[7. Символьные строки.](#_Toc515770000)

[7.1. Массивы символов в C++](#_Toc515770001)

[7.2. Функции работы со строками символов](#_Toc515770002)

[7.2.1. Определение длины строк](#_Toc515770003)

[7.2.2. Копирование и конкатенация строк](#_Toc515770004)

[7.2.3. Сравнение строк](#_Toc515770005)

[7.2.4. Преобразование строк](#_Toc515770006)

[7.2.5. Обращение строк](#_Toc515770007)

[7.2.6. Поиск символов](#_Toc515770008)

[7.2.7. Поиск подстрок](#_Toc515770009)

[7.3. Функции преобразования типа](#_Toc515770010)

[7.4. Функции printf() и scanf().](#_Toc515770011)

[7.4.1. Использование функции printf().](#_Toc515770012)

[7.4.2. Модификаторы спецификации преобразования, используемые в функции printf().](#_Toc515770013)

[7.4.3. Использование функции printf() для преобразования данных.](#_Toc515770014)

[7.4.4. Применение функции scanf().](#_Toc515770015)

1. Символьные строки.

## Массивы символов в C++

В стандарт C++ включена поддержка нескольких наборов символов. Традиционный 8-битовый набор символов называется "узкими" символами. Кроме того, включена поддержка 16-битовых символов, которые называются "широкими". Для каждого из этих наборов символов в библиотеке имеется своя совокупность функций.

Как и в ANSI. С, для представления символьных строк в C++ не существует специального строкового типа. Вместо этого строки в C++ представляются как массивы элементов типа char, заканчивающиеся терминатором строки - символом с нулевым значением ('\0'). Строки, заканчивающиеся нуль-терминатором, часто называют ASCIIZ-строками. Символьные строки состоят из набора символьных констант, заключенного в двойные кавычки:

"Это строка символов..."

В таблице приведен набор констант, применяющихся в C++ в качестве символов.

|  |  |
| --- | --- |
| прописная буква | от 'А1 до 'Z',oт 'А1 до 'Я' |
| строчная буква | от ' а' до ' z ', от ' а' до ' я' |
| цифра | от '0 ' до ' 9' |
| пустое место | горизонтальная табуляция '\9', перевод строки (код ASCII 10), вертикальная табуляция (код ASCII 11), перевод формы код ASCII 12), возврат каретки код ASCII 13) |
| символы пунктуации | ! ” # $ ’ () \* +,-./:;<=>?&% [\]^\_{| }~ |
| управляющий символ | все символы с кодами от 0 до 1 Fh включительно и символ с кодом 7Fh |
| пробел | символ пробела (код ASCII 32) |
| шестнадцатеричная цифра | от '0' до '9', от 'А' до 'F', от ' а ' до ‘ f ’ |

При объявлении строкового массива необходимо принимать во внимание наличие терминатора в конце строки:

// Объявление строки размером 10 символов,

// включая терминатор. Реальный размер

// строки: 9 символов + 1 терминатор

char buffer[10];

Строковый массив может при объявлении инициализироваться начальным значением. При этом компилятор автоматически вычисляет размер будущей строки и добавляет в конец нуль-терминатор:

// Объявление и инициализация строки:

char Wednesday[] = "Среда";

// что равносильно:

char Wednesday[] ={'С''р','е','д','а','\0'};

В качестве оператора ввода при работе со строками вместо оператора записи в поток >> лучше использовать функцию getline (), так как потоковый оператор ввода игнорирует вводимые пробелы, а кроме того, может продолжить ввод элементов за пределами массива, если под строку отводится меньше места, чем вводится символов. Функция getline () принимает два параметра: первый аргумент указывает на строку, в которую осуществляется ввод, а второй - число символов, подлежащих вводу.

Рассмотрим пример объявления символьных строк и использования функции ввода getline ().

#include <iostream>

using namespace std;

int main ()

{

setlocale(LC\_ALL, "Rus");

char myString[4];

cout << "Введите строку: " << '\n';

cin.getline(myString, 4);

cout << "Вы ввели: " << myString;

getchar(); getchar();

return 0;

}



Объявленная в начале программы строка myString может принять только три значащих символа и будет завершена нуль-терминатором. Все последующие символы, вводимые в этот строковый массив, будут отброшены. Как видно из примера, при использовании функции getline () вторым параметром следует указывать число, меньше или равное размеру вводимой символьной строки.

Большинство функций работы со строками содержится в библиотеке string.h. Основные функции работы с символьными массивами сведены в таблицу.

«Символьная строка» — это последовательность, состоящая из одного или более символов. В качестве примера рассмотрим следующую строку:

"Строки изливались прямо из сердца!"

Кавычки не являются частью строки. Они вводятся только для того, чтобы отметить ее начало и конец, т. е. играют ту же роль, что и апострофы в случае одиночного символа.

В языке Си нет специального типа, который можно было бы использовать для описания строк. Вместо этого строки представляются в виде «массива» элементов типа char. Это означает что символы в строке можно представить себе расположенными в соседних ячейках памяти - по одному символу в ячейке (Смотри рисунок).

_05_01

_05_02

Необходимо отметить, что на рисунке последним элементом массива является символ \0. Это «нуль-символ», и в языке Си он используется для того, чтобы отмечать конец строки. Нуль-символ — не цифра 0; он не выводится на печать и в таблице кода ASCII (American Standard Code for Information Interchange — Американский стандартный код для обмена информацией) имеет номер 0. Наличие нуль-символа означает, что количество ячеек массива должно быть по крайней мере на одну больше, чем число символов, которые необходимо размещать в памяти.

Ну, а теперь спросим, что такое массив? Массив можно представить себе как совокупность нескольких ячеек памяти, объединенных в одну строку. Если вы предпочитаете более формальные и строгие определения, то массив — это упорядоченная последовательность элементов данных одного типа. В нашем примере мы создали массив из 40 ячеек памяти, в каждую из которых можно поместить один элемент типа char. Мы осуществили это с помощью оператора описания:

char name [40];

Квадратные скобки указывают, что переменная name — массив, 40 — число его элементов, a char задает тип каждого элемента.

(В комментариях к программе было отмечено, что при желании вы можете воспользоваться более сложным оператором описания):

static char name [40];

Ввиду некоторой специфики, связанной с реализацией функции scanf() в нашей системе, мы вынуждены использовать эту вторую форму, но весьма вероятно, что вы сможете выбрать любую из них. Если обнаружится, что при работе с первой формой оператора описания у вас возникнут трудности при решении наших примеров, попробуйте воспользоваться второй. В действительности вторая форма должна работать в любой системе, но мы не хотим применять тип Static до тех пор, пока не рассмотрим понятие классов памяти.)

На первый взгляд все это выглядит довольно сложным: вы должны создать массив, расположить символы в виде строки и не забыть добавить в конце \0. К счастью, о большинстве деталей компилятор может «позаботиться» сам.

Попробуйте выполнить приведенную ниже программу, чтобы посмотреть, как просто все происходит на практике:

/\* похвала 1\*/

#define PRAISE " Вот эта да, какое великолепное имя!"

main()

{

char name [50];

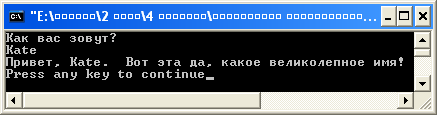
printf("Как вас зовут?\n" );

scanf(" %s", name);

printf("Привет, %s. %s\n" , name, PRAISE);

}

Символ %s служит указанием функции printf() напечатать строку. Результат выполнения программы похвала 1 может выглядеть, например, так:



Как видите, нам не пришлось самим помещать нуль-символ в конец массива. Эта задача была выполнена за нас функцией scanf() при чтении вводимой строки. PRAISE — «символическая строковая константа». Ниже мы рассмотрим директиву #define более подробно, а пока вы должны знать, что кавычки, в которые заключена фраза, следующая за строковой константой PRAISE, идентифицируют эту фразу как строку, и поэтому в ее конец будет помещен нуль-символ.

Заметим (и это очень важно), что функция scanf() при вводе строки «Элмо Бланк» читает только имя Элмо. Дело в том, что, встретив какой-нибудь разделитель (пробел, символ табуляции или перевода строки), функция scanf() прекращает ввод символов, т. е. в данном случае она прекращает опрос переменной name в тот момент, когда доходит до пробела между «Элмо» и «Бланк». Вообще говоря, функция scanf() вводит только одиночные слова, а не целую фразу в качестве строки. Для чтения входной информации в языке Си имеются другие функции, например функция gets(), предназначенная для обработки строк общего вида. Более полно работу со строками мы рассмотрим позднее.

Необходимо заметить также, что строка "х" не то же самое, что символ 'х'. Первое различие: 'x' — объект одного из основных типов (char), в то время как "х" — объект производного типа (массива элементов типа char). Второе различие: "х" на самом деле состоит из двух символов — символа 'х' и нуль-символа.

_05_03

## Функции работы со строками символов

|  |  |
| --- | --- |
| **Наименование** | **Краткое описание** |
| strcpy | копирует строку2 в строку 1 |
| streat | присоединяет строку2 в конец строки 1 |
| strchr | возвращает позицию первого вхождения символа в строку |
| stremp | сравнивает строку1 со строкой2, различая прописные и строчные буквы |
| strempi | см. striemp |
| strcpy | копирует строку2 в строку1 |
| strcspn | возвращает позицию первого вхождения символа из заданного набора символов |
| strdup | распределяет память и делает копию строки |
| strerror | возвращает по заданному номеру системной ошибки указатель на строку текста сообщения об ошибке |
| \_strerror | возвращает указатель на строю, образованию объединением произвольной строки и сообщения об ошибке в библиотечной функции |
| stricmp | сравнивает строку 1 со строкой 2, не различая прописные и строчные буквы |
| strlen | возвращает длину строки в байтах, не учитывая нулевой терминатор |
| strlwr | преобразует все символы строки в строчные буквы |
| strncat | присоединяет заданное число символов строки 2 в конец строки 1 |
| strncmp | сравнивает заданное число символов двух строк, различая прописные и строчные буквы |
| strncmpi | см. strnicmp |
| strncpy | копирует заданное число символов строки2 в строку 1 |
| strnicmp | сравнивает заданное число символов двух строк, не различая прописные и строчные буквы |
| strnset | помещает заданный символ в заданное число позиций строки |
| strpbrk | отыскивает место первого вхождения любого символа из заданного набора |
| strrсhr | отыскивает последнее вхождение символа в строке |
| strrev | реверс строки |
| strset | помешает символ во все позиции строки |
| strspn | возвращает позицию в строке первого символа, который не принадлежит заданному набору символов |
| stratr | отыскивает место первого вхождения строки 2 в строку1 |
| strtok | возвращает указатель на лексему, ограниченную заданным разделителем |
| strpr | преобразует все буквы строки в прописные буквы |
| isalnum(с) | истина, если символ с является буквой цифрой |
| isalpha(с) | истина, если символ с является буквой |
| isascii(с) | истина, если код символа с <= 127 |
| iscntrl(с) | истина, если с - управляющий символ |
| isdigit(с) | истина, если с - символ десятичной цифра |
| isgraph(c) | истина, если с - печатаемый символ (код от 33 до 126) |
| islower(с) | истина если с - строчная буква |
| isprint(с) | истина, если о - печатаемый символ (код от 33 до 126) или пробел |
| ispunct(с) | истина, если с - символ пунктуации |
| isspace(с) | истина, если с - символ пустого места или пробела |
| isupper(c) | истина, если с - прописная буква |
| isxdigit(с) | истина, если с - символ шестнадцатеричной цифры |
| toascii(с) | возвращает код о или 128. если код с больше 127 |
| \_tolower(с) | преобразует символ прописной буквы в символ строчной. Используется, если точно известно, что с - прописная буква. Возвращает код строчной буквы |
| tolower(с) | преобразует символ прописной буквы в символ строчной, не изменяя все остальные символы. В отличие 0T\_tolower () сначала проверяет, является ли с прописной буквой. Не являющиеся прописными буквами символы не преобразуются. Возвращает код строчной буквы |
| \_toupper | преобразует символ строчной буквы в символ прописной. Используется, если точно известно, что с - строчная буква. Возвращает код прописной буквы |
| toupper | преобразует символ строчной буквы в символ прописной, не изменяя все остальные символы. В отличие от \_toupper () сначала проверяет, является ли с строчной буквой. Не являющиеся строчными буквами символы не преобразуются. Возвращает код прописной буквы. |

