Оглавление

[1. Использование функций. Создание и использование простой функции. Прототипы функций. Вызов по значению и вызов по ссылке. 1](#_Toc104735260)

[2. Использование функций. Использование указателей для связи между функциями. Рекурсия. Равноправность функций в языке Си. 2](#_Toc104735261)

[3. Использование функций. Параметры и аргументы функций. Формальные и фактические параметры. Аргумент типа void. Символьные параметры. Целочисленные параметры. Параметры в формате чисел с плавающей точкой. Параметры в формате чисел двойной длины. Массивы в качестве параметров. Аргументы по умолчанию. 3](#_Toc104735262)

[4. Использование функций. Возвращение значения функцией: оператор return. Типы функций. Функции типа void. Функции типа char. Функции типа int. Функции типа long. Функции типа float. Функции типа double. 6](#_Toc104735263)

[5. Использование функций. Важные возможности C++. Встраивание (inline). Перегрузка (overloading). 6](#_Toc104735264)

[6. Использование функций. Функции с переменным числом параметров. Многоточие (...). Задание числа дополнительных параметров с помощью первого параметра. Определение конца списка параметров с помощью параметра индикатора. Использование специального набора макроопределений. Список указателей переменной длины на char (конкатинация строк). Изменение параметров по числу и по типу. Дополнительные примеры функций с произвольным числом параметров. 7](#_Toc104735265)

[7. Использование функций. Аргументы функции main(). Строки. Целые числа. Числа с плавающей точкой. 9](#_Toc104735266)

[8. Использование функций. Области видимости. Локальные и глобальные переменные. Сложности в правилах области действия (scope rules). Неопределенные символы в программе на С. Использование переменной с файловой областью действия. Приоритет переменных с файловой и локальной областями действия. Проблемы области действия в C++. Операция уточнения области действия в C++. 10](#_Toc104735267)

[Локальные объекты 10](#_Toc104735268)

[9. Использование функций. Математические функции. 11](#_Toc104735269)

[10. Использование функций. Указатель на функцию. Указатели на функции. Указатели на методы. Указатель на функцию и динамическое связывание. Таблицы функций, вызов по имени. Указатель на функцию как средство параметризации алгоритма. 12](#_Toc104735270)

[11. Использование функций. Массив указателей на функции. Шаблоны функций в С++. Основные понятия. Параметры шаблонов функций. 13](#_Toc104735271)

[12. Использование функций. Шаблоны функций. Аргументы по умолчанию. Функции округления. Компиляция программ, состоящих из двух или более функций. 13](#_Toc104735272)

[13. Классы памяти. "Зоопарк" классов памяти. Автоматические переменные. Регистровые переменные. Статические переменные. Внешние переменные. Внешние статические переменные. 13](#_Toc104735273)

[14. Классы памяти. Объявление переменных на внутреннем уровне. Объявление переменных на внешнем уровне. 13](#_Toc104735274)

[15. Классы памяти. Переменные класса volatile. Ключевое слово mutable. Классы памяти и область действия. Правила области действия переменной. Операция уточнения области действия в C++. Выбор класса памяти. 13](#_Toc104735275)

[16. Классы памяти. Пространства имен. Функции и классы памяти. Объявления функций на внешнем уровне. 13](#_Toc104735276)

[17. Дополнительные приемы программирования. Совместимость типов. Определение совместимости типов в ANSI С. 13](#_Toc104735277)

[18. Дополнительные приемы программирования. Идентичный тип. Перечисляемые типы. Типы массивов. Типы функций. Типы структур и объединений. Типы указателей. Совместимость нескольких исходных файлов. 13](#_Toc104735278)

[19. Дополнительные приемы программирования. Макроопределения. Определение макросов. Макросы и параметры. Сложности при раскрытии макросов. Создание и использование собственных макросов. Макросы, поставляемые вместе с компилятором. Выбор макроопределения или функции. 13](#_Toc104735279)

[20. Дополнительные приемы программирования. Директивы препроцессора. Директива #define. Директива #include. Заголовочные файлы. Правильное использование заголовочных файлов. Более эффективное использование заголовочных файлов. Новый стиль заголовков. Предварительная компиляция заголовочных файлов. Файлы limits.h и float.h. 13](#_Toc104735280)

[21. Дополнительные приемы программирования. Директивы #ifdef и #endif. Директива #undef. Директива #ifndef. Директива #if. Директива #else. Директива #elif. 13](#_Toc104735281)

