) Число в 8-ричном виде выводится с лидирующим нулем. *printf*(“%#*o*=%*o*”,21,21); // Выводится 025=25

б) Число в 16-ричном виде выводится с префиксом 0*x* или 0*X*. *printf*(“%#*x* = %*x* = %#*X* = %*X* ”,42, 42, 42, 42);
/* Выводится 0*x*2*a* = 2*a* = 0*X*2*A* = 2*A*/.

в) Вещественные числа всегда выводятся с десятичной точкой.

г) При выводе с символом преобразования *g* или *G* не отбрасываются незначащие нули.

В спецификации преобразования можно использовать несколько флагов и перечислять их в любом порядке.

Функция возвращает количество выведенных символов или отрицательное число в случае ошибки.

Форматный ввод

int scanf(<управляющая строка>, <список параметров>);

Функция *scanf* считывает символы из стандартного входного потока (клавиатуры), интерпретирует их в соответствии с форматом, указанным в управляющей строке, и присваивает полученные значения переменным, на которые указывают параметры функции.

Параметрами функции *scanf* являются не переменные, а их адреса.

Операция взятия адреса: &<переменная>.

Управляющая строка содержит спецификации преобразования для разбиения входного потока на поля ввода. Кроме этого, она может содержать пустые символы, которые при вводе игнорируются, и обычные символы, которые предполагаются совпадающими с соответствующими вводимыми символами, такие символы считываются, но никуда не записываются.

Например, при вводе даты в формате 25/01/2018 должна быть вызвана функция *scanf*(“%*i* // %*i* // %*i*”,&*day*, &*month*, &*year*);

Спецификация преобразования имеет вид.

%[*][<ширина>][*F*|*N*|*h*|*l*|*L*] <символ преобразования>

Символ	Назначение	Тип параметра
d	ввод целого десятичного числа	*int**
i	как *d*, но вводимое число может быть 8-ричным, 16-ричным или 10-ным.	*int**
u	ввод целого неотрицательного десятичного числа.	*unsigned**
o	ввод целого неотрицательного числа в 8-ричном виде.	*unsigned**
x	ввод целого неотрицательного числа в 16-ричном виде	*unsigned**
f, e, g	ввод вещественного числа	*float**
c	ввод символа	*char**
s	ввод строки	*char**
p	ввод указателя	*void**
%	считывание из потока ввода %	нет параметров.

Модификаторы *F*, *N, h*, *l*, *L* позволяют уточнить тип данных:

– *F*, *N* для ввода дальних и ближних указателей

– *h*: для ввода значений типа *short int* //%*hd*

– *l*: для ввода значений типов *long* (%*ld*) или *double* //%*lf*, %*le*, %*lg*

– *L*: для ввода значений типа *long double* //%*Lf*, %*Le*, %*Lg*

Ширина определяет максимальное количество символов, принадлежащих вводимому значению.

При считывании очередного значения пропускаются все пустые символы. Если первый непустой символ не может принадлежать вводимому значению, выполнение функции прекращается, иначе считывается последовательность символов, которая преобразуется к типу соответствующего значения и записывается по указанному адресу. Считывание символов происходит до считывания

пустого символа, или до символа, который не может принадлежать вводимому значению, или считывается количество символов, равное ширине, если до этого не встретился пустой символ, или символ, который не может принадлежать вводимому значению.

При вызове функции *scanf*(“%2*i* %3*i* %3*i*”,&*i*1, &*i*2, &*i*3);, если вводится последовательность 172345*a* , переменным будут присвоены следующие значения: *i*1 = 17, *i*2 = 234, *i*3 = 5.

Символ «*» означает подавление ввода, то есть значение, соответствующее такой спецификации преобразования, считывается по всем описанным выше правилам, но оно никуда не записывается.

Согласно описанию *char s*[300]; при вызове функции *scanf*(“%*s*”, *s*); параметр *s* записывается без амперсанта (&), так как имя массива определяет адрес начального символа массива. Так, при вводе строки Белгород, ул. Костюкова по адресу *s* будет записано Белгород, .

Выполнение функции *scanf* заканчивается, когда исчерпана управляющая строка, или элемент ввода не может принадлежать вводимому значению. В этом случае функция возвращает число, равное количеству считанных и записанных по указанным адресам значений. Если первым считанным символом является символ «конец файла», то функция возвращает *EOF*.

ОПЕРАЦИИ В ЯЗЫКЕ СИ

Особенности операций

1. Большое количество и разнообразие.
2. Компактный синтаксис.
3. Наличие операций, которые в других языках программирования являются операторами.
4. При выполнении некоторых операций не только формируется значение, но и изменяются значения операндов.
5. Приоритеты некоторых операций отличаются от приоритетов в Паскале.
6. Менее жесткие требования к совместимости типов.

Неявное приведение типов в операциях

Если в выражении или в списке фактических параметров функции встречаются операнды типа *char* или *short*, то они автоматически приводятся к типу *int*. Эта операция называется повышением целочисленности.

Если при выполнении операции операнды имеют разные типы, то происходит автоматическое приведение типов к общему типу по следующим правилам

Если один из операндов имеет тип *long double*, то второй операнд приводится к типу *long double*,

иначе если один из операндов имеет тип *double*, то второй операнд приводится к типу *double*,

иначе если один из операндов имеет тип *float*, то второй операнд приводится к типу *float*,

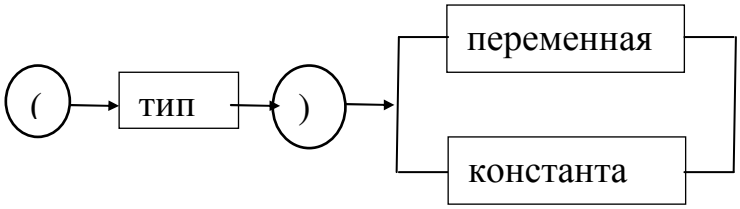
иначе если один из операндов имеет тип *unsi*gne*d long int*, то второй

операнд тоже будет преобразован к типу *unsigned long int*,
иначе если один из операндов имеет тип *long*, то второй операнд приводится к типу *long,* если его значения покрывают все значения типа *unsigned*, иначе – к типу *unsigned*
иначе если один из операндов имеет тип *unsigned*, то второй операнд приводится к типу *unsigned*
иначе выполняется операция повышения целочисленности.

Операция повышения целочисленности не изменяет значения в памяти. При приведении знакового типа к беззнаковому число в дополнительном коде рассматривается как беззнаковое.

Целое преобразуется к вещественному типу неточно, оно заменяется одним из ближайших к нему вещественных.

Операция приведения типа



В некоторых случаях неявные преобразования приводят к неправильным результатам. Например,

unsigned u = 25;

Значением выражения *u* > –1 в языке Си будет ложь, так как –1 приводится к типу *unsigned*. Правильный результат даст выражение (*int*)*u* >–1.

Таким образом, для получения правильного результата может потребоваться явное приведение типа.

Без потери данных проходят следующие преобразования:

signed char –>short –> int –> long,

unsigned char –> unsigned short –> unsigned int –> unsigned long,

float –> double –> long double.

Целое меньшего размера преобразуется к целому типу большего размера, не изменяя своего значения с сохранением знака.

Если целое большего размера приводится к целому меньшего размера, то старшие байты отбрасываются.

Вещественное значение преобразуется к целому отбрасыванием знаков после точки, если такое значение может быть присвоено целому. При описании *int i* = 3.8; получим *i* = 3.

Если вещественное значение не может быть присвоено целому, то результат не определен. Например, это произойдет при приведении вещественного отрицательного к целому беззнаковому.

При приведении целого типа к вещественному целое заменяется одним из ближайших к этому целому вещественным.

Приведение вещественного числа меньшей точности к вещественному большей точности, происходит без искажений.

Вещественное большей точности преобразуется к вещественному меньшей точности приближенно, если оно может быть представлено в типе с меньшей точностью. Если значение большей точности не может быть представлено в типе с меньшей точностью, то результат не определен.

Арифметические операции

Приоритеты арифметических операций в порядке убывания (в скобках указан знак операции):

1) унарные плюс(+) и (–).
2) умножение (*), деление (/), нахождение остатка от деления (%).
3) сложение (+), вычитание (–).

При нахождении остатка от деления типы операндов должны быть целыми. Если хотя бы один из операндов меньше нуля, то результат зависит от реализации, поэтому для мобильности рекомендуется использовать эту операцию с положительными операндами.

Для остальных операций типы операндов могут быть любыми числовыми.

Тип результата бинарных операций сложения, вычитания и умножения, совпадает с типом, к которому неявно приводятся операнды. Операция деления с целыми операндами выполняется как целочисленное деление. Если требуется при этом выполнить обычное деление, один из операндов следует привести к вещественному типу.

Примеры. Частное 17 / 5 равно 3, частное 17. / 5 равно = 3.4,

Побитовые операции

Побитовые операции в языке Си выполняются так же, как и в Паскале, типы операндов – целые. Но приоритеты операций в языках Си и Паскаль отличаются.

Приоритеты побитовых операций в порядке убывания (в скобках указан знак операции):

1) отрицание (~);
2) сдвиг вправо (>>), сдвиг влево (<<);
3) побитовая конъюнкция (&);
4) побитовая дизъюнкция (|);
5) побитовое исключающее или (^).

Бинарные арифметические операции имеют более высокий приоритет, чем бинарные побитовые.

Полем переменной назовем последовательность ее смежных битов. Значение поля длины k переменной a справа от n-го бита, включая n-й бит, определяется выражением $\boxed{\sim(\sim 0 << n)\ \&\ a >> n - k + 1}$.

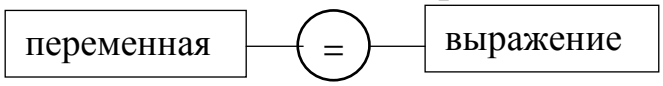
Операции присваивания

Напомним, что результатом выполнения операции является значение, которое в свою очередь может быть операндом другой операции. Оператор является законченной командой исполнителю. Во многих языках программирования

присваивание является оператором. В языке Си присваивание может быть как операцией, так и оператором.

В языке Си есть три вида операций присваивания: обычное присваивание, корректирующее присваивание и операции инкремента и декремента.

Обычное присваивание



Выполнение операции присваивания заключается в том, что значение выражения приводится к типу переменной и присваивается переменной. Результатом этой операции является значение, присвоенное переменной.

Пример.

int m;

float q = 5.7;

printf(*"%i", m = q*);

При выполнении оператора вычисляется и выводится 5 – значение выражения *m = q,* кроме того, переменной *m* присваивается значение 5.

q = m; // Это оператор присваивания.

Операция присваивания занимает предпоследнее место в таблице приоритетов.

Но не только приоритеты определяют очередность выполнения операций. Большинство операций одинакового приоритета в языках программирования при отсутствии скобок выполняются слева направо. Такие операции называются левоассоциативными. В языке Си есть операции, которые выполняются справа налево. Такие операции называются правовссоциативными. Операция присваивания является правоассоциативной.

Выражение *a = b = c = d* равносильно *a* = (*b* = (*c = d*)).

В результате выполнения фрагмента программы

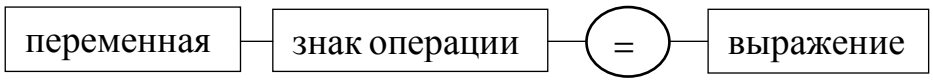
int m;

float q = 5.7;

m = q = q + 1.7;

переменным *m* и *q* будут присвоены значения 7 и 7.4 соответственно.

Корректирующее присваивание



Знаком операции может быть знак бинарной, арифметической или побитовой операции.

Выражение $a[i+5]*= 10$ равносильно выражению $a[i+5] = a[i+5]*10$.

Корректирующее присваивание выполняется эффективнее, чем обычное присваивание.

Корректирующее присваивание имеет такой же приоритет, как и обычное присваивание.

Операции инкремента и декремента

Операции инкремента и декремента – унарные операции. Инкремент (знак ++) увеличивает, а декремент (знак ––) уменьшает значение операнда на 1. Знаки операций инкремента и декремента могут записываться как перед операндом (префиксная форма записи), так и после него (постфиксная форма записи).

Префиксные:

++<выражение>,

––<выражение>.

Постфиксные:

<выражение> ++,

<выражение>––.

Операнд должен иметь числовой тип или быть указателем. Операндом может быть только выражение *L-value*, то есть выражение, которому соответствует объект-переменная в памяти, и значение которого может быть изменено.

Выполнение следующих операторов приводит к одному и тому же результату, но операции инкремента и декремента более эффективны.

1) *a*++;

2) ++*a*;

3) *a*+=1;

4) *a* = *a* + 1;

Если операции инкремента и декремента используются в выражении, где есть другие операции, то выполнение префиксных и постфиксных операций различно. Префиксные инкремент и декремент сначала изменяют значение своего операнда, а затем измененное значение используется при выполнении других операций. Постфиксные операции выполняются после того, как операнд был использован в выражении.