В следующих пунктах данной главы самые популярные го приведенных функций будут рассмотрены подробно, однако следует учесть, что в некоторых версиях поставки библиотек C++ данные функции могут осуществлять неправильно (или не выполнять вовсе) работу с национальными символами.

### Определение длины строк

Очень часто при работе со строками необходимо знать, сколько символов содержит строка. Для выяснения информации о длине строки в заголовочном файле string.h описана функция strlen (). Синтаксис этой функции имеет вид:

size\_t strlen(const char\* string)

Данная функция в качестве единственного параметра принимает указатель на начало строки string, вычисляет количество символов строки и возвращает полученное беззнаковое целое число (size\_t). Функция sizeof () возвращает значение на единицу меньше, чем отводится под массив по причине резервирования места для символа ‘\0’. Следующий фрагмент демонстрирует использование функции strlen ():

char Str[] = "ABCDEFGHIJK";

unsigned int i;

i = strlen(Str);

Часто функция sizeof () используется при вводе строк в качестве второго параметра конструкции cin.getline (), что делает код более универсальным, так как не требуется явного указания числа вводимых символов. Если теперь потребуется изменить размер символьного массива, достаточно модифицировать лишь одно число при объявлении строки символов:

// Было:

// char myString[4];

// cin. getline(myString, 4);

// Стало:

char myString[20];

cin.getline(myString, sizeof(myString));

Программы на языке C++, по сути, являются переносимыми. Переносимость означает, что программы, написанные для работы на компьютерах одного типа, будут (хотя и не обязательно!) работать и на компьютерах другого типа. Эго свойство языка С++ сделало его с самого начала популярным, когда операционная система UNIX, реализованная на C++ для машин PDP, была перенесена с небольшими изменениями на Interdata 8/32. Но при переносе программного обеспечения с ЭВМ одною типа на другой разработчика подстерегают определенные неприятности.

Переменные разных типов данных занимают в памяти компьютера неодинаковое количество байтов. При переменные одного и того же типа могут иметь разный размер ещё и в зависимости от того, на каком компьютере и в какой операционной системе они используются.

Так, чистая целочисленная переменная (тип int без спецификаторов short или long) при работе в 16-разрядной операционной системе (DOS, Windows 4.1 и т.д.) занимает всего 2 байта. В то же время под эту же переменную в 32-разрядной операционной системе (Windows 95. Windows NT и т.д.) отводится уже 4 байта.

Определить размер переменной любого типа данных (как базового, так и производного) можно с помощью оператора размера sizeof. Данный оператор может быть применен к константе типу и переменной. Раньше мы практически без объяснений использовали операцию sizeof, которая дает нам размер объектов в байтах.

Функция strlen() позволяет определять длину строки числом символов. Поскольку для размещения одного символа в памяти отводится 1 байт, можно было бы предположить, что в результате применения любой из этих двух операций к одной строке будет получен одинаковый результат. Оказывается, это не так. Давайте немного изменим нашу предыдущую программу (добавим к ней несколько строк), и тогда мы поймем, в чем дело.

/\* похвала 2\*/

#define PRAISE " Вот это да, какое великолепное имя!"

main()

{

char name [50];

printf("Как вас зовут?\n" );

scanf(" %s" , name);

printf("Привет, %s. %s\n" , name, PRAISE);

printf("Ваше имя состоит из %d букв и занимает %d ячеек памяти. \n",

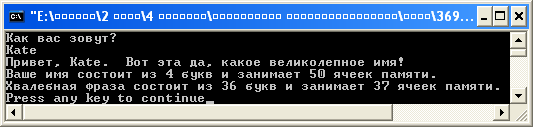
strlen (name), sizeof name);

printf(" Хвалебная фраза состоит из %d букв" , strlen (PRAISE));

printf(" и занимает %d ячеек памяти. \n" , sizeof PRAISE);

}

Заметим, что случайно мы воспользовались двумя методами для обработки длинных операторов printf(). В первом случае мы записали один оператор печати в двух строках программы (строкой программы считается строка до запятой).Мы сделали это, поскольку разрешается разбивать строку между аргументами, но не посередине строки. В другом случае использовались два оператора printf() для печати одной строки; мы указали символ «новая строка» (\ n) только во втором из них. Представленный ниже результат работы данной программы поможет понять подобную ситуацию:



Давайте посмотрим, в чем дело. Массив name занимает 50 ячеек памяти, и именно об этом сообщает операция sizeof. Но для хранения имени Перки требуются только первые пять ячеек, и как раз об этом нас информирует функция strlen(). В шестой ячейке массива name содержится нуль-символ, и его появление служит сигналом для функции strlen() прекратить подсчет символов.

_05_04

При переходе к обработке константы PRAISE обнаруживается, что функция strlen() опять дает нам точное число символов (включая пробелы и знаки пунктуации) в строке. Результат операции sizeof оказывается на единицу большим, поскольку при этом учитывается и «невидимый» нуль-символ, помещенный в конец строки. Мы не указываем компилятору, какой объем памяти он должен отвести для размещения всей фразы; он сам подсчитывает число символов между кавычками.

Еще одно замечание: мы использовали операцию sizeof со скобками, а здесь — без них. Решение, использовать ли скобки или нет, зависит от того, что вы хотите знать: объем памяти, отводимый под элементы конкретного типа, или объем памяти, занимаемый определенным объектом. В первом случае вы писали бы sizeof(char) или sizeof(float), а во втором — sizeof name или sizeof 6.28.

В данном разделе операции strlen() и sizeof использовались только для удовлетворения нашего любопытства; в действительности они представляют собой важные программные средства. Функция strlen(), например, полезна в любого сорта программах обработки строк или символов, в чем вы сможете убедиться позднее.

Здесь мы продолжим работу с данными: покопаемся в вопросах, выходящих за пределы тех, которые были связаны с типами данных, и рассмотрим символьную строку. Сначала опишем важное средство языка — препроцессор Си — и узнаем, как задавать и использовать символические константы. Затем вновь обсудим способы ввода и вывода данных, при этом более полно исследуем возможности функций printf() и scanf(). Ну, а теперь вы, вероятно, ожидаете примера программы, который должен быть помещен в начале; мы не будем вас разочаровывать и приведем его.

/\* непринужденный разговор \*/

#define DENSITY 62.4 /\* плотность тела человека в фунтах на кубический фут \*/

main() /\* любопытствующая программа\*/

{

float weight, volume;

int size, letters;

char name[40]; /\* или попробуйте "static char name [40];" \*/

printf("Привет! Как вас зовут?\n" );

scanf("%s" , name);

printf("%s, Каков ваш вес в фунтах?\n", name);

scanf("%f", &weight);

size = sizeof name;

letters = strlen (name);

volume = weight/DENSITY;

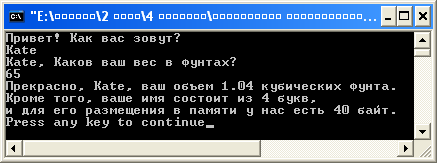
printf("Прекрасно, %s, ваш объем %2.2f кубических фута.\n", name, volume);

printf("Кроме того, ваше имя состоит из %d букв,\n", letters);

printf("и для его размещения в памяти у нас есть %d байт,\n", size);

}

Результат работы программы «непринужденный разговор» может, например, выглядеть следующим образом:



Перечислим основные новые черты этой программы.

1. Мы использовали «массив» для хранения «символьной строки» — в данном случае для некоторого имени.
2. При вводе и выводе строки была использована «спецификация преобразования» %s.
3. Для определения символической константы DENSITY был использован препроцессор языка Си.
4. Для нахождения длины строки была использована функция strlen().

Способ ввода-вывода, реализованный в языке Си, может показаться вначале несколько более сложным по сравнению с вводом-выводом, предусмотренным, например, в Бейсике. Однако эта сложность окупается улучшенными возможностями управления вводом-выводом и большей эффективностью получаемых программ. Указанная трудность — не единственная, с которой вы столкнетесь в дальнейшем.

### Копирование и конкатенация строк

Значения строк могут копироваться из одной в другую. Для этой цели используют ряд стандартных функций, описываемых ниже.

**Функция strcpy**() имеет прототип:

char\* strcpy(char\* stl, const char\* str2)

и выполняет побайтное копирование символов из строки, на которую указывает str2, в строку по указателю stl. Копирование прекращается только в случае достижения нуль-терминатора строки str2, поэтому перед копированием необходимо удостовериться, что длина str2 меньше или равна длине str1. В противном случае возможно возникновение ошибок, связанных с наложением данных.

Например, следующий фрагмент копирует в строку Str значение строки "Проверка копирования":

char Str[20];

strcpy(Str, "Проверка копирования");

Можно производить копирование не всей строки, а лишь отдельного ее фрагмента (до конца строки). При этом второй параметр является указателем на некоторый элемент строкового массива. Например, следующий фрагмент скопирует в str2 окончание строки str1:

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str1[25] = "Проверка копирования";

char str2[25];

char\* ptr = str1;

ptr += 9;

// ptr теперь указывает на

// подстроку "копирования"

// в строке str1

strcpy(str2, ptr);

cout << str2 << '\n';

getchar(); getchar();

return 0;

}



**Функция strncpy()** отличается от strcpy () тем, что в ее параметрах добавляется еще один аргумент, указывающий количество символов, не больше которого будет скопировано. Ее синтаксис имеет вид:

Char\* strncpy(char\* str1, const char\* str2, size\_t num)

Если длина str1 меньше длины str2, происходит урезание символов:

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char cLong[] = "012345678901234567890123456789";

char cShort[]= "abcdef";

strncpy(cShort, cLong, 4);

cout << cShort << '\n';

cout << cLong << '\n';

getchar(); getchar();

return 0;

}



В результате будет выведено:

0123ef

012345678901234567890123456789

То есть из строки cLong в строку cShort скопировано четыре первых символа, затерев тем самым исходное значение начала короткой строки.