[22. Дополнительные приемы программирования. Директива #line. Директива #error. Директива #pragma. 13](#_Toc104735282)

[23. Дополнительные приемы программирования. Оператор defined. Условная компиляция. 13](#_Toc104735283)

[24. Дополнительные приемы программирования. Дополнительные операции препроцессора. Операция подстановки строки (#). Операция конкатенации (##). Операция подстановки символа (#@). 13](#_Toc104735284)

[25. Дополнительные приемы программирования. Обработка ошибок: perror(). Модели памяти. Модель tiny. Модель small. Модель medium. Модель compact. Модель large. Модель huge. 14](#_Toc104735285)

[26. Дополнительные приемы программирования. Модификаторы функций. Модификаторы cdecl и pascal. 14](#_Toc104735286)

[27. Файлы в C. Файлы и потоки. Связь с файлами. Закрытие потоков. Обработка ошибок в C и C++. Переменная errno и коды ошибок. 14](#_Toc104735287)

[28. Файлы в C. Переключение и работа с файлами. Переключение вывода. Переключение ввода. Комбинированное переключение. Конвейерная пересылка. Текстовые и бинарные (двоичные) файлы. 14](#_Toc104735288)

[29. Файлы в C. Потоковый ввод-вывод. Соединение и отсоединение потока от файла. Функция fopen. Функция fclose. Функция freopen. 14](#_Toc104735289)

[30. Файлы в C. Работа с индикаторами ошибки, позиции и конца файла (ferror, clearerr, feof, rewind, fseek, fsetpos, ftell, fgetpos). Блочный ввод-вывод (fwrite, fread). 14](#_Toc104735290)

[31. Файлы в C. Символьный ввод-вывод. Функции fputc, putc, fgetc, getc, ungetc. Функции fputs и fgets. Функции fprintf, fscanf. 14](#_Toc104735291)

[32. Файлы в C. Работа с буферами (setvbuf, setbuf, fflush). Стандартные потоки (putchar, getchar, puts, gets, perror). Служебные функции для работы с файлами (remove, rename, tmpfile, tmpnam). 14](#_Toc104735292)

[33. Файлы в C. Низкоуровневый ввод и вывод в С. Форматированный вывод. Использование функций printf() и fprintf(). 14](#_Toc104735293)

[34. Файлы в C++. Структура стандартной библиотеки ввода-вывода. Предопределенные потоки. От файла STREAM.H к файлу IOSTREAM.H. Список классов iostream. 14](#_Toc104735294)

[35. Файлы в C++. Простота ввода/вывода в С ++. Потоки cin, cout и сегг. Операции выделения ( >> ) и вставки ( << ). Опции ввода/вывода в С/С++. 14](#_Toc104735295)

[36. Файлы в C++. Операции и методы классов. Вывод символов в C++. Преобразование системы счисления в C++. Форматирование строк в C++. Форматирование чисел в C++. Файловый ввод и вывод в C++. 14](#_Toc104735296)

[37. Файлы в C++. Классы ios\_base и ios. Потоки вывода. Функции open, close и is\_open. Функция flush. Функция put. Функция seekp и tellp. Функция write. Функция str. Классы потокового вывода. 14](#_Toc104735297)

[38. Файлы в C++. Потоки ввода. Функция gcount. Функция get. Функция getline. Функция ignore. Функция peek. Функция putback. Функция read. Функция seekg. Функция sync. Функция unget. Функция str. Классы потокового ввода. 14](#_Toc104735298)

[39. Файлы в C++. Потоки ввода-вывода. Классы буферизированных потоков. Класс строковых потоков. Форматирование потока. Резидентные в памяти потоки. Буферы и синхронизация. 14](#_Toc104735299)

[40. Файлы в C++. Условные признаки файлов в C++. Опрос и установка состояния потока. Ошибки потоков. Часто применяемые функции. Двоичные файлы. Объединение программ на С и С++. Использование спецификатора extern "С". 15](#_Toc104735300)

[41. Файлы в C++. Манипуляторы. Манипуляторы без параметров. Манипуляторы с одним параметром. Манипуляторы с несколькими параметрами. 15](#_Toc104735301)

[42. Функции для работы с файлами. Полезные функции для работы с файлами. clearerr(). fclose(). fcloseall(). fdopen(). feof(). ferror(). fflush(). fgetc(). fgetchar(). fgetpos(). 15](#_Toc104735302)