Примеры.

int i = 3, *j* = 3, *n, m*;

n = – –*i*+1;// *n* = 3, *i=2.*

m = *j*– –+1;// *m* = 4, *j=2.*

Выражение *i*+++j рассматривается как(*i*++)+*j.*

Если в выражении есть коммутативные и/или ассоциативные операции, то при их вычислении последовательность может быть разной.

a + *b* + c ≡ (*a* + *b*) + c ≡ *a* + (*b* + c) ≡ (*b* + c) + *a*

Поэтому во избежание неоднозначности результата не рекомендуется использовать одну и ту же переменную с операциями инкремента и декремента в одном выражении или в списке параметров при вызове функции.

Так, если *n* = 1, значение выражения ++*n* + *n* может быть равно 3 или 4.

При вызове функций последовательность вычислений значений фактических параметров может быть разной, поэтому не рекомендуется использовать параметры, в которых есть переменная и эта же переменная с инкрементом или декрементом. Например, при вызове функции *f*(*a*, *a*++) с *a* = 1 возможен вызов *f*(1, 1) или *f*(2, 1).

Операции сравнения

Операции сравнения в языке Си занимают две строки таблице приоритетов.

Операции сравнения в порядке убывания приоритета:

1) >, <, >=, <= ;

2) == (равно), != (не равно).

Выражение $a>b == c<d$ эквивалентно выражению $(a>b) == (c<d)$.

В результате выполнения операции сравнения формируется значение 0 (ложь) или 1 (истина).

Оператор $s+ = a<b;$ можно использовать для определения количества упорядоченных по возрастанию пар во вводимой последовательности вместо условного оператора.

Операции сравнения имеют более низкий приоритет, чем арифметические операции, но более высокий приоритет, чем бинарные побитовые операции.

Логические операции

Специального логического типа в языке Си нет. Выражение, значение которого не равно 0, рассматривается как истина, 0 – ложь. В результате выполнения логических операций формируется результат 0 или 1.

Знаки логических операций (операции перечислены в порядке убывания приоритета):

! – отрицание,

&& – конъюнкция,

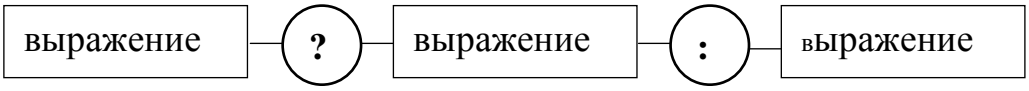
|| – дизъюнкция.

Бинарные логические операции имеют более низкий приоритет, чем операции сравнения.

Пример. Выражение, значение которого равно 1, если год *y* високосный, иначе – равно 0 имеет вид:

$$!(y \,\%\, 100)\ \&\&\ y \,\%\, 4\ ||\ y \,\%\, 400$$

Условная операция



Как видно из синтаксической диаграммы, условная операция является тернарной операцией, то есть она требует трех операндов. Операнды являются выражениями. Первое выражение рассматривается как логическое. Значения второго и третьего выражений приводятся к общему типу. Если значение первого выражения не равно 0 (истинно), то значением условного выражения является значение второго выражения, иначе – третьего.

Пример.

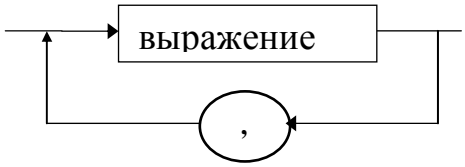
```
int a = 7;
float max, f = 0.5;
max  = a > f ? a : f; //определение максимума
```

Значение условного выражения можно не использовать. В этом случае условная операция аналогична условному оператору.

Условная операция имеет низкий приоритет. Она занимает в таблице приоритетов строку над операцией присваивания.

Операция запятая

Операция запятая позволяет рассматривать последовательность выражений как одно выражение. Она имеет вид:



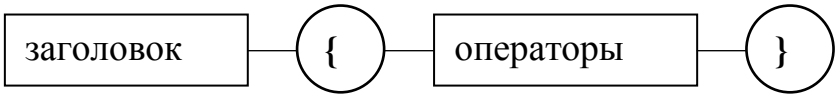
Чаще всего эта операция используется в двух контекстах: как разделитель в списках объектов, например, в операторах описания данных, и как операция, определяющая последовательное вычисление значений выражений, Например в заголовке цикла *for* использование операции запятая аналогично использованию составного оператора. Если в конструкции требуется наличие одного выражения, а согласно алгоритму в этом месте должна быть последовательность выражений, то такую возможность дает операция запятая.

Операция запятая является левоассоциативной. Ее выполнение заключается в том, что последовательно слева направо вычисляются значения выражений. Значение и тип последнего вычисленного значения определяют результат операции запятая.

Операция запятая последняя в таблице приоритетов операций.

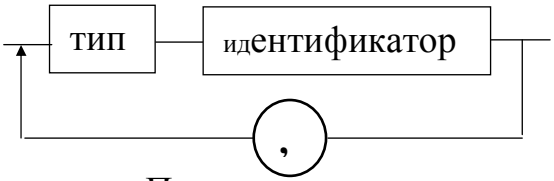
ФУНКЦИИ

Структура функции



Заголовок функции

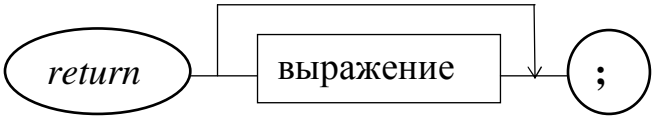
Список параметров



В отличие от языка Паскаль в списке параметров в заголовке функции нельзя описывать секции параметров, должен быть указан тип *каждого* параметра.

Оператор возврата *return* завершает выполнение функции и возвращает управление в вызывающую функцию, в точку, непосредственно следующую за вызовом этой функции. Функция *main* передает управление операционной системе.

Синтаксическая диаграмма оператора *return*:



Если функция возвращает значение, то она должна содержать, по крайней мере, один оператор *return* <выражение>;. Если тип возвращаемого значения *void*, то есть функция не возвращает значение, то она может не содержать оператора *return* или содержать его, но без выражения. Если функция типа *void* не содержит оператор *return*, то ее выполнение заканчивается после выполнения всех операторов.

Пример. Функция *max*_3 возвращает максимальное из трех значений: *a*, *b*, с.

```
int max_3 (int a, int b, int c).
  {
    a = a>b ? a : b;
    return (a>c ? a : c);
  }
```

В языке Си параметры передаются только по значению, то есть после выполнения функции значения всех параметров не изменяются. Если значение простой переменной должно быть изменено после выполнения функции, то функции необходимо передать указатель на эту переменную. В теле функции обращение к переменной выполняется с помощью операции разыменования: *<переменная>.

Пример. Функция *swap* обменивает значениями две переменные по адресам *pa* и *pb*.

```
void swap(int * pa, int * pb)
{
  int t = *pa;
  *pa = *pb;
  *pb = t;
}
```

При вызове функции фактическими параметрами должны быть адреса переменных. Например, при вызове функции *swap*(&*x*, &*y*) будет выполнен обмен значениями целочисленных переменных *x* и *y*.

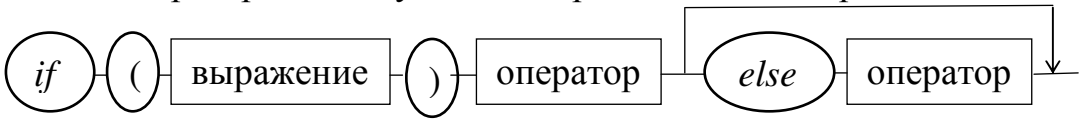
ОПЕРАТОРЫ УПРАВЛЕНИЯ

Составной оператор

Составной оператор (блок) позволяет рассматривать последовательность операторов как один оператор. Он имеет вид:

Условный оператор

Условный оператор используется для организации бинарного ветвления.



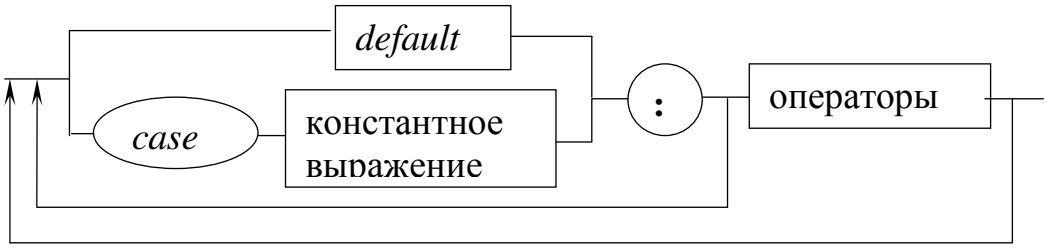
Выражение должно быть арифметическим или указателем.

Как и в языке Паскаль, в случае вложенных условных операторов каждое *else* относится к ближайшему предшествующему *if*.

Оператор-переключатель

Оператор-переключатель используется для организации ~~–~~множествен~~ного~~ ветвления.

Тело оператора-переключателя имеет вид:



Выражение-селектор должно быть целым, как и метки случаев (константные выражения). Метку *default* (аналог *else* в Паскале) можно использовать в любом месте, но только один раз или ни разу. Значения константных выражений должны быть разными.

Выполняются операторы-переключатели в языках Си и Пас~~к~~аль по-разн~~ому~~. Если значение селектора совпадает с одной из меток случаев, то в языке Си выполняются не только операторы, соответствующие этой метке, но и все следующие за ними операторы.

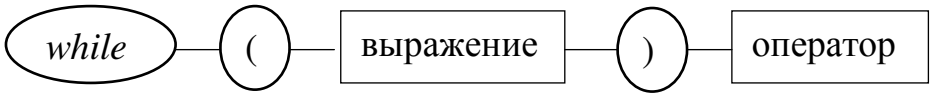
Для того чтобы выполнялись операторы, соответствующие только одной метке, можно использовать оператор разрыва *break* Он прекращает работу производного оператора и передаёт управление оператору, следующему за переключателем.

Пример.

Функция *mark* возвращает оценку в пятибалльной системе, соответствующую числу баллов в 100-балльной.

```
int mark(unsigned n)
  {
   int m;
   switch(n / 20)
     {
       case 0:
       case 1: m = 2; break;
       case 2: m = 3; break;
       case 3: m = 4; break;
       case 4:
       case 5: m = 5; break;
     }
   return m;
  }
```

Цикл с предусловием



Выражение является условием возобновления цикла.

Примеры.

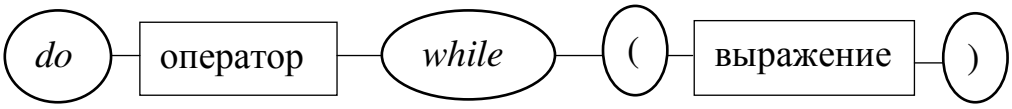
Функция *strlen* возвращает длину строки *s*.

```
int strlen(char s[]) // int strlen(char *s)
 {
  int i = 0;
  while(s[i] != '\0') // можно while (s[i])
     i++;
  return i;
 }
```

Функция *reverse* обращает строку *s*.

```
void reverse (char *s)
 {
  char c;
  int i = 0, j = strlen(s) – 1;
  while(i < j)
    {
      c = s[i];
      s[i++] = s[j];
      s[j++] = c;
    }
 }
```

Цикл с постусловием



Выражение является условием возобновления цикла. Телом цикла является один оператор.

Пример.

Функция *itoa*, записывает число *n* в строку *s* (по адресу *s*) и возвращает указатель на начальный символ строки.

```
char * itoa (int n, char *s)
 {
  int sign, i = 0;
  if ((sign = n) < 0)
       n = –n;
  do
    s[i++] = n % 10 + '0';
  while ((n/=10) > 0);
  if (sign < 0)
```

```
    s[i++] = '–';
  s[i] = '\0';
  reverse (s);
  return(s);
}
```

Цикл *for*

В языке Си нет цикла с фиксированным числом повторений. Цикл *for* является разновидностью цикла с предусловием. Но с его помощью можно описать цикл, как цикл *for* в Паскале.

На языке Бэкуса-Наура цикл *for* имеет вид:

for ([<выражение>]; [<выражение>]; [<выражение>]) <оператор> .

Первое выражение в заголовке – инициализирующее. Второе выражение рассматривается как логическое выражение, это условие возобновления цикла. Третье выражение – корректирующее. Если в заголовке цикла нужно инициализировать или корректировать несколько переменных, то используется операция запятая.

Любого из выражений в заголовке цикла может не быть. При отсутствии логического выражения условие возобновления цикла считается истинным, и выйти из такого цикла можно с помощью оператора *break*, *return* или *goto*.

Примеры.