**Функция strdup()** в качестве параметра получает указатель на строку-источник, осуществляет распределение памяти, копирует в отведенную область строку и возвращает указатель на начало полученной строки-копии. Синтаксис функции следующий:

char\* strdup(const char\* source)

В следующем примере производится копирование строки str1 в строку str2:

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char\* str1 = "Процедура не найдена";

char\* str2;

str2 = strdup(str1);

cout << "str1 = " << str1 << '\n';

cout << "str2 = " << str2 << '\n';

getchar(); getchar();

return 0;

}



Конкатенация (или присоединение) строк довольно часто используется для образования новой строки символов. Для этой операции стандартная библиотека предлагает функции strncat() и strncat().

**Функция strncat()** осуществляет конкатенацию строк, и присоединяет указанное в третьем параметре количество символов (беззнаковое целое):

char\* strncat(char\* str1, const char\* str2, size\_t num)

Функция возвращает указатель на начало сформированной строки str1. Следующий пример производит конкатенацию строки str1 с семью первыми символами подстроки str2:

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str[81];

strcpy(str, "Для продолжения ");

cout << "str = " << str << '\n';

strncat(str, "нажмите клавишу", 7);

cout << "str = " << str << '\n';

getchar(); getchar();

return 0;

}



В результате будет выведена строка:

"Для продолжения нажмите".

### Сравнение строк

Библиотека функций string.h предлагает к использованию готовые функции, выполняющие сравнение строк. Ниже приводятся функции, выполняющие посимвольное сравнение строк.

**Функция stricmp()** имеет синтаксис:

Int stricmp (const char\* str1, const char\* str2)

После сравнения строк str1 и str2 данная функция возвращает в результате одно из следующих значений:

* <0 - если строка str1 меньше, чем str2;
* =0 - если строки эквивалентны;
* >0 - если строка str1 больше, чем str2.

Эта функция производит сравнение, различая прописные и строчные буквы. Следующий пример иллюстрирует работу функции stremp ():

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str1[]="Ошибка открытия базы";

char str2[]="0шибка открытия Базы";

int i;

i = stricmp(str1, str2);

cout << "str1 = " << str1 << '\n';

cout << "str2 = " << str2 << '\n';

cout << "i = " << i << '\n';

getchar(); getchar();

return 0;

}

В результате переменной i будет присвоено положительное значение, так как строка из str1 меньше, чем строка из str2, по той причине, что прописные буквы имеют код символов меньше, чем те же символы в нижнем регистре (слово "базы " в первом случае начинается со строчной литеры, а во втором - с прописной).



Следующий фрагмент программы демонстрирует применение функции stricmp():

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str1[]="Moon";

char str2[]="MOON";

int i;

i = stricmp(str1, str2);

cout << "str1 = " << str1 << '\n';

cout << "str2 = " << str2 << '\n';

cout << "i = " << i << '\n';

getchar(); getchar();

return 0;

}

В данном случае переменной i будет присвоено значение 0 (сигнализируя тем самым совпадение строк), так как str1 и str2 отличаются только регистром.



**Функция strncmp()** проводит сравнение определенного числа первых символов двух строк. Регистр символов при этом учитывается. Функция имеет следующий прототип:

int strnсmp(const char\* str1, const char\* str2, size\_t\_num)

Данная функция сравнивает num первых символов двух строк, на которые указывают str1 и str2, и возвращает одно из следующих значений:

* <0-если строка str1 меньше, чем str2;
* =0 - если строки эквивалентны;
* >0 - если строка str1 больше, чем str2.

Рассмотрим пример использования функции strncmp().

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str1[]="Ошибка открытия базы";

char str2[]="0шибка Открытия базы";

int i;

i = strncmp(str1, str2, 12);

cout << "str1 = " << str1 << '\n';

cout << "str2 = " << str2 << '\n';

cout << "i = " << i << '\n';

getchar(); getchar();

return 0;

}



В результате сравнения первых 12-ти символов обеих строк переменная i получит положительное значение, так как подстроки "Ошибка открытия базы" и "Ошибка открытия базы " отличаются одним символом и в первом случае код символа больше, чем во втором.

В следующем примере производится сравнение заданного числа символов подстрок:

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str1[]="Opening error";

char str2[]="Opening error...";

int i;

i = strncmp(str1, str2, 13);

cout << "str1 = " << str1 << '\n';

cout << "str2 = " << str2 << '\n';

cout << "i = " << i << '\n';

getchar(); getchar();

return 0;

}



В результате переменной i будет присвоено значение 0, так как первые 13 символов обеих подстрок отличаются только регистром.

### Преобразование строк

Элементы символьных строк могут быть преобразованы из одного регистра в другой. Для этого используются стандартные функции strlwr() и strupr(). Следует отметить, что в некоторых версиях компиляторов имена данных функций могут следовать без ведущего символа подчеркивания.

**Функция strlwr()** принимает в качестве параметра указатель на строку символов, преобразует эту строку к нижнему регистру (строчные символы) и возвращает указатель на полученную строку. Данная функция имеет следующий прототип:

char\* strlwr(char\* str)

Следующий фрагмент показывает применение функции strlwr():

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str1[] = "ABRACADABRA";

strlwr(str1);

cout << "str1 = " << str1 << '\n';

getchar(); getchar();

return 0;

}



После вызова функции строка str будет содержать значение "abracadabra".

**Функция \_strupr()** объявлена следующим образом:

char\* \_strupr(char\* str)

Данная функция преобразует строку символов, на которую указывает str в прописные буквы (к верхнему регистру). В результате работы функции возвращается указатель на полученную строку.

Следующий пример переводит символы заданной строки в верхний регистр:

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str1[]="pacific ocean";

\_strupr(str1);

cout << "str1 = " << str1 << '\n';

getchar(); getchar();

return 0;

}



В результате будет выведено:

PACIFIC OCEAN

Приведенные выше функции преобразования строк, работая с указателями, преобразуют исходную строку, которая не всегда может быть восстановлена, поэтому, если в дальнейшем коде программы потребуется воспользоваться оригиналом символьной строки, перед использованием функций strlwr() и strupr () необходимо сделать копию их аргументов.

На практике довольно широко используются функции проверки принадлежности символов какому-либо диапазону, такие как isalnum(), isalpha(). isascii(), isdigit() и т.д., объявленные в заголовочном файле ctype.h. Ниже рассматривается пример использования этого вида функций.

#include <ctype.h>

#include <iostream>

using namespace std;

int main (void)

{

setlocale(LC\_ALL, "Rus");

char str[4];

do{

cout << "Enter your age, please… ";

cin.getline(str,4);

if (isalpha (str[0])){

cout << "You entered a letter,";

cout << " try again\n";

continue;

}

if (iscntrl(str[0])){

cout << "You entered a control";

cout << " symbol, try again\n";

continue;

}

if (ispunct(str[0])){

cout << "You entered a ";

cout << "punctuation, try again\n";

continue;

}

for(int i= 0; i< strlen(str); i++){

if (!isdigit(str[i]))

continue;

else {

cout << "Your age is " << str;

getchar(); getchar();

return 0;

}

}

} while(true);

}



Здесь пользователю предлагается ввести свой возраст. Функция cin.getline() помещает в строку str введенную последовательность (до трех) символов, после чего следует проверка первого введенного элемента массива на принадлежность к буквенной, управляющей последовательностей или символам пунктуации. Если результат соответствующей проверки положительный, пользователю предлагается ввести данные повторно. В противном случае все введенные элементы строки проверяются на принадлежность к цифровому набору данных. Если хотя бы один из символов не удовлетворяет проверяемому условию, цикл ввода повторяется сначала. После корректного ввода данных на экран выводится сообщение о возрасте пользователя.

### Обращение строк

Функция обращения строки **strrev()** меняет порядок следования символов на обратный (реверс строки). Данная функция имеет прототип:

char\* strrev(char\* str)

Следующий пример демонстрирует работу функции strrev ()

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str1[]="Привет";

cout << "str1 = " << strrev(str1) << '\n';

getchar(); getchar();

return 0;

}



В результате на экране будет выведена строка " тевирп". Эта функция также преобразует строку - оригинал.

### Поиск символов

Одна из часто встречаемых задач при работе со строками - поиск отдельного символа или даже группы символов. Библиотека string.h предлагает следующий набор стандартных функций.

Функция нахождения символа в строке **strchr()** имеет следующий прототип:

char\* strchr(const char\* string, int c)

Данная функция производит поиск символа с в строке string и в случае успешного поиска возвращает указатель на место первого вхождения символа в строку. Если указанный символ не найден, функция возвращает NULL. Поиск символа осуществляется с начала строки. Ниже рассматривается фрагмент, осуществляющий поиск заданного символа в строке.

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char\* pStr, str[] = "Привет Джейк";

pStr = strchr(str, 'ж');

cout << "str1 = " << pStr << '\n';

getchar(); getchar();

return 0;

}



В результате работы программы указатель pStr будет указывать на подстроку "жэийк" в строке str.

**Функция strrchr()** осуществляет поиск заданного символа с конца строки. Она имеет следующий синтаксис:

сhar\* strrchr(const char\* string, int c)

Данная функция возвращает указатель на последний, совпавший с заданным с символ в строке string. Если символ не найден, возвращается значение NULL.

В следующем примере производится поиск заданного символа с конца строки:

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char\* pStr, str[] = "абвгдеёжзийк";

pStr = strrchr(str, 'и');

cout << "str1 = " << pStr << '\n';

getchar(); getchar();

return 0;

}



Таким образом, указатель pStr будет указывать на подстроку "ийк" в строке str.

**Функция strspn()** проводит сравнение символов одной строки с символами другой и возвращает позицию (начиная с нуля), в которой строки перестают совпадать. Данная функция имеет следующий прототип:

size\_t strspn(const char\* string, const char\* group)

Функция проверяет каждый символ строки string на соответствие каждому из символов строки group. В результате работы функции возвращается число совпавших символов.

Следующий пример демонстрирует использование данной функции:

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str1[] = "Загрузка параметров БД";

char str2[] = "Загрузка параметррррр";

int index=0;

index=strspn(str1, str2);

cout << "str1 = " << str1 << '\n';

cout << "str2 = " << str2 << '\n';

cout << "index = " << index;

getchar(); getchar();

return 0;

}



На экран будет выведено число 17, так как символы строки str и подстроки substr совпадают вплоть до 17-й позиции. Приведенная функция различает регистр символов.

**Функция strcspn()** имеет синтаксис:

size\_t strcpn(const char\* strl, const char\* str2)

Эта функция сопоставляет символы строки strl и str2 и возвращает длину строки strl, не входящей в str2. Таким образом, можно определить, в какой позиции происходит пересечение двух символьных массивов:

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str1[] = "abcdefghijk";

int index;

index = strcspn(str1, "elf");

cout << "str1 = " << str1 << '\n';

cout << "index = " << index;

getchar(); getchar();

return 0;

}



Переменная index получит значение 4, так как в этой позиции строки имеют первый общий элемент.

**Функция strpbrk()** объявлена следующим образом:

char\* strpbrk(const char\* strl, const char\* str2)

Эта функция отыскивает место вхождения в строку strl любого из символов строки str2. Если символы найдены, возвращается место первого вхождения любого символа из str2 в строку strl. В противном случае функция возвращает NULL.

Ниже иллюстрируется использование функции strpbrk():

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str1[] = "abcdefghijk";

char str2[] = "ecb";

char\* ptr;

ptr = strpbrk(str1, str2);

cout << "str1 = " << str1 << '\n';

cout << "str2 = " << str2 << '\n';

cout << "ptr = " << ptr;

getchar(); getchar();

return 0;

}



В результате будет выведена подстрока "bcdefghijk", так как символ 'b' из строки str2 встречается в строке str1 раньше других.

### Поиск подстрок

При необходимости поиска в одной строке последовательности символов, заданной в другом символьном массиве (подстроке, лексеме), стандартная библиотека string.h предлагает воспользоваться одной из следующих функций.

**Функция strstr()** описана следующим образом:

char\* strstr(const char\* str, const char\* substr)

Данная функция осуществляет сканирование строки str и находит место первого вхождения подстроки substr в строку str. В случае успешного поиска функция strstr возвращает указатель на первый символ строки str, начиная с которого следует точное совпадение части str обязательно со всей лексемой substr. Если подстрока substr не найдена в str, возвращается NULL.

Следующий пример показывает использование функции strstr ().

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

char str1[] = "Производится поиск элемента";

char str2[] = "поиск";

char\* ptr;

ptr=strstr(str1, str2);

cout << "str1 = " << str1 << '\n';

cout << "str2 = " << str2 << '\n';

cout << "ptr = " << ptr;

getchar(); getchar();

return 0;

}



На экран будет выведено "поиск элемента", так как подстрока, содержащаяся в str2, находится внутри строки strl и функция strstr () установит указатель ptr на соответствующий элемент символьного массива strl.

Чтобы найти место последнего вхождения подстроки в строку, можно воспользоваться следующим приемов: обе строки реверсируются с помощью функции strrev(), а затем полученный результат анализируется в функции strstr ().

**Функция strtok()** имеет синтаксис:

char\* strtok(char\* str, const char\* delim)

Эта функция выполняет поиск в строке str подстроки, обрамленной с обеих сторон любым символом-разделителем из строки delim. В случае успешного поиска данная функция обрезает строку str, помещая символ ' \0 ' в месте, где заканчивается найденная лексема. Таким образом, при повторном поиске лексемы в указанной строке str первым параметром следует указывать NULL. Так как strtok () модифицирует строку- оригинал, рекомендуется предварительно сохранять копию последней. Приведенный ниже пример иллюстрирует вышесказанное.

Предположим, необходимо разбить имеющееся в строковом массиве предложение по словам и вывести каждое из них на экран.

#include <string.h>

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

char str[]="Язык программирования C++";

char \*Delimiters = ".!?,;:\"'/0123456789 4> @#$%AS\*()<>{}[]~+ -=";

char \*ptr;

ptr = strtok(str, Delimiters);

if (ptr)

cout << ptr << '\n';

while (ptr) {

ptr = strtok(NULL, Delimiters);

if (ptr)

cout << ptr << '\n';

}

getchar(); getchar();

return 0;

}



В данной программе объявляется подлежащая анализу строка str, подстрока, содержащая набор разделителей Delimiters и указатель на символьный тип данных ptr. Вызов функции strtok (str, Delimiters) сканирует строку str, и как только в ней встретится любой символ, входящий в подстроку Delimiters (в данном случае это символ пробела), указатель ptr станет ссылаться на начало исходной строки до найденного символа. То есть ptr будет содержать:

\*ptr=”Язык”.

Благодаря тому, что функция strtok () помещает в найденном месте нуль-терминатор ('\n') - исходная строка модифицируется. Таким образом, массив символов str примет значение:

"программирования C++"

Осуществив проверку указателя ptr на существование в операторе if (ptr), найденное слово выводится на экран. Далее в цикле с помощью функции strtok () находится последний нуль-терминатор строки str;

ptr = strtok(NULL, Delimiters);

что, фактически, соответствует локализации следующего слова предложения, и найденная последовательность символов выводится на экран.

## Функции преобразования типа

Функции преобразования данных довольно часто используются, как следует из названия, для преобразования одного типа данных в другой тип. В приведенной ниже таблице перечислены основные функции, их прототипы подключаются в заголовочном файле stdlib.h.

**Таблица Преобразование данных**

|  |  |
| --- | --- |
| **Наименование** | **Краткое описание** |
| atof | преобразует строку символов в число с плавающей точкой |
| atoi | преобразует строку символов в строку типа int |
| atol | преобразует строку символов в число типа long |
| ecvt | преобразует число с плавающей точкой типа double в строку символов; десятичная точка и знак числа не включаются в полученную строку; позиция точки и знак числа возвращаются отдельно |
| fcvt | идентично ecvt, но округляет полученное значение до заданного числа цифр |
| gcvt | преобразует число с плавающей точкой типа double в строку символов, включая символ десятичной точки и используя специфицированное число цифр |
| itoa | преобразует число типа int в строку символов |
| ltoa | преобразует число типа long в строку символов |
| strtod | преобразует строку символов в число с плавающей точкой типа double |
| strtol | преобразует строку символов в число типа long |
| strtoul | преобразует строку символов в число типа unsigned long |
| ultoa | преобразует число типа unsigned long в строку символов |

Чаще всего данные функции используются для преобразования чисел, введенных в виде символьных строк, в числовое представление, а также для выполнения определенных арифметических операций над ними и обратное преобразование в строку символов. Рассмотрим самые широко используемые из них.

**Функция atoi()**, синтаксис которой

int atoi(const char\* ptr)

преобразует ASCIIZ-строку символов, на которую указывает ptr, в число типа int. Если в строке встречается символ, который не может быть преобразован, данная функция возвращает 0. В случае если преобразуемое число превышает диапазон представления типа int, возвращается только два младших байта числа.

В отличие от нее, функция atol() преобразует заданное строковое число в тип long. Эта функция имеет аналогичный синтаксис:

int atoi(const char\* ptr)

Если преобразуемое число превышает диапазон значений типа long, функция возвратит непредсказуемое значение.

Рассмотрим пример преобразования строки цифровых символов в целое и длинное целое.

#include <stdlib.h>

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

char str[]= "70000";

int i = atoi(str);

long l = atol(str);

cout << i << '\n';

cout << l;

getchar(); getchar();

return 0;

}

Если используется модель памяти, в которой тип int представляется двумя байтами, результат работы приведенной программы будет выглядеть следующим образом:

4464 70000

Дело в том, что число 70000 в шестнадцатеричной системе счисления представляется как 0x11170, но, поскольку функция atoi () при переполнении результата возвращает только два младших байта, переменная и примет шестнадцатеричное значение 0x1170, что эквивалентно десятичному 4464. Так как atol () оперирует с четырехбайтными числами, переполнения не произойдет и переменной l присвоится значение 70000.



**Функция atof()**, определенная как

double atof(const char\* ptr)

выполняет преобразование ASCIIZ-строки в число с плавающей точкой типа double. Строка символов должна быть представлена с учетом формата:

[пробелы] [знак] [цифры] [. ] [цифры] [е|Е [знак]цифры],

где

[пробелы] - последовательность пробелов или табуляторов;

[знак] - символ ' + ' или ' - ';

[цифры] - десятичные цифры;

[е |Е] символ показателя степени.

Преобразование символов прекращается, как только найден первый неконвертируемый символ или достигнут конец строки.

Функции обратного преобразования **itoa() и ltoa()** производят конвертирование чисел типа int и long соответственно. Они имеют следующий синтаксис:

char \*\_ltoa(long num, char\* str, int radix);

и

char\* itoa(int num, char\* str, int radix);

или

char\* \_itoa(int num, char \*str, int radix);

Данные функции принимают в качестве аргумента число num и преобразуют его в строку str с учетом основания системы счисления, представленной в переменной radix. Следующий фрагмент программы преобразует целое число 98765 в строку, используя десятеричную систему счисления:

#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

int numb = 98765;

char str[10];

itoa(numb, str, 10);

cout << "numb = " << numb << '\n';

cout << " str = " << str << '\n';

getchar(); getchar();

return 0;

}



**Функция strtod ()** преобразует строку символов в число с плавающей точкой. Ее синтаксис имеет следующий вид:

double strtod(const char \*s, char \*\*endptr);

Эта функция так же, как и функция atof (), преобразует строку, на которую указывает s, в число типа double, с тем лишь отличием, что в случае прекращения конвертирования строки возвращает указатель на первый непреобразуемый символ. Это позволяет организовать в дальнейшем обработку оставшейся части строки.

**Функция gcvt()** имеет прототип:

char\* gcvt(double val, int ndec, char \*buf

и осуществляет конвертирование числа val типа double в ASCIIZ-строку, помещая ее в буфер buf. Если число цифр, подлежащих преобразованию, меньше целого числа, указанного в ndec, в преобразованном числе указываются символы знака и десятичной точки, при этом младшие разряды дробной части отбрасываются. В противном случае число преобразуется в экспоненциальную форму. Функция возвращает указатель на начало сформированной строки.

Следующий ниже пример демонстрирует использование функции gcvt () для преобразования чисел, имеющих различное представление в массивы цифровых символов.

#include <stdlib.h>

#include <iostream>

using namespace std;

int main (void)

{

setlocale(LC\_ALL, "Rus");

char str[10];

double num;

// Значащих цифр:

int sig = 4;

// Обычное представление числа

num = 3.547;

gcvt(num, sig, str);

cout << str << '\n';

// Отрицательное число

num = -843.7105;

gcvt(num, sig, str);

cout << str << '\n';

// Экспоненциальное представление

num = 0.135e4;

gcvt(num, sig, str);

cout << str << '\n';

getchar(); getchar();

return 0;

}

В результате будет выведено:

3.547

-843.7

1350



## Функции printf() и scanf().

Функции printf() и scanf() дают нам возможность взаимодействовать с программой. Мы называем их функциями ввода-вывода. Это не единственные функции, которыми мы можем воспользоваться для ввода и вывода данных с помощью программ на языке Си, но они наиболее универсальны. Указанные функции не входят в описание языка Си. И действительно, при работе с языком Си реализация функций ввода-вывода возлагается на создателей компилятора; это дает возможность более эффективно организовать ввод-вывод на конкретных машинах. Однако в интересах обеспечения совместимости различные системы имеют дело с некоторыми вариантами функций scanf() и printf(). Все, о чем мы здесь говорим, должно быть в основном справедливо для большинства систем, но не удивляйтесь, если обнаружите некоторые отличия в имеющейся у вас версии.