Функция *reverse* с циклом *for.*

```
char* reverse(char* s)
 {
  int i, j;
  char c;
  for (i = 0, j = strlen(s); i < j; i++, j– –)
    {
      c = s[i];
      s[i] = s[j];
      s[j] = c;
    }
  return (s);
}
```

Функция *strcmp* сравнивает строки *s*1 и *s*2 и возвращает значение, меньшее 0, если *s*1 < *s*2, иначе, если строки равны, возвращает 0, иначе возвращает значение, большее 0.

```
int strcmp (char* s1, char* s2)
{
  int i;
  for (i = 0; s1[i] = =s2[i]; i++)
    if (s1[i] = ='\0')
      return (0);
  return (s1[i] – s2[i]);
}
```

Оператор *break* можно использовать для досрочного выхода из любого цикла. Если *break* находится во вложенном цикле, то выход происходит только из этого вложенного цикла.

Пример.

Функция *del_white* удаляет из строки *s* конечные пустые символы и возвращает длину полученной строки.

```
void del_white (char *s)
 {
  int i = strlen(s);
  while(– –i>0)
    if (s1[i] >'\40')  //если символ больше пробела
      break
  s[++i] = '\0';
 }
```

Проверку символа можно было бы внести в заголовок цикла, но описанный вариант нагляднее.

Оператор продолжения (continue)

Оператор *continue* может находиться в теле любого из циклов. Его выполнение заключается в том, что следующие за ним операторы тела цикла не выполняются, и происходит переход к очередной итерации. Во многих случаях можно обойтись без оператора *continue,* заменив его условным оператором с противоположным условием. Рассмотрим, например, два варианта одного и того же алгоритма.

```
while(<выражение>)          while(<выражение>)
  {                           {
    <операторы>                 <операторы>
    if (<выражение>)            if (!<выражение>)
      continue;                     <оператор>
    <операторы>                 }
  }
```

Оператор безусловного перехода (goto)

Синтаксис оператор безусловного перехода:

goto < метка>; .

Метка – идентификатор. Оператор *goto* передает управление оператору, помеченному меткой: < метка> : <оператор>. Такой оператор обязательно должен быть в программе.

Оператор *goto* позволяет передать управление из вложенного оператора любому внешнему оператору. Не рекомендуется передавать управление извне внутрь производного оператора.

Но так как язык Си является языком структурного программирования, не рекомендуется использовать оператор ***goto***, чтобы не нарушать принципы структурного программирования.

КЛАССЫ ПАМЯТИ

Классы памяти переменных

Класс памяти переменной определяет область действия переменной и время ее жизни. К какому классу памяти относится переменная, зависит от того, где и как она описана.

В языке Си четыре класса памяти: автоматический (*auto*), регистровый (*register*),внешний (*extern*) и статический(*static*).

Класс auto

К классу *auto* относятся по умолчанию параметры функций и локальные переменные (локальные переменные в языке Си называются автоматическими). Название класса может предшествовать описанию переменных. Например: *auto int n*;.

Эти переменные могут инициализироваться при описании выражениями, которые к данному моменту уже имеют определенные значения. По умолчанию автоматические переменные не инициализируется.

Автоматическая переменная доступна от точки описания до конца блока, в котором она описана. Например,

```
if (<выражение>)
  {
    int i;
    <операторы>
  }
```

Описание переменных внутри блоков в некоторых случаях делает программу более наглядной, облегчает поиск ошибок.

Время жизни автоматических переменных – время работы функции.

Регистровый класс (register)

К регистровому классу можно отнести автоматические переменные и параметры функций, предварив их описание ключевым словом *register*. Например, *register int i = strlen(s)*;.

Если переменная относится к регистровому классу, то программист надеется, что она будет размещена в одном из регистров, и что операции над ней будут выполняться быстрее.

Для регистровых переменных невозможна операция взятия адреса. В остальном регистровые переменные не отличаются от автоматических.

Аппаратные средства накладывают ограничения на количество регистровых переменных и их типы. Типами регистровых переменных, как правило, могут быть целые типы, а в некоторых реализациях только беззнаковые. Если число регистровых переменных невелико или их типы недопустимы, то компилятор игнорирует слово *register* и размещает их, как автоматические переменные, в стеке.

Внешний класс (extern)

В Си можно описывать переменные вне функций: до описания функций или между ними. Эти переменные называются внешними и относятся к классу

extern. Область действия внешних переменных – часть программы от точки описания до конца файла, в котором они описаны, время жизни – время работы программы. Внешние переменные размещаются в сегменте данных. По умолчанию внешние переменные инициализируются нулями, их можно инициализировать явно, но только константными выражениями.

Внешние переменные можно использовать до их описания в том же файле, или в другом файле этой программы, сделав *extern*-объявление:

extern<тип><список переменных без инициализации >;.

При этом память для таких переменных не выделяется. При объявлении одномерных массивов можно не указывать их размеры, а при объявлении многомерных массивов можно не указывать размер только по первому измерению.

Если в области действия внешней переменной используется автоматическая переменная с таким же именем, то её смысл понимается в соответствии с ближайшим предшествующим описанием.

```
int n = 10;
float f (float a)
{
 float n = 0.5;
  a = n; // a = 0.5
return a;
}
```

```
int n = 10;
float f (float a)
 {
   a = n; // a = 10.0
          return a;
 }
```

Статический класс (static)

Описание статических переменных имеет вид:

static <тип> <список переменных>;.

Статические переменные могут быть внутренними (описанными внутри функции) и внешними (описанными вне функций). По умолчанию статические переменные инициализируются нулями, но они могут явно инициализироваться только константными выражениями.

Область действия внешней статической переменной – часть программы от точки описания до конца файла, в котором она описана, а внутренней – от точки описания до конца блока, в котором она описана.

Внутренние статические переменные инициализируется только один раз при первом вызове функции, они сохраняют свои значения между вызовами этой функции. Поэтому внутреннюю статическую переменную можно использовать как счетчик, например, для определения глубины рекурсии.

Время жизни как внешних, так и внутренних статических переменных – время работы программы.

Внешние статические переменные являются глобальными переменными с ограниченной областью действия, что позволяет избежать побочных эффектов.

Классы памяти функций

Функции обычно являются внешними объектами. Область действия функции – от точки описания до конца файла. Использовать функцию до ее описания или в другом файле той же программы можно, описав прототип этой функции. Прототип представляет собой заголовок функции, но в списке параметров мож-

но описать только типы формальных параметров без указания имен параметров, так как при объявлении функции компилятору необходимо знать только имя функции, количество аргументов, их типы и тип возвращаемого функцией значения. Поэтому имена формальных параметров, если они есть в прототипе, не имеют никакого значения и игнорируются компилятором. Таким образом, объявление функции (прототип) имеет вид:

<тип> <идентификатор> ([<тип параметра>[,<список типов>]]); .

Например, *void f*(*int*, *int*);

Функция может быть отнесена к статическому классу, если ее заголовок предварить ключевым словом *static*:

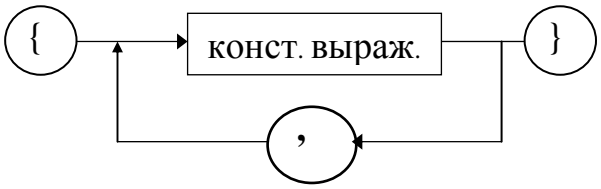
static <описание функции>

Статическая функция доступна *только* после ее описания и *только* до конца файла, в котором она описана.

МАССИВЫ И УКАЗАТЕЛИ

Инициализация одномерных массивов

Массивы можно инициализировать при их описании константными выражениями базового типа:



Способы инициализации одномерных массивов

1. Инициализация всех элементов массива.

int a[5] = {1,2,3,4,5};

2. Инициализация начальной части массива. При этом оставшимся не инициализированным элементам массива присваиваются нули.

int b[5] = {1,2,3}; // равносильно описанию *int b*[5] = {1,2,3,0,0};

3. Инициализация без указания размера массива. При этом размер массива определяется числом констант-инициализаторов.

int c[] = {1,2,3,4,5};// равносильно описанию *int c*[5] = {1,2,3,4,5};

Символьные массивы могут инициализироваться как числовые массивы, но последним элементом должен быть символ конца строки.

char s[10]={’a’, ‘b’, ‘c’, \0’};

Инициализировать символьный массив можно строковой константой, указывая или не указывая размер массива. В последнем случае размер будет равен длине строки +1.

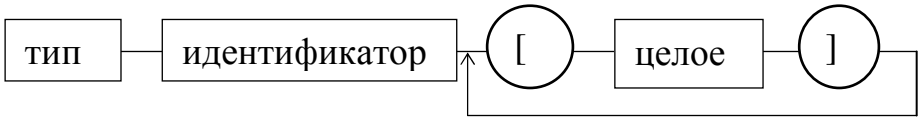
*char s*1[30]=”Университет”; // Описан массив размером 30.

*char s*2[]=”Университет”; // Описан массив размером 12.

В Си отсутствует проверка выхода за границы массивов. Можно выйти за границу массива и записать значение в какую-либо ячейку, не принадлежащую массиву. Проверка выхода за границы массива возлагается на программиста.

Многомерные массивы и их инициализация

Описание многомерного массива:



Примеры.

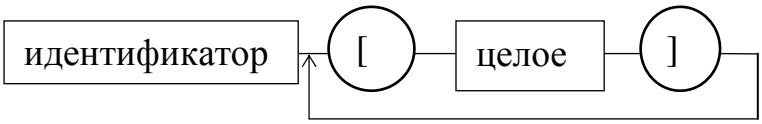
int a[5][3];

float b[4][3][5];

Согласно этим описаниям в памяти выделяется место для целочисленного двумерного массива *a* размером 5×3 и вещественного трехмерного массива *b* размером 4×3×5.

Массив *a* можно рассматривать как одномерный массив размером 5 с базовым типом «одномерный целочисленный массив размером 3», а массив *b* – как одномерный массив размером 4 с базовым типом «двумерный вещественный массив размером 3×5».

Обращение к элементам многомерного массива:



Многомерные массивы хранятся в памяти так же, как и в языке Паскаль: быстрее всего изменяется последний индекс, затем предпоследний и так далее.

Способы инициализации многомерных массивов.

При инициализации многомерных массивов инициализаторы по каждому измерению могут заключаться в фигурные скобки или элементы могут быть перечислены и заключены в одни фигурные скобки.

При описании многомерного массива можно не указывать размер по первому измерению, но в этом случае он должен инициализироваться при описании, и его размер по первому измерению определяется количеством инициализаторов.

Примеры.

int a[3][2] = { {1, 2}, {3, 4}, {5, 6} }; //или { 1, 2, 3, 4, 5, 6 }

int b[3][2] = { {1}, {2}, {3} }; //равносильно { {1, 0}, {2, 0}, {3, 0} }

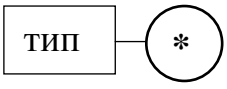
int c[3][2] = { 1, 2, 3 }; //равносильно { {1, 2}, {3, 0}, {0, 0} }

int d[][2] = { {1, 2}, {3, 4}, {5, 6} }; // равносильно *int d*[3][2]

int e[][2] = { 1, 2, 3, 4, 5, 6, 7};// равносильно *int e* [4][2]

Указатели

Если тип имеет имя, то описание типа указатель имеет вид:



Тип и * рекомендуется разделять пробелом. Тип является базовым типом указателя. Значением переменной-указателя является адрес, по которому находится значение базового типа.

Например,

*int *pi*, **qi*; // *pi*, *qi*; – неинициализированные указатели на целые.

void **p=NULL*; // *p* – нетипизированный указатель.

Переменная *p* инициализирована константой *NULL* (пустой указатель), имеющей значение ноль. Константа *NULL* определена в файле <*stdio.h*>.

Для совместимости типизированных указателей по присваиванию требуется, чтобы их базовые типы были одинаковыми. Нетипизированному указателю можно присваивать значение типизированного указателя. Если типизированному указателю нужно присвоить значение указателя с другим базовым типом, то следует использовать явное преобразование типа. Константу *NULL* можно присвоить указателю с любым базовым типом.

Типизированные указатели можно инициализировать адресами ранее описанных переменных базового типа.

int m, *n*;

pi =&m; // & – знак операции взятия адреса

qi=&n;

После такой инициализации значения *pi* и *qi* содержат адреса переменных *m* и *n* соответственно. Операторы *scanf*(“%*i*”, *&m*); и *scanf*(“%*i*”, *pi*); равносильны, так же как и операторы *printf*(“%*i*”, *m*); и *printf* (“%*i*”, **pi*); (* – знак операции разыменования).

Адресная арифметика

С указателями можно выполнять не только операции присваивания и разыменования, но и при определенных условиях операции сравнения и ряд арифметических операций. Сравнение всегда выполняется корректно, если указатели ссылаются на элементы одного и того же массива, Любой указатель можно сравнить с *NULL* только на равенство или неравенство.

Рассмотрим указатели, которые ссылаются на элементы одного массива.

*int *p*, **q*;

int a[10]={3. 7, 5, 8, 9};

p = a; // то же, что и *p =&a*[0];

q = &a[3];

Переменным *p* и *q* присвоены адреса различных элементов массива, поэтому значения выражений *p*!=*q, p < q*, *p*!= *NULL* равны 1 (истина).