Обычно функции printf() и scanf() «работают» во многом одинаково — каждая использует «управляющую строку» и список «аргументов». Сначала мы рассмотрим работу функции printf(), затем scanf().

Инструкции, передаваемые функции printf(), когда мы «просим» ее напечатать некоторую переменную, зависят от того, какого типа эта переменная. Например, при выводе на печать целого числа применяется формат %d, а при выводе символа — %с. Ниже перечислены все форматы, указываемые при обращениях к функции printf(), а затем показано, как они используются. Каждому формату соответствует тип выводимой (с их помощью) информации, причем первые пять покрывают большинство возникающих потребностей, а остальные четыре применяются достаточно редко.

**Формат Тип выводимой информации**

%d Десятичное целое число

%с Один символ

%s Строка символов

%е Число с плавающей точкой, экспоненциальная запись

%f Число с плавающей точкой, десятичная запись

%g Используется вместо записей %f или %е, если он короче

%u Десятичное целое число без знака

%o Восьмеричное целое число без знака

%х Шестнадцатеричное целое число без знака

Посмотрим теперь, как эти форматы применяются.

### Использование функции printf().

Функция printf () имеет следующий прототип:

int printf(const char \*format[, argument,...]);

Она позволяет осуществлять форматированный вывод в стандартный поток вывода stdout. Функция принимает последовательность аргументов, применяя к каждому аргументу спецификатор формата, который содержится в строке форматирования format. Если аргументов меньше, чем спецификаторов формата, результат непредсказуем. С другой стороны, если аргументов больше, чем спецификаторов формата, лишние аргументы просто игнорируются. В случае успеха функция возвращает число выведенных байтов (без учета завершающего нуль-символа); в случае ошибки - значение EOF.

Строка форматирования в функции printf() управляет тем, как эта функция будет преобразовывать, форматировать и выводить свои аргументы. Она представляет собой символьную строку, которая содержит два типа объектов:

* Простые символы, которые копируются в выходной поток;
* Спецификаторы формата, которые применяются к аргументам, выбираемым из списка аргументов
* Спецификаторы формата имеют вид:

%[flags][width] [.prec][F|N|h|1|L]type\_char

Каждый спецификатор формата начинается с символа процента (%), после которого идут необязательные спецификаторы в порядке, указанном в таблице.

**Компоненты спецификатора формата и их назначение**

|  |  |  |
| --- | --- | --- |
| **Компонент** | **Обязательный или нет** | **Назначение** |
| [flags] | нет | Флаговые символы. Управляют выравниванием, знаком числа, десятичной точкой, конечными нулями, префиксами для восьмеричных и 16-ричных чисел. |
| [width] | нет | Спецификатор ширины (поля). Указывает минимальное число выводимых символов (дополняемое в случае необходимости пробелами или нулями). |
| [prec] | нет | Спецификатор точности. Указывает максимальное число выводимых символов; для целых чисел - минимальное чисто выводимых цифр. |
| [F|N|h|l|L] | нет | Модификатор размера. Переопределяет размер входного аргумента по умолчанию. |
| Type\_char | да | Символ типа преобразования. |

Флаговые символы (флажки) могут появляться в любом порядке и комбинации.

Типы флажков и их смысл приведены в таблице.

**Флажки и их назначение**

|  |  |
| --- | --- |
| **Флажок** | **Назначение** |
| \* | Выравнивает результат преобразования влево, дополняя его справа пробелами. По умолчанию (т.е. когда этот флажок не задан) результат преобразования выравнивается вправо и дополняется слева нулями или пробелами. |
| + | Если результат преобразования имеет знак, этот знак всегда выводится Это означает, что для положительных чисел выводится знак плюс (+). |
| Пробел (' ') | Если значение неотрицательно, вывод начинается с пробела вместо знака плюс. Отрицательные числа по-прежнему выводятся со знаком минус. |
| # | Задает, что аргумент преобразуется с использованием альтернативной формы. |

**Замечание**: Флажок "плюс" (+) имеет преимущество перед флажком "пробел" (' '), если они заданы оба.

Спецификатор ширины поля устанавливает минимальную ширину поля для выводимого значения. Ширина задается одним из двух способов:

* непосредственно, с помощью десятичного числа;
* косвенно, с помощью символа звездочки (\*).

Если вы используете в качестве спецификатора ширины звездочку, следующий аргумент в списке аргументов (который должен быть целым числом) задает не выводимое значение, а минимальную ширит поля для вывода. Если ширина поля вывода не указана или мала, это не вызывает усечения значения до ширины поля. Вместо этого поле расширяется до нужного размера, чтобы вместить результат преобразования значения. Спецификаторы ширины и их влияние на вывод перечислены в таблице.

**Спецификаторы ширины и их влияние на вывод**

|  |  |
| --- | --- |
| **Спецификаторы ширины** | **Влияние на вывод** |
| N | По крайней мере, n стволов выводятся Если выводимое значение имеет менее чем n символов, вывод дополняется пробелами (справа, если укатан флажок минус (-), слева - в противном случае). |
| On | Выводится по меньшей мере n символов. Если выводимое значение имеет меньше чем n символов, оно дополняется слева нулями. |
| \* | Список аргументов предоставляет спецификатор ширины, который должен предшествовать реально выводимому аргументу. |

Спецификатор точности функции printf() устанавливает максимальное число выводимых символов или минимальное число выводимых цифр для целых чисел. Этот спецификатор всегда начинается с точки, чтобы отделить его от предыдущего спецификатора ширины. Как и спецификатор ширины, он задается одним из двух способов:

* непосредственно, с помощью десятичного числа;
* косвенно, с помощью символа звездочки (\*).

Если вы используете звездочку в качестве спецификатора точности, то следующий аргумент в списке аргументов (который должен быть целым числом) задает точность. Если вы задаете звездочки в качестве спецификаторов ширины и точности, то вслед за ними в списке аргументов должен идти спецификатор ширины, затем спецификатор точности, а за ними - само выводимое значение.

Символ типа преобразования задает преобразование типа выводимого аргумента. Возможные значения символа типа преобразования и их назначение представлены в таблице. Информация в этой таблице основывается на предположении, что спецификатор формата не содержит ни флажков, ни спецификатора ширины, ни спецификатора точности, ни модификатора размера входного значения.

**Значения символа типа преобразования и их назначение**

|  |  |  |  |
| --- | --- | --- | --- |
| **Символ типа преобразования** | **Ожидаемый ввод** | **Формат вывода** | |
| **Числовые значения** | | | |
| d | Целое число | | десятичное целое со знаком |
| i | Целое число | | десятичное целое со знаком |
| о | Целое число | | восьмеричное целое без знака |
| u | Целое число | | десятичное целое без знака |
| x | Целое число | | шестнадцатеричное целое без знака (причем для 16-рнчных цифр используются символы а, b, с, d, e, f) |
| X | Целое число | | шестнадцатеричное целое без знака (причем для 16-ричных цифр используются символы А, В. С, D, E, F) |
| f | Число с плавающей точкой | | Число с плавающей точкой в виде [-]dddd.dddd. |
| e | Число с плавающей точкой | | Число с плавающей точкой в виде [-]d.dddd или e[+/-]ddd |
| g | Число с плавающей точкой | | Число с плавающей точкой в форме е или f с учетом заданного значения и точности. Конечные нули и десятичная точка выводятся, если это необходимо |
| E | Число с плавающей точкой | | То же самое, что и e; но для экспоненты используется символ Е |
| G | Число с плавающей точкой | | То же самое, что и g; но для экспоненты используется символ Е, если используется формат e |
| **Символы** | | | |
| с | Символ | Один символ | |
| s | Указатель на строку | Выводит символы до тех пор, пока не встретится 0-символ или не будет достигнута заданная точность | |
| % | Никакой | Выводит символ процента | |
| **Указатели** |  |  | |
| n | Указатель на int | Сохраняет по адресу, указанному входным аргументом, число символов, записанных до сих пор. | |
| P | Указатель | Выводит входной аргумент как указатель; формат зависит от используемой модели памяти. Указатель будет представлен в виде XXXXYYYY или YYYY |(только смещение)\_\_ | |

Модификаторы размера определяют, как функция printf() интерпретирует следующий входной аргумент. Возможные значения модификатора размера и их действие представлены в таблице.

**Модификаторы размера и их действие на входную величину**

|  |  |  |
| --- | --- | --- |
| **Модификатор размера** | **Символ типа преобразования** | **Приписываемый входному аргументу тип** |
| F | p s | far-указатель |
| N | N | near-указатель |
| Н | D 1 о и х X | short int |
| L | d i о и х X | long int |
| L | e Е f g G | double |
| L | e Е f g G | long double |
| L | d i о u x X | \_int64 |
| h | с С | Однобайтовый символ |
| l | с С | Широкий символ |
| h | s S | Строка однобайтовых символов |
| l | s S | Строка широких символов |

Модификаторы F и N изменяют интерпретацию входного аргумента в случае, если задан символ преобразования %р, %s или %n, трактуя его, соответственно, как дальний или ближний указатель. По умолчанию для этих символов преобразования входной аргумент трактуется как указатель в используемой модели памяти. Модификаторы размера h, 1 и L переопределяют величину по умолчанию числовых входных аргументов, как указано в таблице.

Приведем пример, демонстрирующий различные возможности функции printf():

#include <stdio.h>

int main()

{

setlocale(LC\_ALL, "Rus");

char ch = 'A', \*string = "string";

int count = -1234;

double fp =-123.4567;

//Отображение целых

printf("Integer formats: \n"

"\tDecimal: %d Justified: %.6d Unsigned: \*u\n",

count, count, count, count);

printf("Decimal \*d as:\n\tHex: "

"%Xh C-hex: Ox%x Octal: %o\n", count, count, count, count);

//Отображение чисел с разным основанием

printf("Digits 10 equal:\n\tHex: %i Octal: %i Decimal: %i\n",

0x10, 010, 10);

// Отображение символов

printf("Characters:\n%10c%5hc\n",ch, ch);

//Отображение строк

printf("Strings:\n%16s\n%16.4hs\n", string, string);

//Отображение чисел с плавающей точкой

printf("Read numbers:\n%f %.2f %e %E\n", fp, fp, fp, fp);

//Отображение указателя

printf("\nAddress as:\t%p\n", &count);

//Вывод числа символов

printf("\nDisplay to here:\n");

printf("1234567890123456%u78901234567890\n" ,&count);

printf("Number displayed: %d\n\n", count );

getchar(); getchar();

return 0;

}

При выполнении программа выводит на экран:

Integer formats:

Decimal: -1234 Justified: -001234

Unsigned: 4294966062 Decimal -1234 as:

Hex: FFFFFB2Eh C-hex: Oxfffffb2e

Octal: 37777775456 Digits 10 equal:

Hex: 16 Octal: 8 Decimal: 10

Characters:

A A

Strings:

string

str1

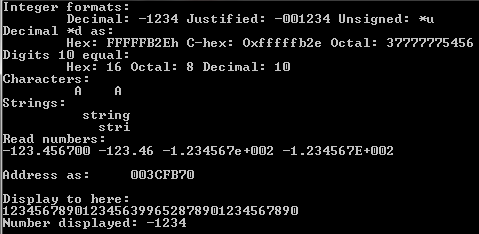
Read number:

123.456700 123.46 1.234567e+02 1.234567E+02

Address as: 0012FF80

Display to here:

12345678 9012345678901234567890 Number displayed: 16



**Функция вывода sprintf()** является в некотором смысле обратной функции sscanf(): она осуществляет вывод в символьную строку. Функция имеет следующий прототип:

int sprintf (char \*buffer, const char \*format[, argument,...]);

Она во всем аналогична функции printf(), но осуществляет форматированный вывод не на стандартное устройство вывода, а в символьную строку, указатель на которую задан параметром buffer.

Приведем программу, иллюстрирующую обсуждаемые вопросы.

/\* печать чепухи\*/

#define PI 3.14159

main()

{

int number = 5;

float ouzo = 13.5;

int cost = 31000;

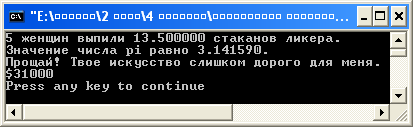
printf("%d женщин выпили %f стаканов ликера.\n", number, ouzo);

printf("Значение числа pi равно %f.\n", PI);

printf("Прощай! Твое искусство слишком дорого для меня. \n");

printf("%c%d\n", '$', cost);

}



Формат, указываемый при обращении к функции printf(), выглядит следующим образом:

рrintf(Управляющая строка, аргумент1, аргумент2, ...,);

*Аргумент1, Аргумент2* и т. д. — это печатаемые параметры, которые могут быть переменными, константами или даже выражениями, вычисляемыми вначале, перед выводом на печать.

*Управляющая строка* — строка символов, показывающая, как должны быть напечатаны параметры. Например, в операторе

printf("%d женщин выпили %f стаканов ликера\n", number, ouzo);

управляющей строкой служит фраза в кавычках (учитывая предыдущие замечания, это — строка символов), a number и ouzo — аргументы или в данном случае значения двух переменных.

05_05

Приведем еще пример.

printf("Значение числа pi равно %f.\n", PI);

На этот раз список аргументов содержит только один элемент — символическую константу PI.

Мы видим, что в управляющей строке содержится информация двух различных видов:

1. Символы, печатаемые текстуально.
2. Идентификаторы данных, называемые также «спецификациями преобразования».

_05_07

Каждому аргументу из списка, следующего за управляющей строкой, должна соответствовать одна спецификация преобразования. Горе вам, если вы забудете это основное требование! Никогда не пишите, например, так:

printf(" Количество слизняков %d, червяков %d.\n" , score1);

Здесь отсутствует аргумент для второй спецификации преобразования %d. Способ проявления этой ошибки целиком зависит от вашей вычислительной системы, но в лучшем случае вы получите бессмыслицу.

Если вам нужно напечатать какую-нибудь фразу, то нет необходимости использовать спецификацию преобразования; если же требуется только вывести данные на печать, то можно обойтись и без использования комментария. Поэтому каждый из операторов, приведенных ниже, вполне приемлем.

printf("Прощай! Твое искусство слишком дорого для меня\n");

printf(" %c%d\n", '$', cost);

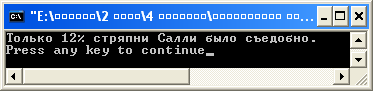
Заметим, что во втором примере первый аргумент из печатаемого списка является символьной константой, а не переменной.

Поскольку символ % используется в функции printf() для идентифицирования спецификаций преобразования, возникает небольшая проблема в том случае, если вам нужно напечатать сам символ %. Если просто написать один знак %, то компилятор примет его за ошибочную спецификацию преобразования. Выходом из создавшейся ситуации служит довольно простое решение — писать два символа % подряд:

рс = 2\*6;

printf("Только %d%% стряпни Салли было съедобно.\n", рс);

Результат работы программы будет выглядеть следующим образом:



### Модификаторы спецификации преобразования, используемые в функции printf().

Мы можем несколько расширить основное определение спецификации преобразования, поместив модификаторы между знаком % и символом, определяющим тип преобразования. В приводимой ниже таблице дан список тех символов, которые вы имеете право туда поместить. При использовании одновременно нескольких модификаторов они должны быть указаны в том порядке, в котором перечислены в таблице. Заметим, что при этом допускаются не все комбинации.

|  |  |
| --- | --- |
| **Модификатор** | **Значение** |
| — | Аргумент будет печататься с левой позиции поля заданной ширины (как объяснено ниже). Обычно печать аргумента оканчивается в самой правой позиции поля  Пример: %—10d |
| строка цифр | Задает минимальную ширину поля. Большее поле будет использоваться, если печатаемое число или строка не помещаются в исходном поле  Пример: %4d |
| строка цифр | Определяет точность: для типов данных с плавающей точкой — число печатаемых цифр справа от десятичной точки; для символьных строк — максимальное число печатаемых символов  Пример: %4.2f (две десятичные цифры для поля шириной в четыре символа) |
| l | Соответствующий элемент данных имеет тип long, а не int.  Пример: %ld |

Примеры

Посмотрим, как эти модификаторы работают. Начнем с того, что продемонстрируем влияние модификатора ширины поля на печать целого числа. Рассмотрим следующую программу:

main()

{

printf ("/%d/\n", 336);

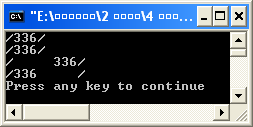
printf ("/%2d/\n", 336);

printf ("/%10d/\n", 336);

printf ("/%-10d/\n", 336);

}

Эта программа печатает одно и то же значение четыре раза, но использует при этом четыре различные спецификации преобразования. Мы вводим также символы /, чтобы вы могли видеть, где начинается и кончается каждое поле. Результат выполнения программы выглядит следующим образом:



Первая спецификация преобразования %d не содержит модификаторов. Мы видим, что поле печати здесь имеет ширину, равную количеству цифр данного целого числа. Это так называемый выбор «по умолчанию», т. е. результат действия компилятора в случае, если вы не дали ему никаких дополнительных инструкций. Вторая спецификация преобразования — %2d. Она указывает, что ширина поля должна равняться 2, но, поскольку число состоит из трех цифр, поле автоматически расширяется до необходимого размера. Следующая спецификация %10d показывает, что ширина поля равна 10. И действительно, между символами / имеется семь пробелов и три цифры, причем число сдвинуто к правому краю поля. Последняя спецификация %— 10d также указывает ширину поля, равную 10, а знак — приводит к сдвигу всего числа к левому краю, как показано в приведенном выше примере. Когда вы привыкнете к этой системе обозначений, она покажется вам простой и вы сумеете по вашему усмотрению менять вид выходной информации.

Рассмотрим теперь некоторые форматы, соответствующие данным с плавающей точкой. Допустим, у нас имеется следующая программа:

main()

{

printf ("/%f/\n", 1234.56);

printf ("/%10.6e/\n" , 1234.56);

printf ("/%4.2f/\n", 1234.56);

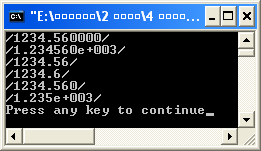
printf ("/%3.1f/\n", 1234.56);

printf ("/%6.3f/\n" , 1234.56);

printf ("/%10.3e/\n" , 1234.56);

}

На этот раз результат работы программы будет выглядеть так:



Мы снова начинаем с варианта, выбранного по умолчанию, т. е. со спецификации %f. В этом случае имеется две величины, значения которых используются по умолчанию: ширина поля и число цифр справа от десятичной точки. Вторая величина задает шесть цифр, а ширина поля берется такой, чтобы в нем могло поместиться число. Заметим, что печатаемое число несколько отличается от исходного. Это происходит потому, что на печать выводится 10 цифр, в то время как числа с плавающей точкой в нашей системе изображаются приблизительно с точностью до 6 или 7 цифр.

Рассмотрим вариант по умолчанию для спецификации %е. Как мы видим, при ее использовании печатается одна цифра слева от десятичной точки и шесть справа. В результате получается слишком много цифр! Чтобы избежать этого, необходимо задать число цифр справа от десятичной точки, и последние четыре оператора программы реализуют как раз указанную возможность. Обратите внимание на то, как четвертый и шестой операторы производят округление выводимых данных.

Теперь исследуем некоторые варианты строк. Рассмотрим пример:

#define BLURB "Выдающееся исполнение!"

main()

{

printf(" /%2s/\n" , BLURB);

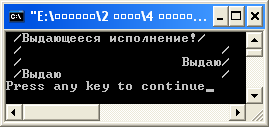
printf(" /%25.s/\n" , BLURB);

printf(" /%25.5s/\n" , BLURB);

printf(" /%-25.5s/\n" , BLURB);

}

Вот результат работы программы:



Обратите внимание на то, как поле расширяется для того, чтобы поместились все указанные символы. Заметим также, как спецификация точности ограничивает число символов, выводимых на печать. Символы .5 в спецификации формата указывают функции printf() на необходимость напечатать только пять символов.

Теперь вы ознакомились с некоторым количеством примеров. А знаете ли вы, как подготовить оператор печати, чтобы напечатать нечто вроде следующей фразы:

Семья NAME, возможно, лишь на XXX.XX долларов богаче!

Здесь NAME и XXX.XX представляют значения соответствующих переменных в программе, скажем name [40] и cash. Вот одно из решений:

printf(" Семья %s, возможно, лишь на %.2f долларов богаче!\n", name, cash);

До сих пор мы без тени сомнения применяли спецификации преобразования для переменных разных типов: например, %1 для типа float и т. д. Но, как мы уже видели в нашей программе поиска кода ASCII, для некоторого символа функцию printf() можно использовать также для преобразования данных из одного типа в другой! Мы не намерены, однако, терять чувство реальности и по-прежнему будем работать с целыми типами.