К указателю можно прибавить или из него можно вычесть целое число. Значением выражения *p + n* является адрес *n*-го элемента правее или левее элемента, на который установлен указатель *p* в зависимости от знака *n*: правее при *n* больше 0, левее при *n* меньше нуля. Выражение *p–n* рассматривается как *p* +(– *n*).

Для описанных выше указателей значение выражения *(*p*+2) равно 5, *(*q*– 2) равно 7.

Переменные-указатели можно инкрементировать и декрементировать:

++ *p; p++;* // смещение/указателя к следующему элементу массива

– –*p; p– –;* // смещение/указателя к предыдущему элементу массива

Оператор ++ *p;* равносилен *p*+=1;

Постфиксные инкремент и декремент имеют более высокий приоритет по сравнению с операцией разыменования: **p– –* ≡ *(*p– –*).

При выполнении арифметических операций указатель *p* не должен выйти за границы массива. Минимальное значение указателя *p* равно адресу начального элемента, максимальное – адресу последнего элемента +1.

Указатели, ссылающиеся на элементы одного и того же массива можно вычитать. Результат такой операции равен числу элементов массива между уменьшаемым и вычитаемым со знаком плюс или минус. Если первый адрес меньше, то результат отрицательный. Так для приведенного выше фрагмента *p* – *q* равно –3, а *q* – *p* равно +3.

Связь массивов и указателей

Имя массива является константным указателем на базовый тип массива. Значение этого указателя равно адресу начального элемента массива. Так как имя массива является типизированным константным указателем, оно может быть операндом тех операций над указателями, которые его не изменяют.

Пример.

int a[10];

$a \equiv \&a[0]$, $*a \equiv a[0]$.

$a + 1 \equiv \&a[1]$, $*(a+1) \equiv a[1]$.

$a + i \equiv \&a[i]$, $*(a + i) \equiv a[i]$.

Базовым типом многомерного массива является массив на единицу меньшей размерности. Так, базовым типом двумерного массива является одномерный массив. Базовым типом трехмерного массива является двумерный массив.

Пример.

int b[10][5];

Типом константы *b* является указатель на одномерный массив из 10 целых чисел. Значение *b* равно адресу начального элемента, то есть адресу массива *b*[0].

$b \equiv \&b[0]$, $*b \equiv b[0]$.

Так как *b*[0] является одномерным массивом, то *b*[0] – указатель на целое, равный адресу начального элемента этого массива &*b*[0][0]. После разыменования этого указателя на целое получим значение элемента массива.

$**b \equiv *b[0] \equiv b[0][0]$.

Заметим, что, хотя типы *b* и **b* разные, но их значения совпадают.

Прибавляя к *b* целое *i*, получаем &*b*[*i*], – адрес *i*-го одномерного массива (адрес *i*-й строки матрицы *b*), к указателю прибавляется *i**(10**sizeof*(*int*)) байтов. Прибавляя к **b* целое *j,* получаем &*b*[0][*j*], – адрес *j*-го элемента начальной строки.

$*(b + i) \equiv b[i]$ – указатель на целое, адрес элемента *b*[*i*][0].

$*(b + i)+ j \equiv b[i] + j \equiv \&b[i][j]$.

$*(*(b + i)+ j) \equiv *(b[i]+ j) \equiv b[i][j]$.

Компилятор каждое выражение с индексами преобразует в выражение с указателями. Хотя программисту удобнее использовать выражения с индексами.

Индексировать можно и переменные-указатели вместо их разыменования.

int, **p*;

int a[10] = {3. 7, 5, 8, 9},

**p = a*;

Теперь к элементам массива *a* можно обращаться, заменив *a* на *p*.

a[*i*] ≡ *p*[*i*] ≡ *(*a*+*i*) ≡ *(*p*+*i*).

После переноса указателя индексация для него начинается с 0.

p+=2;

p[0] ≡ *a*[*2*].

Массивы указателей и указатели на массивы

Описание массива указателей имеет вид:

*int *p*[3]; // Массив из трех указателей на целое.

В качестве примера использования массива указателей опишем функцию, которая возвращает название месяца по его номеру или сообщение об ошибочном номере месяца. Массив указателей статический, поэтому инициализация его будет выполнена один раз при первом вызове функции.

```
char *month_name(int n)
 {
  static char * name[]= //Описан массив указателей на символы
   {
    “Недопустимый номер месяца”,
    “январь”,
    “февраль”,
    “март”,
    “апрель”,
    “май”,
    “июнь”,
    “июль”,
    “август”,
    “сентябрь”,
    “октябрь”,
    “ноябрь”,
    “декабрь”
   }
  return n<1 || n>12 ? name[0] : name[n];
 }
```

Описание указателя на массив имеет вид:

int (**q*)[3]; // Указатель на массив из трёх целых

float a[3][4][5], (**pa*)[4][5]=*a*;

Типы переменных *a* и *pa* одинаковы (указатель на двумерный массив размером 4×5).

Тип size_t

size_t – базовый беззнаковый целочисленный тип языка Си и Си++. Он является типом результата, возвращаемого операцией *sizeof*. Размер типа выбирается таким образом, чтобы в него можно было записать максимальный размер теоретически возможного массива любого типа. На 32-битной системе *size_t* будет занимать 32-бита, на 64-битной – 64-бита. Другими словами, в переменную типа *size_t* может быть безопасно помещен указатель. Тип *size_t* обычно применяется для параметров циклов, индексации массивов, хранения размеров, адресной арифметики. Тип *size_t* описан в файле <*strdef.h*>.

Передача массивов функциям в качестве параметров

Так как имя одномерного массива является указателем на базовый тип, то такой параметр можно передать двумя способами: как указатель на базовый тип или как массив, не указывая в квадратных скобках размер массива. При этом размер массива нужно передать как параметр типа *size_t*. Функция получает полный доступ к элементам массива. Например, заголовок функции ввода целочисленного массива может иметь вид

void read_arr(*int a*[], *size_t n*) или *void read_arr*(*int *a*, *size_t n*)

При передаче функциям строк размер символьного массива передавать не нужно, за исключением тех алгоритмов, в которых контролируется выход за границы массива.

Опишем три варианта тела функции, которая копирует строку *t* в строку *s* и возвращает указатель на начальный символ строки *s*.

*char * strcpy*(*char *s*, *const char *t*);

```
{                       {                       {
  size_t i = 0;           size_t i = 0;           while (*s++=*t++);
  while (s[i]=t[i])       while (s[i++]=t[i++]):  return(s);
    i++;                  return(s);              }
  return(s);              }
}
```

Передать многомерный массив функции можно без указания размера по первому измерению, размеры по всем остальным измерениям размеры должны быть указаны, или можно передать указатель на массив на единицу меньшей размерности с указанием размеров по каждому из оставшихся измерений. При этом размер по первой размерности передается как параметр типа *size_t*.

Например, заголовок функции вывода матрицы размером *m*×*N*, где *N* – константа, может иметь вид:

void write_arr(*int a*[][N],*size_t m*) или

void write_arr(*int* (**a*)[N],*size_t m*);

МОДЕЛИ ПАМЯТИ В СИ

В языке Си размер памяти, занимаемой указателем, зависит от модели памяти, задаваемой при компиляции программы. Модель памяти определяет объём памяти, выделяемый для данных и для кода. Существует шесть моделей памяти.

Крошечная (*tin*y). Для кода, статических данных, динамически размещаемых данных (кучи) и стека выделяется 64 Кб. Код и данные адресуются только смещением. Указатель на данные и указатель кода занимает 2 байта (близкие указатели).

Маленькая (*small*). Под код программе отводится сегмент размером 64 Кб. Статические данные, куча и стек тоже занимают 64 Кб. Указатели на данные и указатели кода занимают по 2 байта (близкие указатели), но их сегменты разные. Эта модель установлена по умолчанию. Она подходит для небольших задач.

Компактная (*compact*). Используется для небольших программ с большим объемом данных. Код занимает до 64 Кб, а данные – до 1 Мб. Причем статическим данным выделяется 64Кб, а объем стека, как и для всех моделей, не превышает 64Кб. Код адресуется двумя байтами, а для адресации данных используются четырехбайтовые (дальние) указатели.

Средняя (*medium*). Размер памяти для кода ограничен 1 Мб. Это означает, что в коде используются дальние указатели. Стек, куча и статические данные занимают 64 Кб. То есть данные адресуются близкими указателями. Эту модель рекомендуется использовать для больших программ с небольшим количеством данных.

Большая (*large*). В этой модели, как код, так и данные занимают до 1 Мб. Но объем статических данных не превышает 64 Кб. Код и данные адресуются дальними указателями. Ни одна отдельная единица данных не может занимать больше 64 Кб.

Огромная (*huge*). Эта модель аналогична большой модели, но объем статических данных может превышать 64 Кб.

В операционной системе *Windows* можно использовать только четыре модели памяти: крошечную и огромную использовать нельзя.

ДИНАМИЧЕСКИЕ ПЕРЕМЕННЫЕ

Функции динамического распределения памяти

Интерфейс для работы с динамической памятью находится в файле <*stdlib.h*>.

*void *malloc*(*size_t n*) выделяет в динамической памяти блок размером *n* байтов и возвращает нетипизированный указатель на начальный байт выделенного блока. Выделенная память содержит мусор (не инициализируется).

*void *calloc*(*size_t n*, *int size*). выделяет *n* блоков каждый размером *size*. Выделенная память инициализируется нулями.

*void *realloc*(*void *p*, *size_t n*) используется для изменения размера блока памяти, выделенного ранее, по адресу *p. n* – новый размер в байтах. Если такое перевыделение выполнено успешно, функция возвращает нетипизированный указатель на вновь выделенный блок. Причем расширение блока выполняется с сохранением значений исходного блока, добавленные байты не инициализируются нулями (содержат мусор). Если новый блок меньше исходного, то значения исходного блока сохраняются. Если доступной памяти недостаточно, чтобы

расширить блок до заданного размера, исходный блок останется без изменений по адресу *p*, а функция возвращает значение *NULL*.

void free(*void *p*) освобождает выделенную память по адресу *p*. Размер освобождаемой памяти не указывается, так как он сохраняется для каждого указателя при выделении памяти.

Спецификатор typedef

В языке Си любой тип можно описать контекстно. Но если описание типа громоздкое или программист хочет заменить предопределенное имя типа на более привычное для него имя, то спецификатор *typedef* позволит дать новое имя для существующего типа следующим образом:

typedef <описание, аналогичное описанию переменной>;

Идентификатор в этом описании является именем вводимого в рассмотрение типа, а не именем переменной. Рекомендуется давать типам осмысленные имена и предварять их префиксом *t_*. Это позволит отличать имена типов от имен других программных объектов.

Использование *typedef* может помочь при создании более легкого для чтения и более переносимого кода.

Примеры.

typedef int boolean;

Имя типа *boolean* можно использовать для описания переменных, которые, по сути, являются логическими, так как специального логического типа в языке Си нет.

typedef float (**t_pmatr*)[*M*][*N*];

t_pmatr – тип «указатель на матрицу размером *M*×*N*». Теперь можно описывать переменные и передавать функциям параметры типа *t_pmatr*.

float a[10][M][N]; // по адресу *a* размещен массив матриц.

t_pmatr pa=a; // Типы *pa* и *a* одинаковы.

void read_mart_arr(*t_pmatr a*, *size_t n*) – заголовок функции для ввода массива из *n* матриц размером *M*×*N*.

Размещение массивов в динамически распределяемой области памяти

Размещение в куче одномерного массива размера *n* с базовым типом *t_base* выполняется функциями *malloc* или *calloc*. Например,

*t_base *pa* = (*t_base**)*malloc*(*n * sizeof*(*t_base*));

*t_base *pa* = (*t_base**)*calloc*(*n*, *sizeof*(*t_base*));

Опишем функцию *del_el* для удаления из динамического целочисленного массива размера **pn*, размещенного по адресу *pa* элементов, удовлетворяющих условию, определяемому функцией *int f*(*int el*). Функция *del_el* возвращает указатель на преобразованный массив и записывает его новый размер по адресу *pn.*

```
int del_el(int *pa, int *pn)
  {
    int i, j;
```

```
for (i=j=0; i < *pn; i++)
  if(!f(pa[i],)
    pa[j++]= pa[i];
*pn= j;
return (int*)realloc(pa, j*sizeof(int));
```

Способы размещения матриц в динамической памяти

Матрицы могут быть размещены в динамической памяти различными способами.

1. Матрица, длина строки которой задана константой *N*, а количество строк равно *m,* может быть размещена в динамической памяти с помощью указателя на строку матрицы.

```
<б.т.> (*pa)[N]=((<б.т.>*)[N])calloc(m, N*sizeof(<б.т.>));
```

Здесь и далее сокращение *б.т.* обозначает *базовый тип*.

Обращение к элементу *i*-й строки *j*-го столбца матрицы имеет вид: *pa*[*i*][*j*].