### Использование функции printf() для преобразования данных.

Здесь мы снова займемся выводом на печать целых чисел. Поскольку мы уже осведомлены о полях, то не будем заботиться об использовании символа /, чтобы отмечать их начало и конец:

main()

{

printf(" %d\n", 336);

printf(" %o\n", 336);

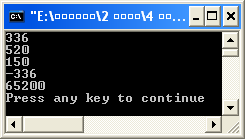
printf(" %x\n", 336);

printf(" %d\n", -336);

printf(" %u\n" , -336);

}

В нашей системе результат будет выглядеть следующим образом:



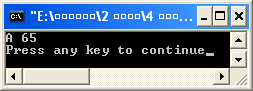
Как вы, по-видимому, и ожидали, при использовании спецификации %d будет получено число 336 точно так же, как в примере, обсуждавшемся чуть выше. Но давайте посмотрим, что произойдет, когда вы «попросите» программу напечатать это десятичное целое число в восьмеричном коде. Она напечатает число 520, являющееся восьмеричным эквивалентом 336 (5\*64+2\*8 + 0\*1 = 336). Аналогично при печати этого числа в шестнадцатеричном коде мы получим 150.

Таким образом, мы можем использовать спецификации, применяемые для функции printf() с целью преобразования десятичных чисел в восьмеричные или шестнадцатеричные и наоборот. Или же если вы захотите напечатать данные в желаемом для вас виде, то необходимо указать спецификацию %d для получения десятичных чисел, %о — для восьмеричных, а %х— для шестнадцатеричных. При этом не имеет ни малейшего значения, в какой форме число первоначально появилось в программе.

Сделаем еще несколько замечаний относительно вывода на печать. Печать числа -336 при использовании спецификации %d не вызывает никакого затруднения. При применении же спецификации %u (unsigned — беззнаковая) получаем число 65200, а не 336, как можно было бы ожидать. Причина получения такого результата лежит в способе представления отрицательных чисел в нашей системе. Здесь используется так называемый «дополнительный код». Числа от 0 до 32767 отображаются обычным образом, а от 32768 до 65535 представляют отрицательные числа, причем 65535 кодирует число — 1, 65534 — число -2 и т. д. Поэтому числу -336 соответствует 65536 — 336 = 65200. Этот метод применяется не во всех системах. Тем не менее отсюда следует вывод: не ожидайте, что спецификация преобразования %u приводит просто к отбрасыванию знака числа.

Сейчас мы переходим к обсуждению интересного примера, которого мы уже касались ранее, а именно к использованию функции printf() для нахождения кода ASCII некоторого символа. Например, оператор

printf(" %c%d\n" , ' А', ' А'); выдаст следующий результат:

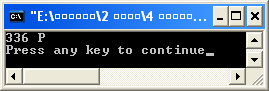


А — это буква, а 65 — десятичный код ASCII символа А. Мы могли бы использовать спецификацию %o, если бы хотели получить восьмеричный код ASCII символа А.

Все вышесказанное дает хороший способ нахождения кодов ASCII для различных символов и наоборот. Вполне возможно, конечно, что вы предпочтете ему поиск кодов в приложении Ж.

Что произойдет, если вы попробуете преобразовать число, большее 255, в символ? Следующая строка и результат ее выполнения дадут ответ на этот вопрос:

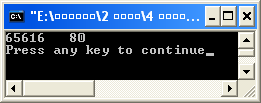
printf(" %d %c\n" , 336, 336);



Десятичный код ASCII символа Р равен 80, а 336 — это 256 + 80. Данное число, очевидно, интерпретируется по модулю 256. (Это математический термин, обозначающий остаток от деления числа на 256.) Другими словами, всякий раз при получении числа, кратного 256, отсчет начинается сначала, и 256 рассматривается как 0, 257 — как 1, 511 — как 255, 512 — как 0, 513 — как 1 и т. д. И наконец, попытаемся напечатать число (65616), превышающее максимальное значение, которое могут принимать данные типа int в нашей системе (32767):

printf("%d %d\n", 65616, 65616);

Результат будет выглядеть так:



Мы снова видим, что действия выполняются по «модулю». На этот раз счет ведется группами по 65536. Числа между 32767 и 65536 будут выводиться на печать как отрицательные из-за способа их представления в памяти машины. Системы с разными размерами ячеек памяти, отводимых под данные целого типа, ведут себя в общем одинаково, но при этом дают разные числовые значения.

Мы не исчерпали всех возможных комбинаций данных и спецификаций преобразования, поэтому вы можете пытаться экспериментировать сами. Но будет лучше, конечно, если вы сможете заранее предсказать результат, который будет получен при печати данных, когда используется какая-нибудь спецификация преобразования, выбранная вами.

### Применение функции scanf().

Функция scanf() имеет следующий прототип:

Int scanf(const char \*format[, address,...]);

Она позволяет осуществить форматный ввод информации из стандартного потока ввода. Функция сканирует (откуда и её название) последовательность входных полей по одному символу и форматирует каждое поле в соответствии со спецификатором формата, преданном в строке форматирования format. Для каждого поля должен существовать спецификатор формата и адрес переменной, предназначенной для размещения результата преобразования поля в соответствии с заданным спецификатором формата. Функция может прекратить сканирования поля до достижения конца поля (символа-заполнителя) или прекратить сканирование полей вообще. В случае успеха scanf () возвращает число успешно просканированных, преобразованных и сохраненных входных полей. В это число не входят не сохраненные поля. Если ни одно поле не было сохранено, функция возвращает 0. Если делается попытка прочесть конец файла или конец строки, она возвращает EOF.

Строка форматирования format представляет собой символьную строку, которая содержит объекты трех типов:

* символы-заполнители;
* символы, отличные от символов-заполнителей;
* спецификаторы формата.

Символы-заполнители - это пробел, символ табуляции (\t) и перевод строки (\n). Эти символы считываются из форматной строки, но не сохраняются.

Символы, отличные от символов-заполнителей — это все остальные ASCII-символы, за исключением знака процента (%). Эти символы в форматной строке считываются. но не сохраняются.

Спецификаторы формата направляют процесс сканирования, считывания и преобразования символов из входного поля в заданные переменные, заданные своими адресами. Каждому спецификатору формата должен соответствовать адрес переменной. Если спецификаторов формата больше, чем переменных, результат непредсказуем. Наоборот, если имеется больше переменных, чем спецификаторов формата, они игнорируются.

Спецификаторы формата имеют следующий вид:

% [\*] [width] [F|N] [h|l|L] type\_char

Каждый спецификатор формата начинается с символа процента (%).

После знака процента идут следующие знаки в указанном в таблице порядке.

**Знаки спецификатора формата и их назначение**

|  |  |  |
| --- | --- | --- |
| **Компонент** | **Обязательный или нет** | **Назначение** |
| [\*] | Нет | Символ подавления присвоения переменной значения следующего поля. Текущее поле ввода сканируется, но не сохраняется в переменной. Предполагается, что аргумент, соответствующий спецификатору формата, содержащему звездочку, имеет тип, укачанный символом типа преобразования (type\_char), который идет за звездочкой |
| [width] | Нет | Спецификатор ширины поля. Задает максимальное число считываемых символов. Функция может прочесть меньше символов, если в потоке ввода встретится символ-заполнитель или непреобразуемый символ |
| [F|N] | Нет | Модификатор величины указателя. Переопределяет величину по умолчанию аргумента, задающего адрес: N = near pointer F = far pointer |
| [h] l [L] | Нет | Модификатор типа аргумента. Переопределяет тип по умолчанию аргумента, задающего адрес: h = short int l = long int, если type\_char задает преобразование в целое 1 = double если type\_char задает преобразованный тип с плавающей точкой, L = long double верно только для преобразования в тип с плавающей точкой). |
| type\_char | Да | Символ типа (преобразования) (или спецификатор преобразования) |

**Символы типа преобразования**

|  |  |  |
| --- | --- | --- |
| **Тип** | **Ожидаемый ввод** | **Тип аргумента** |
| **Числовой** | | |
| d | Десятичное целое | Указатель на целое (int \*arg) |
| D | Десятичное целое | Указатель на длинное целое (long \*arg) |
| e, E | Число с плавающей точкой | Указатель на float (float \*arg) |
| f | Число с плавающей точкой | Указатель на float (float \*arg) |
| g,G | Число с плавающей точкой | Указатель на float (float \*arg) |
| о | Восьмеричное целое | Указатель на int (int \*arg) |
| О | Восьмеричное целое | Указатель на long (long \*arg) |
| i | Десятичное, восьмеричное или шестнадцатеричное целое | Указатель на int (int \*arg) |
| l | Десятичное, восьмеричное или шестнадцатеричное целое | Указатель на long (long \*arg) |
| u | Беззнаковое десятичное целое | Указатель на unsigned int (unsigned int \*arg) |
| U | Беззнаковое десятичное целое | Указатель на unsigned long (unsigned long \*argl |
| X | Шестнадцатеричное целое | Указатель на int (int \*arg) |
| X | Шестнадцатеричное целое | Указатель на int (int \*arg) |
| **Символы** | | |
| S | Символьная строка | Указатель на символьный массив (char arg[]) |
| с | Символ | Указатель на символ (char \*arg), если задана ширина поля для символов (например. %5с). Указатель на массив из N символов (char arg[N]) |
| % | % символ | Никакое преобразование не выполняется: знак % сохраняется |
| **Указатели** | | |
| n | Указатель на int (int \*arg) | Число успешно прочитанных символов, вплоть до %n сохраняется в этом int. |
| Р | Шестнадцатеричная форма  YYYY:ZZZZ или ZZZZ | Указатель на объект (far\* или near\*). По умолчанию %р преобразовывает в указатель, величина которого принята в данной модели памяти |

Кроме того, в качестве спецификатора формата можно задать совокупность символов, заключенных в квадратные скобки. Соответствующий ему аргумент должен указывать на массив символов. Эти квадратные скобки определяют совокупность символов поиска. Если первым символом в квадратных скобках является каре (^), то условие соответствия инвертируется: во входном поле ищутся символы, не соответствующие указанным в квадратных скобках. Например.

%[abed] Поиск во входном поле любого из символов а, b, c и d

%[^abcd] Поиск во входном поле любого из символов, за исключением а, b, с и d.

Вы также можете указать диапазон поиска, задав символы через тире в качестве совокупности символов поиска. Например,

%[ 0 - 9] Поиск во входном поле любых десятичных цифр.

%[0-9A-Za-z] Поиск во входном поле любых десятичных цифр и букв.

Сканирование поля прекращается, если очередной символ не попадает в совокупность символов поиска.

Функция может прекратить сканирование какого-то поля еще до достижения его естественного конца (символа-заполнителя) или даже прекратить сканирование совсем. В частности, это происходит, когда в поле ввода появляется символ, не соответствующий по типу заданному преобразованию в спецификаторе формата. В этом случае конфликтующий символ остается в поле ввода. Таким образом, использование функции scanf() может приводить к неожиданным результатам, если вы отходите от заданного формата. По этой причине рекомендуется вместо данной функции использовать комбинацию функций gets () и sscanf().