2. Матрица размером *m*×*n* может быть размещена в куче как одномерный массив размером *m*×*n*×(размер базового типа) с помощью указателя на базовый тип элементов матрицы.

```
<б.т.>*pa=(< б.т.>*)calloc(m, n*sizeof(<б.т.>));
```

Обращение к элементу *i*-й строки *j*-го столбца матрицы имеет вид: *pa*[*i***n*+*j*].

3. Матрица размером *m*×*n* может быть размещена в куче одним блоком размером *m*×*n*×(размер базового типа) и массива из *m* указателей на начальные элементы строк матрицы.

```
<б т.>*pa = (<б.т.>*)calloc(m, n*sizeof(<б.т.>)); /*размещение матрицы*/
<б. m.>**pr = (<б. m.>**)calloc(m, sizeof(<б. m.>*)); /*размещение массива
указателей на базовый тип элементов матрицы */
for (size_t i=0; i<m;++)
   pr[i]=pa+i* n*sizeof(<б. m.>); /* инициализация массива указателей адре-
   сами начальных элементов строк матрицы.*/
```

Обращение к элементу *i*-й строки *j*-го столбца матрицы имеет вид: *pa*[*i*][*j*].

4. Матрица размером *m*×*n* может быть размещена в куче в *m* блоках и массива указателей на начальные элементы строк.. Каждый блок размером *n*×(размер базового типа) хранит одну строку матрицы.

```
<б.т.>**p=(<б.т.>**)calloc(m, sizeof(<б.т.>*)); /* размещение массива ука-
зателей на базовый тип элементов матрицы.*/
for (<б.т.> i=0; i < m; i++)
   p[i]=(<б.т.>*)calloc(n, sizeof (<б.т.>)); /* размещение строк матрицы и
  инициализация массива указателей адресами строк матрицы.*/
```

Обращение к элементу *i*-й строки *j*-го столбца матрицы имеет вид: *pa*[*i*][*j*].

При размещении матриц с помощью массива указателей массив указателей может находиться в куче, или в статической памяти, или в стеке. В приведенных выше примерах массивы указателей размещались в куче.

Свободные массивы

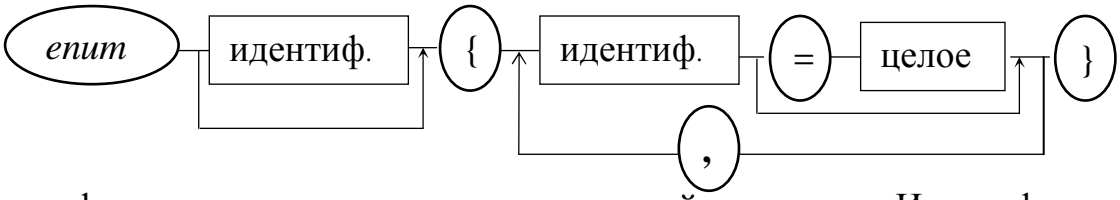
Свободными массивами будем называть двумерные массивы, длины строк которых могут быть различными. Свободный массив представляет собой массив указателей на начальные элементы одномерных массивов разного размера с общим базовым типом.

Свободные массивы удобно использовать для хранения симметричных матриц, сохраняя в каждой *i*-й строке *i*+1 первых ее элементов.

В свободных массивах можно хранить слова вводимого текста. Первоначально слово считывается в буфер достаточного размера, а затем помещается в свободный массив. Для этого выделяется для соответствующего указателя массива указателей блок памяти по размеру слова, и слово копируется в этот блок.

ПЕРЕЧИСЛЯЕМЫЕ ТИПЫ

Перечисляемый тип задает множество именованных констант, что позволяет улучшить читабельность программы, Определение перечисляемого типа имеет вид:



Идентификатор после *enum* называют меткой или тегом. Идентификаторы в фигурных скобках являются именами констант, их можно использовать в конструкциях, требующих наличия констант, например, при описании размеров внешних и статических массивов.

После описания типа можно перечислить имена переменных. Например,

enum seasons {*SPRING, SUMMER, AUTUMN, WINTER*} *s*1, *s*2;

Если потребуется описать переменные перечисляемого типа, описанного ранее, или передать их функции, то в качестве типа нужно указать *enum* <метка>.

По умолчанию, то есть, если константы не инициализируются явно, первой перечисляемой константе присваивается значение 0, а каждой следующей – значение на единицу большее, чем предыдущей.

Рассмотрим функцию, которая возвращает русское название времени года.

```
char * season _name(enum seasons s)
{
  static char * season_name[]=
  {
    “Зима”,
    “Весна”,
    “Лето”,
    “Осень”,
  }
  return season_name[s];
}
```

Часто перечисления используются только для задания набора именованных констант, например, для задания логических констант *FALSE*= 0 и *TRUE*=1. В таком случае метка может отсутствовать.

enum {*FALSE*, *TRUE*};

При отсутствии метки переменные перечисляемого типа можно описать только непосредственно в операторе определения перечисляемого типа.

enum {*RED*, *YELLOW*, *GREEN*} *s*1, *s*2;

Если какой-либо константе присвоено значение, то каждой следующей присваивается значение на единицу большее предыдущей.

enum days {*MON*=1, *TUES*, *WED*, *THUR*, *FRI*, *SAT*, *SUN*} *week*;

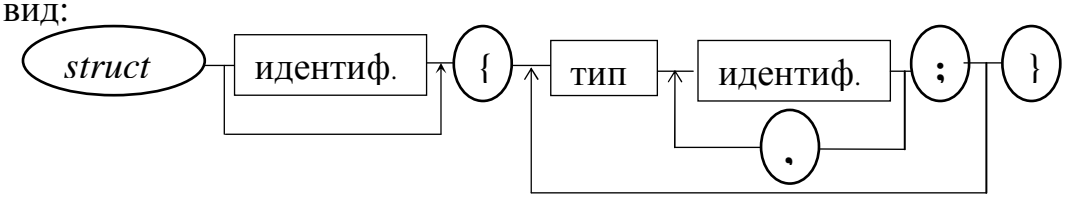
В приведенном выше примере дни недели нумеруются с 1.

Увеличение на 1 предыдущего значения происходит до тех пор, пока очередной константе не будет присвоено значение явно.

Переменные перечисляемого типа являются обычными целочисленными переменными. Компилятор не отслеживает, присвоены им значения констант этого типа или значение, не принадлежащие этому типу.

СТРУКТУРЫ

Структуры в языке Си аналогичны записям языка Паскаль. Поля записей языка Паскаль называются членами структур в Си. Описание типа структура имеет вид:



Идентификатор после *struct* – метка или тег. Идентификаторы в фигурных скобках являются именами членов.

struct coord { *float x, y, z*; };

Если метка есть в описании структуры, то описать переменные можно в любом подходящем месте программы в виде:

struct <метка> <список переменных>;.

Например, *struct coord a, b, c*;

При отсутствии метки описать переменные-структуры можно только непосредственно после описания типа:

Структуры можно инициализировать так же, как и массивы.

struct date{*int day*, *month*, *year*;} *d*={25, 1. 2018};

Если некоторые члены структур являются структурами или массивами, то их инициализаторы заключаются в свои фигурные скобки.

```
struct student
{
   char * name[20];
   struct date b_day;
   marks[3];
}={“Иванов”, {12, 7, 2000},{5, 4, 3}};
```

Над структурами одинакового типа определена операция присваивания. Структуры можно передавать функциям в качестве параметров, и функции могут возвращать структуры. Обращение к членам структуры такое же, как в Паскале. Операция выделения члена структуры с помощью знака . (точка) называется *прямым селектором* и имеет вид:

<имя структуры> .<имя члена>

```
struct compl
{
   float Re, Im;
 }z1, z2;
struct compl compl_sum (struct compl struct compl)
  {
   struct compl z3;
   z3.Re = z1.Re + z2.Re;
   z3. Im = z1.Im + z2.Im;
   return z3;
  }
```

К структурам, как и членам структур, применима операция взятия адреса.

Размер структуры не всегда равен сумме размеров ее членов вследствие выравнивания объектов разного размера по размеру машинного слова. Например, структура *struct {char c; int i}* занимает 8 байтов, если размер машинного слова равен 4 байтам.

Указатели на структуры

Указатель на структуру описывается обычным способом. Например, *struct date *pd;*.

Инициализировать указатель на структуру можно адресом ранее описанной переменной (*pd=&d;*) или с помощью функций размещения структуры в куче (*pd=(struct date*)malloc(sizeof(struct date));*.

Обратиться к члену структуры в этом случае можно одним из двух способов: используя операцию разыменования, а затем прямой селектор или используя операцию *косвенный селектор*.

Обращение к члену структуры с помощью операции косвенный селектор имеет вид: <указатель на структуру> **–>**<имя члена>.

В первом случае обращение к члену – *(*pd).day*, во втором – *pd –>day*.

Операции прямой и косвенный селектор имеют самый высокий приоритет, такой же, как и выделение элемента массива (квадратные скобки) и вызов функции (круглые скобки). Вторую строку в таблице приоритетов занимают все остальные унарные операции. С учетом сказанного, в выражении *(*pd).day* круглые скобки обязательны.

Битовые поля структур

Битовое поле – последовательность смежных битов внутри машинного слова. Битовые поля позволяют экономить память, так как в одном машинном слове может храниться несколько значений. Битовые поля обеспечивают удобный доступ к отдельным битам и группам битов данных. Это позволяют и побитовые операции, но с использованием битовых полей такие действия выглядят нагляднее.

Битовыми полями могут быть члены структуры. Описание битового поля:

Целое определяет ширину битового поля, оно может принимать значения от 0 до длины машинного слова в битах.

Битовые поля должны иметь типы *int* или *unsigned*. Битовые поля длины 1 должны объявляться как *unsigned*, поскольку 1 бит не может быть знаковым.

Если имя какого-то битового поля отсутствует, то это поле недоступно.

Битовое поле должно размещаться в одном машинном слове. Если в текущем машинном слове недостаточно битов для очередного поля, то оно размещается в следующем машинном слове.

Если ширина поля равна 0, то происходит выравнивание по границе следующего слова.

В зависимости от реализации битовые поля в машинном слове могут размещаться справа налево или наоборот.

Если структура имеет битовые поля, то они должны перечисляться в начале описания типа.

```
struct {
        unsigned f1:1;
        int f5:5;
        unsigned f2:2
        char * err[20];
       }u;
```

Доступ к элементам битового поля осуществляется так же, как и доступ к обычным членам структуры, например, *u. f*5.

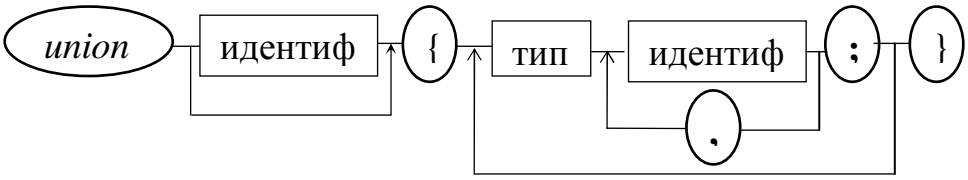
Над битовыми полями допустимы те же операции, что и над целочисленными значениями, кроме операции взятия адреса.

ОБЪЕДИНЕНИЯ

В отличие от записей языка Паскаль, в структурах языка Си нет вариантной части. Роль, аналогичную вариантной части, в Си играют объединения.

Объединение – это объект, позволяющий нескольким переменным различных типов в разные моменты времени занимать один и тот же участок памяти.

Описание типа объединение похоже на описание типа структура:



Как и переменные-структуры, описать переменные-объединения можно непосредственно после описания типа объединение или с помощью оператора описания вида *union*<метка> <список переменных> если тип имеет метку.

Все члены объединения хранятся по одному адресу. Поэтому в каждый момент времени в памяти хранится значение только одного члена объединения.

Память для объединения выделяется по размеру члена, занимающего максимальный объем.

Для доступа к членам объединения используется прямой или косвенный селектор, как и для доступа к членам структур.

Объединения могут быть полезны для экономии памяти при наличии множества объектов и/или при ограниченном объеме памяти. Однако для их правильного использования требуется повышенное внимание, поскольку нужно всегда сохранять уверенность, что используется последний записанный член. Наиболее распространенное решение этой проблемы – заключение объединения в структуру вместе с дополнительным членом структуры (селектором), значение которого определяет, какой из членов хранятся в объединении в текущий момент, Такая структура аналогична записи с вариантами языка Паскаль. При обработке объединения удобно использовать оператор-переключатель.

Объединения можно использовать при необходимости преобразования типов, поскольку можно обращаться к данным, хранящимся в объединении как к значениям разных типов. Например, чтобы проанализировать, как хранится вещественное число типа *float* в оперативной памяти, опишем объединение с двумя полями:

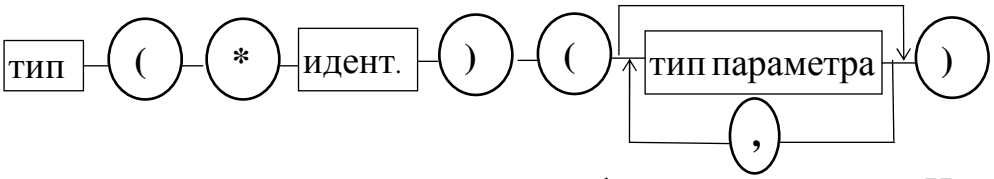
union {*float f*; *unsigned char a*[4];} *u*;.

После инициализации члена *u.f* выведем с помощью побитовых операций в двоичном виде содержимое каждого элемента массива *u.a.*

ПЕРЕДАЧА ФУНКЦИЙ ФУНКЦИЯМ В КАЧЕСТВЕ ПАРАМЕТРОВ

Код функции хранится в области памяти, называемой сегментом кода. Указатель на функцию – это переменная, которая содержит начальный адрес кода функции.

Описание переменной-указателя на функцию имеет вид:



Вначале указывается тип возвращаемого функцией значения. Идентификатор – имя указателя на функцию с перечисленными типами параметров. Скобки вокруг *<идент> необходимы, так как при их отсутствии описание будет рассматриваться как прототип функции, возвращающей указатель.

Если перед таким описанием поместить спецификатор *typedef*, то идентификатор будет именем типа «указатель на функцию».

typedef float(**t_pf*)(*float*);/* *t_pf* – имя типа «указатель на вещественную функцию вещественного аргумента».*/

float (**pf*)(*float*); // *pf* – указатель на вещественную функцию вещественного аргумента.

float f(*float*); // *f* – прототип вещественной функции вещественного аргумента.

Для работы с указателем на функцию его нужно инициализировать адресом кода функции такого же типа. Таким адресом является имя функции. Имя функции является константой-указателем.

Выражение &<имя функции> и имя функции имеют одно и то же значение, поэтому можно инициализировать указатель двумя способами: *pf = &f;* или *pf = f;* .

После такого присваивания вызвать функцию с помощью указателя можно так же двумя способами: *(*pf)(x)* или *pf(x)*. В последнем случае выполняется автоматическое приведение типа.

Большое значение указатели на функции имеют в системном программировании. В прикладном программировании указатели на функции используют для передачи функций функциям в качестве параметров.

В качестве примера опишем функцию, которая возвращает корень уравнения $f(x) = 0$ на интервале (a, b) точностью *eps*, где $f(x)$ – непрерывная на $[a, b]$ функция, имеющая на концах промежутка разные знаки

```
float f (float (*pf)(float), float a, float b, float eps)
  {
   float c;
   do
     {
      c = (a+b)/2;
      if (pf(a)*pf(c) < 0)    // можно if ((*pf)(a)(*pf)(c) < 0)
        b = c;
      else
        a = c;
     }
   while (b–a > eps)
   return c;
  }
```

Вызов этой функции для нахождения корня уравнения $g(x) = 0$ может иметь вид *f (&g, 1.3, 5.7, 1E–5)* или *f (g, 1.3, 5.7, 1E–5)*.

Массивы указателей на функции

Кроме одиночных указателей на функции можно рассматривать массивы указателей на функции. Описание массива указателей на функцию имеет вид:

<тип> (*<имя массива>[размер]) (<список типов параметров>);.

Рассмотрим программу, в которой определяются некоторые характеристики массива *a* размера *n*. В программе описаны функции со следующими прототипами.

float max (*float *a*, *size_t n*); /*Возвращает. максимальный элемент массива *a* размера *n*.*/

float min (*float *a*, *size_t n*); /*Возвращает. минимальный элемент массива *a* размера *n*.*/

float aver(*float *a*, *size_t n*);/*Возвращает среднее арифметическое. элементов массива *a* размера *n*.*/

float num_max (*float *a*, *size_t n*); /*Возвращает количество максимальных элементов массива *a* размера *n*.*/

float num_min (*float *a*, *size_t n*); /*Возвращает количество минимальных элементов массива *a* размера *n*.*/

```
int main()
{
 char s[5]={“ max”, “min”, “aver”, “num_max”, ”num_min“};
 size_t i;
 float((*pf_arr)[5])(float *a, size_t n) = {max, min, aver, num_max, num_min};
 /* Описан и инициализирован массив указателей на функции.*/
 for (i=0; i<5; i++)
      printf(“%s =%f\n”, s[i], (*pf_arr)[i])(float *a, size_t n));
 return 0;
}
```

ФАЙЛЫ В СИ

Физический файл – это набор данных, размещенный на внешнем носителе, имеющий имя и рассматриваемый в процессе обработки как единое целое. В файлах находятся данные, предназначенные для длительного хранения.

Система ввода и вывода в Си поддерживает постоянный интерфейс, независимо от устройства, откуда поступают или куда выводятся данные. Это могут быть дисковые накопители, терминалы и т. п. Хотя каждое устройство имеет свои отличительные черты, система ввода-вывода преобразует их в единое логическое устройство, называемое потоком. Поскольку потоки не зависят от устройств, одни и те же функции могут записывать данные в файл на диске или выводить, например, на экран монитора, или считывать данные из файла или с консоли.

Потоки бывают двух видов файлов: текстовые и бинарные. Текстовые потоки представляют собой последовательности символов таблицы *ASCII*, разбитые на строки. Каждая строка в файле заканчивается двумя символами с кодами 10 («перевод строки») и 13 («возврат каретки»). Бинарные (двоичные) потоки – последовательности байтов. Они создаются и обрабатываются программно. Интерфейс для работы с потоками находится в файле *stdio.h.*

Для работы с потоком используется файловая переменная (логический файл), которая является указателем на структуру типа *FILE.*

*FILE *f*; // Описана файловая переменная.

Структура типа *FILE* содержит такую информацию, как имя физического файла, его размер в байтах, номер открытого файла, размер и адрес буфера ввода и вывода, режим открытия файла, достигнут ли конец файла, коды ошибок и т.п. Прямого доступа к членам структуры *FILE* нет.

Связь потока с файлом выполняется функцией

*FILE *fopen*(*char** <имя физического файла>, *char**<режим>);.

При открытии файла с ним связывается поток ввода-вывода. Выводимая информация записывается в поток, вводимая информация считывается из потока. Указатель, который возвращает функция *fopen*, используется для всех операций с потоком. Если поток не открыт функция возвращает пустой указатель *NULL*.

Для программиста открытый поток представляется как последовательность считываемых или записываемых данных.

Режимы открытия файла:

“*r*” Существующий файл открывается для чтения.

“*w*” Файл открывается для записи. Если файл существует, то его содержимое стирается, если файл не существует, то он создаётся и открывается для записи.

“*a*” Существующий файл открывается для дозаписи в конец. Если файл не существует, то он создаётся и открывается для записи.

“*r+*” Существующий файл открывается для чтения и изменения, но без изменения размера файла.

“*w+*” Файл открывается для записи и изменения. Если файл существует, то его содержимое теряется; Размер файла может увеличиться.

“*a+*” Существующий файл открывается для добавления в конец файла и изменения. Если файл не существует, то он создается и открывается для чтения и записи.

Поток может быть открыт в текстовом или бинарном режиме.

Текстовый режим отличается от бинарного тем, что при открытии текстового файла пара символов «перевод строки» и, «возврат каретки» заменяется одним символом «перевод строки» для всех функций чтения данных из файла, а для всех функций записи данных в файл символ «перевод строки» заменяется двумя символами: «перевод строки», «возврат каретки». Следовательно, нет взаимно однозначного соответствия между считанными или записываемыми символами потока и символами на внешнем устройстве.

При открытии в бинарном режиме такое преобразование не производится.

По умолчанию потоки открываются в текстовом режиме. Для явного указания, что файл открывается в текстовом режиме, к режиму добавляется, буква *t*.. То есть выражения “*wt*” и “*w*” определяют один и тот же режим.

Если требуется открыть файл в бинарном режиме, к любому из перечисленных режимов добавляется, буква *b*. Например,

*FILE *f = fopen*(“*d*:*my_file*”, “*r+b*”);

Обратите внимание: обратный слеш (\), как специальный символ, в полном имени файла записывается дважды.

В начале выполнения программы автоматически открываются три стандартных текстовых потока: *stdin* (стандартный поток ввода), *stdout* (стандартный

поток вывода) и *stderr* (стандартный поток ошибок). Поток *stdin* связывается с клавиатурой и открывается для чтения. Потоки *stdout* и *stderr* связываются с дисплеем и. открываются для записи.

Потоки *stdin, stdout* и *stderr* можно использовать в любой функции, где требуется переменная типа *FILE**.

Не рекомендуется использовать имена конкретных файлов при открытии потока. Для того, чтобы программа была более гибкой, могла обрабатывать файлы с разными именами следует предложить пользователю ввести имя файла.

*FILE *f*;
char f_name[20];
gets(*f_name*);
f = fopen,(*f_name* , "*ab*");

После работы с файлом поток должен быть закрыт функцией *int fclose*(*FILE *f*);. Если поток успешно закрыт функция возвращает 0, иначе – ненулевое значение. Попытка закрытия уже закрытого файла или файла, который не был открыт, является ошибкой.

Функция *int fcloseall* (*void*); закрывает все открытые потоки ввода-вывода за исключением стандартных: *stdin*, *stdout*, *stderr*. Функция *fcloseall* возвращает общее число закрытых потоков, или *EOF*, если обнаружены какие-либо ошибки при закрытии файлов. Она не определена стандартом ANSI C.

В языке Си имеется возможность работы с временными файлами, которые нужны только в процессе работы программы и должны быть удалены после окончания ее работы. В этом случае используется функция *FILE* tmpfile* (*void*);.

Она создает на диске временный файл с правами доступа "*w+b*". После завершения работы программы или закрытия этого (временного) файла он автоматически удаляется.

Функция *char *tmpnam*(*char *name*); создает уникальное имя файла и сохраняет его в массиве *name*. Возвращает указатель на *name*. Это имя может быть использовано как имя временного файла без опасения изменить существующий файл.

Функция *int feof*(*FILE *f*); возвращает значение 1, если достигнут конец файла, связанного с потоком *f*, иначе – 0.

Функция *int ferror*(*FILE *f*); возвращает значение 1, если при работе с потоком *f* произошла ошибка или достигнут конец файла, иначе – 0. Определить, произошла ли ошибка или достигнут конец файла, поможет предопределенная глобальная переменная *int errno*. Ее значение равно нулю, если нет ошибок, иначе оно равно коду ошибки.

Функция *void perror*(*const char *str*); выводит сообщение об ошибке в поток *stderr*, то есть на экран. Сначала выводится строка *str*, за ней следует двоеточие и системное сообщение об ошибке, соответствующее значению *errno*. Если на момент вызова функции *perror* ошибок не было или информация о них удалена, то в качестве сообщения об ошибке будет выведено сообщение: *Success*.

Функция *int rename(char *old_name, char *new_name);* изменяет название файла или каталога со старого *old_name* на новое *new_name*. Оба имени должны быть или именами файлов, или именами каталогов.

Имя, заданное параметром *new_name*, не должно совпадать ни с одним из существующих в каталоге имен.

Функцию *rename* можно использовать для перемещения файла из одного каталога в другой, указав путь в аргументе *new_name*. Но файлы не могут быть перемещены с одного устройства на другое, например, с дисковода *A* на дисковод *B*. Каталоги можно переименовывать, но нельзя перемещать.

Функция *rename* возвращает 0 если переименование удачно и ненулевое значение в случае ошибки, при этом переменная *errno* устанавливается в одно из следующих значений

EACESS – файл или каталог *new_name* существует, или не может быть создан (неверный путь), или *old_name* является каталогом, а *new_name* определяет другой путь;

ENOENT – файл или каталог, заданные параметром *old_name*, не найдены;

ENOTSAM – попытка перемещения файла или каталога на другое устройство.

Функция *int remove(const char *fname);* удаляет файл или каталог с именем *fname*. Удаляемый каталог должен быть пустым. При успешном удалении функция возвращает 0, а в случае ошибки – ненулевое значение.

Изменение размера файла

int chsize(int handle, long size); – прототип функции, которая изменяет размер файла с дескриптором *handle*. Дескриптор – целое положительное число, связанное с каждым открытым потоком, Дескриптор открытого потока, хранящийся в структуре *FILE*, возвращает при успешном выполнении функция *int fileno (FILE *f);*, объявленная в файле *stdio.h.* В противном случае эта функция возвращает –1.

Прототип функции *chsize* объявлен в файле *io.h*. Режим, в котором открыт поток, должен допускать запись. После вызова функции *chsize* файл будет иметь размер *size* байтов. Функция может уменьшить или увеличить текущий размер файла. Если при этом размер файла увеличивается, то он дополняется нулевыми символами, если же он уменьшается, то в файле сохраняются *size* начальных байтов.

При успешном завершении *chsize* возвращает 0, а в случае ошибки функция возвращает –1, при этом переменной *errno* присваивается одно из следующих значений:

EACCES – отказано в доступе;

EBADF – неверный дескриптор файла;

ENOSPC – UNIX – не для *DOS*.