Рассмотрим пример, демонстрирующий возможности функции scanf():

#include <stdio.h>

using namespace std;

int main ()

{

setlocale(LC\_ALL, "Rus");

int i, res;

float fp;

char c, s[81];

printf( "\n\nEnter an int, a float, char and string\n");

res = scanf ( "%d %f %c %s",&i, &fp, &c, &s);

printf( "\nThe number of fields input is %d\n", res);

printf("The contents are:\n %d %f %c %s \n", i, fp, c, s);

getchar(); getchar();

return 0;

}

При выполнении программа выводит на экран:

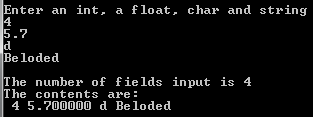
Enter an int, a float, char and string

1234.5678 F string

The number of fields input is 4

The contents are:

1234 1234.567749 F string



Упомянутая выше **функция sscanf()** имеет следующий прототип:

Int sscanf(const char \*buffer, const char \*format[, address,...]);

Она во всем аналогична функции scanf (), за исключением того, что выполняет сканирование и форматирование ввода из строки символов. В частности, она использует те же спецификаторы формата в строке форматирования. Прочитанные из символьной строки в соответствии с заданным форматом значения помещаются в переменные, адреса которых заданы в списке аргументов функции. В случае успеха sscanf () возвращает число входных полей, успешно просканированных, преобразованных к нужному формату и сохраненных в переменных. Если в sscanf () делается попытка прочесть конец строки, она возвращает EOF. В случае ошибки (если ни одно поле не было сохранено в переменной), она возвращает 0. Приведем пример использования функции sscanf ():

#include <stdio.h>

using namespace std;

int main ()

{

setlocale(LC\_ALL, "Rus");

char szBufl[] = "This function can parse string\*\*\*";

char szBuf2[] = "Integer i = 12345, fl = 1234.567";

char s[5] [11]; char str1[11] , str2[11];

char c1, c2, c3; int i; float f1;

//Input various data from string buffer

sscanf( szBufl, "%s %s %s %s %s",s[0], s[1], s[2],s[3], s[4]);

sscanf ( szBuf2, "%s %c %c %d %s %c %f",str1, &c1, &c2, &i, str2, &c3, &f1);

//Output the data read

printf( "sscanf() parse this string: \n""%s %s %s %s %s\n",

s[0], s[1], s[2],s[3], s[4]);

printf("szBuf2 contains:\n""%s %c %c %d %s %c %8.3f",str1, c1, c2, i, str2, c3, f1);

getchar(); getchar();

return 0;

}

При выполнении программа выводит на экран:

sscanf() parse this string:

\*\*\*This function can parse string\*\*\*

szBuf2 contains:

Integer i = 12345,fl = 1234.567



Поскольку в дальнейшем мы будем пользоваться функцией scanf() лишь эпизодически, мы рассмотрим здесь только основные особенности ее применения.

Так же как для функции printf(), для функции scanf() указываются управляющая строка и следующий за ней список аргументов. Основное различие двух этих функций заключается в особенностях данного списка. Функция printf() использует имена переменных, константы и выражения, в то время как функция scanf() — только указатели на переменные. К счастью, при применении этой функции мы ничего не должны знать о таких указателях. Необходимо помнить только два правила:

1. Если вам нужно ввести некоторое значение и присвоить его переменной одного из основных типов, то перед именем переменной требуется писать символ &.
2. Если вы хотите ввести значение строковой переменной, использовать символ & не нужно.

Приведем правильную программу:

main()

{

int age;

float assets;

char pet [30];

printf("Укажите ваш возраст, состояние и любимое животное.\n");

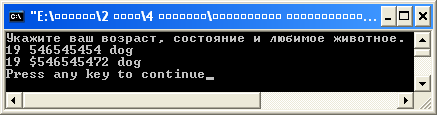
scanf("%d %f", &age, &assets);

scanf("%s", pet); /\* & отсутствует при указании массива символов \*/

printf("%d $%.0f %s\n" , age, assets, pet);

}

Вот пример диалога:



Функция scanf() использует некоторые специальные знаки (пробелы, символы табуляции и «новая строка») для разбиения входного потока символов на отдельные поля. Она согласует последовательность спецификаций преобразования с последовательностью полей, опуская упомянутые специальные знаки между ними. Обратите внимание, что наша входная информация располагается на двух строках. Точно так же мы могли бы использовать одну или пять строк при условии, что вводимые величины разделяются по крайней мере одним знаком типа «новой строки», пробела или символа табуляции. Единственным исключением из этого является спецификация %с, обеспечивающая чтение каждого следующего символа даже в том случае, если это «пустой символ».

Функция scanf() использует практически тот же набор символов спецификации преобразования, что и функция printf(). Основные отличия в случае функции scanf() следующие:

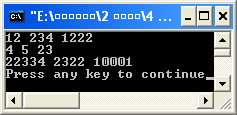
1. Отсутствует спецификация %g.
2. Спецификации %f и %е эквивалентны. Обе спецификации допускают наличие (или отсутствие) знака, строки цифр с десятичной точкой или без нее и поля показателя степени.
3. Для чтения целых чисел типа short применяется спецификация %h.

Функция scanf() не является одной из наиболее часто используемых функций языка Си. Мы обсуждаем ее здесь главным образом из-за ее универсальности (она позволяет читать данные всех имеющихся типов); однако в Си имеется еще несколько других функций, осуществляющих ввод, например getchar() и gets(), которые более удобны для выполнения конкретных задач — чтения одиночных символов или строк, содержащих пробелы.

Советы по применению. Задание фиксированной ширины полей оказывается полезным при печати данных столбцами. Поскольку шириной поля по умолчанию является «ширина» числа, при повторном использовании оператора

printf("%d %d %d\n", val1, val2, val3);

будут получены неровные столбцы чисел, если эти числа состоят из разного количества цифр. Например, результат мог бы выглядеть следующим образом:

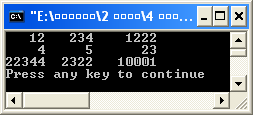


(Здесь предполагается, что между обращениями к оператору печати значения переменных изменялись.)

Эти же данные можно представить в улучшенном виде, если задать достаточно большую фиксированную ширину поля. При использовании оператора

printf("%9d %9d %9d\n", val1, val2, val3);

результат будет выглядеть так:

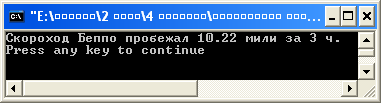


Наличие пробелов между спецификациями преобразования гарантирует, что даже в том случае, если все поле будет заполнено, символы, соответствующие данному числу, не перейдут в следующее поле. Это вызвано тем обстоятельством, что обычные символы, имеющиеся в управляющей строке, включая пробелы, всегда печатаются.

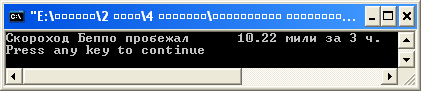
С другой стороны, если печатаемое число включено в некоторую фразу, то часто при его выводе оказывается удобным задать ширину поля равной или меньше требуемой Это дает возможность включить число в фразу без добавления лишних пробелов. Например, результатом работы оператора

printf("Скороход Беппо пробежал %.2f мили за 3 ч.\n", distance);

могла бы быть следующая фраза:



Изменяя спецификацию преобразования на %10.2f, получим



**Что вы должны были узнать.**

* Что такое строка символов: несколько символов, расположенных в ряд
* Как записывать строку символов: " несколько символов, расположенных в ряд"
* Как строка хранится в памяти: " несколько символов, расположенных в ряд\0"
* Где разместить строку: char phrase [25] или static char phrase [25]
* Как определить длину строки: использовать функцию strlen (строка)
* Как напечатать строку: printf(" %s", phrase) Как прочитать строку, состоящую из одного слова-scanf(" %s" ,&name)
* Как задать числовую константу: #define TWO 2 Как задать символьную константу: #define WOW !' Как задать строковую константу: #define WARN "He делай этого!"
* Спецификации преобразования при вводе-выводе: %d %f %e %g %c %s %u %o %x
* Как улучшить вид входной информации: %-10d %3.2f Как выполнять преобразования: printf(" %d %o %c\h", WOW, WOW, WOW);

**Вопросы и ответы.**

**Вопросы.**

1. Выполните снова программу, приведенную в начале, но на этот раз в ответ на вопрос о вашем имени введите имя и фамилию. Что произойдет? Почему?
2. Что выведет на печать каждый из нижеприведенных программных фрагментов в предположении, что они являются частью некоторой полной программы?
3. printf("Oн продал картину за $%2.2f.\n", 2.345e2);
4. printf("%c%c%c\n", 'H', 105, '\41');
5. #define Q "Его Гамлет был смешным, но не вульгарным.". printf("%s\n имеет %d символов.\n", Q, strlen(Q));
6. printf("%2.2e то же самое, что и %2.2f?\n", 1201.0, 1201.0);
7. Какие изменения необходимо внести в программу п. 2в, чтобы строка Q была выведена на печать заключенной в апострофы?
8. Очередная задача по обнаружению ошибок в программе.

define В а-яй-яй

define X 10

main()

{

int age;

char name;

printf(" Укажите, пожалуйста, свое имя.");

scanf(" %s" , name);

printf(" Прекрасно, %с, сколько вам лет?\n", name);

scanf(" %f", age);

xp = age + X;

printf(" %s! Вам должно быть по крайней мере %d.\n", В, хр);

}

**Ответы.**

1. «Взрывоопасная» программа. Первый оператор scanf() читает ваше имя, оставляя фамилию непрочитанной; при этом она все-таки попадает во входной «буфер». (Этот буфер выполняет функции области памяти, используемой для временного хранения поступающих данных.) Следующий оператор scanf() должен ввести в программу величину вашего веса; он начинает вводить символы как раз с того места, где завершился предыдущий ввод, и поэтому читает вашу фамилию, принимая ее за вес. В результате в программу попадает «мусор». С другой стороны, если вы в ответ на вопрос об имени введете строку типа «Саша 144», то величина 144 будет рассматриваться как ваш вес, несмотря на то что вы ввели ее до того, как программа запросила величину веса.
2. Он продал картину за 234.50 долл.
3. Hi!

Примечание: первый символ — это символическая константа, второй — десятичное целое число, преобразованное в символ, а третий — представление символической константы в коде ASCII.

1. Его Гамлет был смешным, но не вульгарным, имеет 41 символ.
2. 1.20Е+03 то же самое, что и 1201.00?
3. Вспомните, что раньше говорилось по поводу управляющих последовательностей, и попробуйте записать оператор в таком виде:

printf(" \" %s\" \n имеет %d символов.\n", Q, strlen(Q));

* Строка 1: символ # опущен; вместо а-яй-яй должно стоять " а-яй-яй".
* Строка 2: символ # опущен.
* Строка 6: переменная name должна быть массивом, например char name [25].
* Строка 8: в управляющей строке должен стоять символ \n.
* Строка 10: вместо %с должно быть %s.
* Строка 11: поскольку переменная age целого типа, необходимо использовать %d, а не %f; кроме того, вместо age должно стоять &аgе.
* Строка 12: имя хр нигде не было описано.
* Строка 13: правильная, но при выводе на печать результат будет испорчен из-за ошибки, допущенной при определении В.

Кроме того, программа служит примером плохого стиля программирования.