Функции ввода и вывода потоком

В языке Си имеется большой набор функций для ввода и вывода потоком, Потоки, с которыми работают функции ввода-вывода, буферизированы. Это означает, что при открытии потока с ним автоматически связывается опреде-

ленный участок оперативной памяти, который называется буфером. В буфер, предназначенный для чтения, переносится блок данных из файла, а затем функции ввода считывают данные из этого буфера. При записи данные записываются в буфер вывода, а когда буфер полон или вызвана фунция *fflush*, его содержимое сбрасывается в файл. Размер буфера по умолчанию задан константой *BUFSIZE*, которая определена в файле *stdio.h* как 512 (хотя программно ее можно изменить).

Ввод и вывод символов

Функция *int fgetc*(*FILE *f*); возвращает считанный из потока *f* символ как целое, или *EOF* в случае ошибки.

Функция *int fputc*(*int c, FILE *f*); записывает в поток *f* символ *c* и возвращает этот символ, а в случае ошибки возвращает *EOF*.

Вызов функций стандартного ввода и вывода символов на самом деле заменяется вызовом *fgetc*(*stdin*) и *fputc*(*stdout*).

Ввод и вывод строк

Функция *char* fgets*(*char *s*, *size_t max*, *FILE *f*); считывает из потока *f* символы и записывает их в строку *s* до тех пор, пока не будет прочитан символ «новая строка» ('\n'), который включается в строку, или пока не наступит конец потока, или не будет считан *max*–1 символ. В конец строки записывается нольсимвол (‘\0’). Функция возвращает адрес строки *s* или *NULL* в случае ошибки.

Функцию *fgets* рекомендуется использовать вместо стандартной *gets*, так как *gets* не выполняет проверку выхода за границы массива *s*. В этом случае в качестве потока следует указать стандартный поток *stdin.*

Например, ввести строку с клавиатуры в символьный массив *s* размером 200 с консоли можно оператором *fgets*(*s*, 200, *stdin*):

Функция *int fputs*(*char *s*, *FILE *f*); записывает строку *s* в поток *f*. При этом завершающий нулевой символ (т.е. символ конца строки ('\0')) не записывается.

При успешном выполнении функция *fputs* возвращает последний записанный символ, а в случае ошибки – *EOF*.

Форматный ввод и вывод

Функции форматного ввода *fscanf* и форматного вывода *fprintf* выполняются так же, как и стандартные функции *scanf* и *printf,* но первым параметром является файловая переменная.

int fscanf(*FILE *f*, <управляющая строка>, <параметры>);

int fprintf(*FILE *f*, <управляющая строка>, [<параметры>]);

Ввод и вывод данных в машинном представлении

Функции для чтения из файла *fread* и записи в файл *fwrite* позволяют программе обмениваться с внешними устройствами данными в машинном представлении, что аналогично работе с типизированными файлами в языке Паскаль.

Функция *size_t fread*(*void *buf*, *size_t size*, *size_t count*, *FILE* f*); считывает из потока *f count* блоков, каждый размером *size* и записывает по адресу *buf*. Воз-

вращает количество фактически считанных и записанных в *buf* блоков. Если возвращаемое значение отличается от количества запрашиваемых блоков, значит, произошла ошибка или был достигнут конец файла.

Функция *size_t fwrite(void *buf, size_t size, size_t count, FILE* f);* записывает в поток *f count* блоков каждый размером *size*, считанных по адресу *buf*. Возвращает количество записанных в поток блоков. Если возвращаемое значение отличается от количества запрашиваемых блоков, значит, произошла ошибка.

Приведем фрагмент программы для записи целочисленного массива в бинарный файл.

```
int a[10] = {23, 41, –15, 7, 87, 2, 158, 5, 34, –1};
FILE* f = fopen("d:\\my_file", "w+b");
fwrite(a, sizeof (int), 10, f);
```

Позиционирование в потоках Си

Функция *int fseek(FILE *f, long n, int org);* переносит указатель в потоке *f* на *n* байтов относительно начала отсчета *org*. Параметр *org* может принимать одно из трех значений, определенных в файле *stdio.h*:

SEEK_SET (=0). Началом отсчета является начало файла;

SEEK_CUR (=1). Началом отсчета является текущее положение указателя *f*;

SEEK_END (=2). Началом отсчета является конец файла.

Перенести указатель *f* в начало файла можно, переместив указатель *f* относительно начала на 0 байтов: *fseek(f, 0L, SEEK_SET);*.

Функция *fseek* возвращает 0 в случае успешного перемещения указателя и ненулевое значение в случае ошибки.

Функция *void rewind(FILE *f);* переносит указатель *f* в начало файла

Функция *long ftell(FILE *f);* возвращает текущую позицию указателя *f*.

Для определения размера файла в байтах перенесем указатель в конец файла и определим текущее положение указателя.

```
long n;
fseek(f, 0L, SEEK_SET);
n = ftell(f);
```

Если файл открыт с возможностью чтения и записи, то между этими действиями должно быть какое-то действие, связанное с переносом указателя или сбросом буфера.

Функция *int fflush(FILE*f);* сбрасывает буфер, связанный с потоком *f*. Если поток связан с файлом, открытым для записи, то вызов *fflush* приводит к принудительному переносу содержимого буфера в файл и очистке буфера. Если же поток связан с файлом, открытым для чтения, то входной буфер очищается. В обоих случаях файл остается открытым.

При удачном выполнении функция *fflush* возвращает 0, а в случае ошибки – ненулевое значение. Очистка всех буферов производится автоматически при нормальном завершении программы или при заполнении буферов. Закрытие файла также приводит к очистке буферов.

Функция *int fflushall*(); сбрасывает буферы всех открытых потоков и возвращает количество открытых потоков (входных и выходных). Эта функция не определена стандартом ANSI C.

Особенности позиционирования в текстовых файлах

При переносе указателя в текстовом файле началом отсчёта должно быть начало файла. Переносить указатель можно только в позицию, которая была получена с помощью функции *ftell*.

ПЕРЕНАПРАВЛЕНИЕ СТАНДАРТНОГО ВВОДА И ВЫВОДА

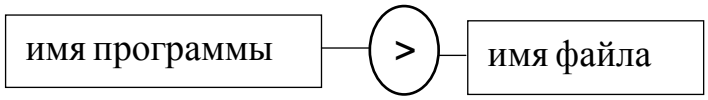
Исполняемые программы могут запускаться из командной строки. Командную строку операционной системы *Windows* можно запустить следующими способами:

Пуск ⟶ Программы (все программы) ⟶ Стандартные ⟶ Командная строка

Пуск ⟶ Выполнить ⟶ (ввести) *cmd*

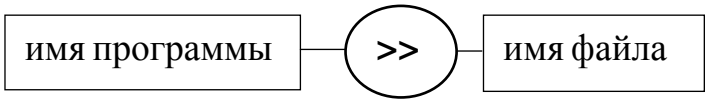
При этом с помощью переназначения устройств ввода/вывода можно выводить сообщения программ не на экран (в стандартный выходной поток *stdout*), а в файл или на принтер (перенаправление вывода) или читать входные данные не с клавиатуры (из стандартного входного потока *stdin*), а из заранее подготовленного файла (перенаправление ввода);

Из командной строки эти возможности реализуются следующим образом. Для того, чтобы перенаправить текстовые сообщения, предназначенные для вывода в текстовый файл, а на не экран, нужно использовать конструкцию.

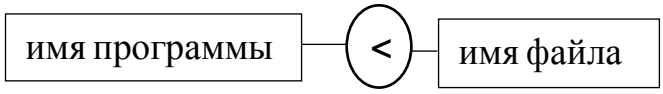


Если при этом заданный для вывода файл уже существовал, то он перезаписывается (старое содержимое теряется), если он не существует, то он создается.

При перенаправлении для дозаписи в конец существующего файла используется конструкция



Для чтения входных данных не с клавиатуры, а из существующего файла в командной строке должна быть задана конструкция



КОМАНДНАЯ СТРОКА АРГУМЕНТОВ

Программе можно передавать исходные данные не только с помощью функций ввода, но и во время ее запуска из командной строки. В этом случае заголовок функции *main* должен иметь вид

void main(*int argc*, *char* ***argv*)

Значением аргумента *argc* (***arg****ument* ***c****ount*) является количество аргументов, переданных программе. Значение *argc* всегда будет не меньше 1, так как пер-

вым аргументом является имя самой программы. Второй аргумент *argv* (***arg****ument value*) – массив строк, которые собственно и являются параметрами. *argv*[0] – это имя программы. После имени исполняемого файла в командной строке перечисляются через пробелы или символы табуляции параметры командной строки. В случае, если параметр представляет собой строку с пробелами, то его следует заключить в кавычки.

Если при запуске программы с аргументами функции *main* аргументы в командной строке отсутствуют, а в программе есть обращение к ним, появляется сообщение об ошибке.

Среды разработки C и C++-программ дают возможность аргументы командной строки задать в самой среде. Например, в Microsoft Visual Studio это делается так: щелкните правой кнопкой мыши на нужный проект в Обозревателе Решений, затем выберите «Свойства». Далее «Свойства конфигурации» и затем «Отладка». На правой панели будет строка «Аргументы команды». Вы сможете там ввести аргументы командной строки, и они будут автоматически переданы вашей программе при её запуске. В Code::Blocks для этого нужно будет выбрать «Project > Set program`s arguments».

Так как исходные данные функции могут быть любыми, а передаются программе они как строки, то уже в программе они должны приводиться к нужному типу.

Рассмотрим программу, которая выводит ее имя и сумму целых чисел, переданных ей в командной строке.

```
#include <stdio.h>
void main(int argc, char **argv)
{
  int i;
  long s;
  printf("%s\n", argv[0]); // Вывод имения программы
  for (i = 1, s=0; i < argc; i++)
    s+=atoi(argv[i]);
  printf("%s\n", s);
}
```

ПРЕПРОЦЕССОР ЯЗЫКА СИ

Как следует из названия, препроцессор обрабатывает программу до ее компиляции. Он позволяет использовать в программе символические имена, включать в текст программы содержимое других файлов, изменять условия компиляции и многое другое. Для этого существует набор директив препроцессора и предопределенных констант.

Препроцессор преобразует исходный текстовый файл в текстовый файл, предназначенный для обработки транслятором.

Препроцессор выполняет склеивание строк, которые в исходном тексте записаны с использованием знака переноса (\), при этом символ переноса ‘\’ удаляется. Комментарии заменяются пустыми строками, табуляции – пробелами, Символы перехода к новой строке заменяется пробелами.

Директивы препроцессора

Директива имеет вид:

#<имя директивы>[<параметры>]

Символ # должен быть первым в строке.

Рассмотрим некоторые директивы.

Директива include

Директива *include* позволяет включить в текст программы содержимое указанного в качестве параметра файла, имя файла может быть заключено в угловые скобки или в кавычки:

#*include* <имя файла> или **#*include* “имя файла”**.

В первом случае файл ищется в специальных стандартных каталогах, во втором – в первую очередь файл ищется в текущем каталоге, а если он там не найден, то он также ищется в специальных стандартных каталогах. Если подключаемый файл не найден, процесс компиляции завершается с ошибкой.

Директива define

#*define*<макрос>[<строка замещения>]

Макрос или макроопределение не содержит пробелов. Макросы бывают макросами-константами и макрофункциями. Строкой замещения является всё, что находится после макроса до конца строки. Если строка замещения не помещается на одной строке экрана, ее можно продолжить на другой строке с помощью знака переноса «обратный слэш» (\). Выполнение этой директивы называется макрорасширением.

Макрос-константа синтаксически – это идентификатор, семантически – это именованная константа, значение и тип которой определяется строкой замещения, которая является константным выражением. Традиционно имя макроса записывается прописными буквами. Препроцессор при макрорасширении заменяет макрос-константу строкой замещения, не выполняя при этом никаких вычислений и преобразований, символ перехода к новой строке заменяется пробелом. Макрорасширения не выполняются внутри символьных строк.

Например, исходная программа имеет вид:

```
#include  <stdio.h>
#define N 100
#define M  2*N
int main(void)
{
  int a[M][N];
  printf(“Описана матрица размером M*N a[%i][%i]\n”, M, N);
}
```

После выполнения макрорасширения программа будет иметь вид:

```
#include  <stdio.h>
int main(void)
{
  int a[2*100][100];
  printf(“Описана матрица размером M*N a[%i][%i]\n”, 2*100, 100);
```

}

Макрос действует до конца файла или до директивы **#*undef*<имя макроса>**, которая отменяет определение этого макроса. Далее этот макрос может быть переопределен.

#*define N* 100

int a[*N*]; // Описан целочисленный массив размером 100.

#*undef N*

#*define N* 30

float b[*N*]; // Описан вещественный массив размером 30.

Описание макрофункции имеет вид:

#*define*<имя функции>(<параметры>) <тело функции>

В заголовке макрофункции не должно быть пробелов, параметры являются идентификаторами и перечисляются через запятые. Строка замещения является телом макрофункции.

При макрорасширении формальные параметры заменяются фактическими в том же виде, в котором они записаны при вызове макрофункции. В описании тела макрофункции, предназначенной для вычисления значения, во избежание побочных эффектов каждый параметр и все тело заключаются в скобки.

Опишем макрофункцию для определения значения квадрата числа:

#*define SQR*(*x*) ((*x*)*(*x*))

Выражение 3**SQR*(*a*+2)/5 будет заменено при макрорасширении выражением 3*((*a*+2)* (*a*+2))/5. Если бы в теле макрофункции скобок не было, то после замены выражение имело бы вид: 3**a*+2**a*+2/5, что является ошибкой.

Если макрофункция аналогична процедуре языка Паскаль, то тело ее рекомендуется заключить в операторные скобки. Кроме того, в макросах можно использовать локальные переменные. Рассмотрим, например, макрофункцию, обменивающую значениями две переменные.

#*define SWAP*(*type*, *a*, *b*) {*type t*=*a*; *a*=*b*; *b*=*t*;}

Достоинства макрофункций.

Макрофункции

ускоряют время выполнения программы,

не требуют описаний типов,

типы могут быть параметрами.

Недостатки макрофункций.

Макрофункции

увеличивают размер программы,

требуют аккуратного описания во избежание побочных эффектов.

Стандартные библиотеки языка Си содержат большое количество макроконстант и макрофункций. Например,

#*define EOF* –1

#*define getchar*() *getc*(*stdio*)

#*define putchar*() *putc*(*stdout*)

В макроопределении можно использовать символ # перед параметром макрофункции в ее теле, при этом выражение #<параметр> представляет собой строку

#define PRINT_INT(*x*) *printf*("*#x*= %*i**n*", *x*)

Здесь #*x* рассматриваться как строка, а *x* – как выражение. При вызове *PRINT_INT*(1+2) будет выведено: 1+2=3.

С помощью макрофункций можно выполнять склеивание лексем, используя операцию ##.

#define INDEX(*x*, *y*) *x##y*

Оператор *INDEX*(*a*,1) = 139; при макрорасширении будет заменен оператором *a*1 = 139;.

Если строка замещения пустая, то происходит эффективное удаление из текста программы макроса.

Препроцессорные макросы

В стандарте Си предопределено несколько полезных макросов. Рассмотрим некоторые из них.

__LINE__ – номер текущей строки в исходном файле, в которой встречается этот макрос. Этот макрос полезен для сообщений об ошибках с указанием номера строки при отладке кода.

__DATE__ –текущая дата (дату компиляции) в формате ммм дд гггг (например, "*Dec* 31 2017"). Макрос *__DATE__* может быть использован для вывода информации о времени компиляции. Если дата компиляции не может быть получена, то будет выведена какая-то действительная дата, в зависимости от реализации.

__TIME__ – текущее время (время компиляции) в формате чч:мм:сс в 24-часовом формате (например, "22:29:12"). Макрос *__TIME__* дает информацию о времени в конкретный момент компиляции. Если время компиляции не может быть получено, то будет выведено какое-то действительное время, в зависимости от реализации.

__FILE__ – имя текущего файла. Макрос полезен, в том случае, если программа состоит из нескольких файлов.

__STDC__ – макрос определён, если программа была откомпилирована с использованием стандарта *ANSI C* со включенной проверкой на совместимость. В противном случае *__STDC__* не определен.

Директива #error

Директива #*erro*r предписывает компилятору в случае ее обнаружения остановить компиляцию. Она используется для отладки. Общий вид директивы следующий:

#error <сообщение_об_ошибке>

Сообщение об ошибке не заключается в кавычки. Когда компилятор обнаруживает директиву, он выводит сообщение в следующем виде: *Fatal:<имя файла><номер строки>:Error directive:<сообщение об ошибке>* .

Здесь <имя_файла> – имя файла, где была найдена директива #*error*, <номер_строки> – номер строки директивы, а <сообщение_об_ошибке> – это собственно само сообщение.

Условная компиляция

Существует ряд директив, которые указывают компилятору, какие фрагменты текста программы должны компилироваться, а какие должны быть исключены из текста программы. Эти директивы называются директивами условной компиляции.

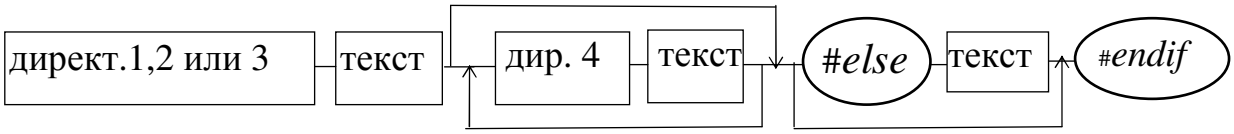
Выполнение директив условной компиляции аналогично выполнению условного оператора в программе. Разница заключается в том, что результатом условной компиляции является текст программы, подготовленный для компиляции, а условные операторы выполняются во время работы программы.

Для условной компиляции используются следующие директивы.

1. #*if*<константное выражение>//если конст. выражение истинно
2. #*ifdef*<макрос> //если макрос определен
3. #*ifndef*<макрос>//если макрос не определен
4. #*elif*<конст. выражение>//иначе, если конст. выражение истинно
5. #*else*
6. #*endif*

В константных выражениях может использоваться операция *defined*. Форма записи этой операции: *defined*(<макрос>). Результатом этой операции является 1, если макрос был определен ранее с помощью директивы #*define* и 0 в противном случае. Нетрудно заметить, что директивы #*ifdef A* и #*if defined*(*A*) выполняются одинаково. Но операцией *defined* удобнее пользоваться в сложном выражении, входящем в состав директивы #*if* или #*elif* как например в следующем случае: #*if defined*(*A*) || *defined*(*B*) || *defined*(*C*)

Синтаксически фрагмент программы с условной компиляцией имеет вид:



За любой из директив #*if, #ifdef, ifndef*, #*else, #elif* может следовать произвольное число строк текста программы, которые будут включены в программу для компиляции, или исключены из нее в зависимости от выполнения или не выполнения условий.

Приведенная выше конструкция должна находиться в одном файле.

Директиву #*ifdef* удобно применять при отладке программ для включения в программу отладочных секций. Например,

#*ifdef DEBUG*

<Отладочная секция>

#endif

Отладочных секций в программе может быть несколько. После того, как программа будет отлажена, директиву *#define DEBUG* можно отключить с помощью директивы #*undef DEBUG* или закомментировать.

Директива #*ifndef* часто применяется для того, чтобы не происходило повторного включения файлов, текст которых вставляется в программу директивой #*include*. Такая ситуация может иметь место, когда программа состоит из нескольких файлов, в каждом из которых может быть подключен один и тот же файл. Чтобы этого не произошло, в каждый файл необходимо включить специальные средства защиты от повторного включения. Такими средствами защиты снабжены все заголовочные файлы стандартной библиотеки.

Пример.

#*ifndef FILE_NAME* // если макрос *FILE_NAME* не определен
#*include* "*filename.h*" // включение текста файла *filename.h*
#*define FILE_NAME* // определение *FILE_NAME*
#*endif*

Здесь *FILE_NAME* – зарезервированный для файла с именем *filename.h* макрос. Его нежелательно использовать по другому назначению.

Условная компиляция также используется для настройки программы под платформу компилятора.

КОМПИЛЯЦИЯ И СБОРКА ПРОГРАММЫ

Файл – единица хранения информации в файловой системе. Язык Си дает возможность разбивать программу на части, каждая из которых хранится в отдельном файле. Хранение программы в разных файлах может подчеркнуть логическую структуру программы, что облегчает понимание ее другими пользователями. Каждый файл может быть откомпилирован независимо от других файлов. Файл проекта, объединяет отдельные части программы в единое целое. Даже если программа небольшая, она не создается из одного файла, так как она обращается к функциям стандартных библиотек, которые не предоставляются в виде исходных текстов для их включения в программу.

Раздельная компиляция

В результате обработки файла препроцессором получается текстовый файл, который называется *единицей компиляции*. После успешной обработки препроцессором файл может быть откомпилирован. Если большая программа хранится в одном файле, то после внесения в нее изменений вся программа должна перекомпилироваться. Разбиение программы на несколько файлов позволяет сократить время на перекомпиляцию.

Фазы компиляции

Компиляция начинается с *лексического анализа*: выделяются лексемы (идентификаторов, констант, служебных слов и т.п.) и проверяется правильность их написания.

Далее следует *синтаксический анализ* – проверка правильности построения конструкций языка (выражений, операторов). Синтаксический анализ выполняется рекурсивно.

Третьим этапом является *семантический анализ* – проверка смысловой правильности конструкций (инициализации переменных, совместимости типов операндов в операциях и т. п.).

Если в результате анализа ошибки не обнаружены, то происходит генерация кода, то есть преобразование описанных в этом файле программы данных и команд в машинное представление всюду, где возможно. В процессе генерации кода происходит его оптимизация. В результате получается объектный модуль – файл, имя которого совпадает с именем исходного файла, а расширение зависит от системы программирования, например, *.*obj*, *.o. Объектный код содержит двоичное представление готовых к выполнению кодов команд, адреса и содержимое памяти данных для внутренних объектов. Внешние объекты остаются в объектном модуле в текстовом представлении.

Если в файле есть внешний объект, для него создается внешняя ссылка, и он остается в текстовом виде. Для того, чтобы компиляция была возможной, программист должен включить в программу объявления внешних объектов.

Для объекта, который может быть использован в других файлах, создаётся точка входа – адрес во внутреннем представлении и имя в текстовом виде.

Компоновка

Отдельно откомпилированные части связывает в единое целое программа-компоновщик (*linker*).

Компоновка Си-программы должна быть полностью завершена до ее запуска. В C++ новый код может быть добавлен к программе (динамически скомпилирован) уже после загрузки.

После того, как были получены все объектные коды частей программы, происходит сборка модулей (линовка, компоновка): объектные модули размещаются один за другим в загрузочном модуле. Каждый объектный модуль получает начальный адрес. Если объектный модуль имеет точку входа, то ее адрес определяется абсолютным адресом модуля и собственным адресом точки входа. Для каждой внешней ссылки ищется точка входа с таким же именем. Если она найдена, то ее адрес заменяет внешнюю ссылку. Если остаются неразрешенные внешние ссылки, то компоновщик просматривает подключенные библиотеки.

Библиотека представляет собой набор объектных модулей с общим каталогом в начале. Если в одном из объектных модулей библиотеки найдена необходимая точка входа, то весь этот объектный модуль добавляется в загрузочный модуль программы. Затем повторяются действия по связыванию внешних ссылок и точек входа модуля.

Объектные модули библиотек могут, в свою очередь содержать внешние ссылки к объектным модулям, как своей библиотеки, так и к объектным модулям других библиотек. Поэтому процесс компоновки является итерационным.

Если в программе используются внешние объекты, то имена всех этих объектов должны быть согласованы во всех файлах программы. Это достигается использованием заголовочных файлов.

Негласное требование: *h*-файл может содержать описания типов, переменных, именованных констант, объявления функций, директивы включения, макроопределения, директивы условной компиляции и комментарии. *h*-файл не должен включать определений функций и объявлений переменных.

Для создания программы их нескольких файлов должен быть создан *файл проекта*, в который включаются имена исходных файлов и/или объектных модулей. Системы программирования предоставляют возможность создать новый или открыть существующий проект, включить или исключить файлы, установить параметры компиляции.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Подбельский В. В.* Курс программирования на языке Си: учебник / В. В. Подбельский, С. С. Фомин.– М.: ДМК-Пресс, 2018. – 384 с.
2. *Керниган Б. В.* Язык программирования С [Электронный ресурс] / Б. В. Керниган, Д. М. Ричи. – 2-е изд. – Электрон. текстовые данные. – М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. – 313 с. – 2227-8397. – Режим доступа: http://www.iprbookshop.ru/73736.html
3. *Костюкова Н. И.* Программирование на языке Си [Электронный ресурс]: методические рекомендации и задачи по программированию / сост. Н. И. Костюкова. – Электрон. текстовые данные. – Новосибирск: Сибирское университетское издательство, 2017. – 160 с. – 978-5-379-02016-3. – Режим доступа: http://www.iprbookshop.ru/65289.html
4. *Перри Г.* Программирование на С для начинающих: библиотека программиста /Г. Перри, Д. Миллер.– М.: Эксмо, 2015. – 368 с.
5. *Васильев А. Н.* Программирование на С в примерах и задачах: учебник: /А. Н Васильев.– М.: Эксмо-Пресс, 2017. – 560 с.

Учебное издание

Брусенцева Валентина Станиславовна

Язык программирования Си

Учебное пособие

Подписано в печать 12.10.18. Формат 60х84/16. Усл. печ. л. 3,8. Уч.-изд. л. 4,0.
Тираж 95 экз. Заказ Цена
Отпечатано в Белгородском государственном технологическом университете им. В.Г. Шухова
308012, г. Белгород, ул. Костюкова, 46