[43. Функции для работы с файлами. fgets(). filelength(). fileno(). flushall() (fflush()). fopen(). fprintf(). fputc(). fputchar(). fputs(). fread(). 15](#_Toc104735303)

[44. Функции для работы с файлами. freopen(). fscanf(). fseek(). fsetpos(). fstat(). ftell(). fwrite(). getc(). getchar(). gets(). getw(). 15](#_Toc104735304)

[45. Функции для работы с файлами. perror(). printf(). putc(). putchar(). puts(). putw(). remove(). rename(). rewind(). scanf(). setbuf(). 15](#_Toc104735305)

[46. Функции для работы с файлами. setvbuf(). sprintf(). sscanf(). tmpfile(). tmpnam(). ungetc(). vfprintf(). vfscanf(). vprintf(). vsprintf(). vsscanf(). 15](#_Toc104735306)

[47. Функции для работы с файлами. Низкоуровневый ввод и вывод. close(). lseek(). open(). read(). unlink() (\_unlink()). write(). 15](#_Toc104735307)

[48. Структуры и другие типы данных. Структуры С и C++. Структуры С и C++: синтаксис и правила. Структуры C++: расширения синтаксиса и правил. Доступ к элементам структуры. Инициализация структуры. Передача структур в функции. 15](#_Toc104735308)

[49. Структуры и другие типы данных. Массив структур. Описание массива структур. Определение элементов массива структур. Вложенные структуры. 15](#_Toc104735309)

[50. Структуры и другие типы данных. Использование указателей на структуры. Описание и инициализация указателя на структуру. Доступ к элементу структуры при помощи указателя. 15](#_Toc104735310)

## Использование функций. Создание и использование простой функции. Прототипы функций. Вызов по значению и вызов по ссылке.

1. #include<stdio.h>
2. #define NAME "MEGATHINK, INC."
3. #define ADDRESS "10 Megabuck Plaza"
4. #define PLACE "Megapolis, CA 94904"
5. void starbar();
6. void main()
7. {
8. starbar();
9. printf("%s\n", NAME);
10. printf("%s\n", ADDRESS);
11. printf("%s\n", PLACE);
12. starbar();
13. }
14. /\* далее следует функция starbar() \*/
15. #define LIMIT 65
16. void starbar()
17. {
18. int count;
19. for (count=1; count<=LIMIT; count++)
20. putchar('\*');
21. putchar('\n');
22. }

***тип* имя\_функции(*список\_параметров или void*)**

**{**

**тело\_функции**

**[return] (выражение);**

**}**

где

* ***тип***– тип значения, возвращаемого функцией. Если поле «*тип*» содержит ключевое слово void, то функция не возвращает никакого значения;
* *имя\_функции* – это непосредственно имя функции. По этому имени осуществляется вызов программного кода, реализованного в *теле\_функции*. Кроме того, *имя\_функции* есть [**указателем на эту функцию**](https://www.bestprog.net/ru/2017/03/21/%d1%83%d0%ba%d0%b0%d0%b7%d0%b0%d1%82%d0%b5%d0%bb%d0%b8-%d1%87%d0%b0%d1%81%d1%82%d1%8c-1-%d0%be%d0%b1%d1%89%d0%b8%d0%b5-%d0%bf%d0%be%d0%bd%d1%8f%d1%82%d0%b8%d1%8f-%d1%82%d0%b8%d0%bf%d1%8b-%d1%83/). Значением указателя есть адрес точки входа в функцию;
* *список\_параметров* – параметры, которые передаются в функцию. Функция может получать любое число параметров. Если описывается функция без параметров, то в скобках указывается ключевое слово void;
* *тело\_функции* – набор операторов программного кода, реализующих алгоритм вычисления внутри функции;
* *return (выражение)* – ключевое слово return указывает, что функция возвращает значение заданное в *(выражение)*. Слово return может встречаться в нескольких местах тела функции в зависимости от алгоритма (повторяться).
* передача параметра**по значению** (Call-By-Value). Это простая передача копий переменных в функцию. В этом случае изменение значений параметров в теле функции не изменит значений, которые передавались в функцию извне (при ее вызове);
* передача параметра **по адресу переменной**. В этом случае функции в качестве параметров передаются не копии переменных, а копии адресов переменных, то есть указатель на переменную. Используя этот указатель функция осуществляет доступ к нужным ячейкам памяти где расположена передаваемая переменная и может изменять ее значение. Общие сведения об указателях приведены [**здесь**](https://www.bestprog.net/ru/2017/03/21/%d1%83%d0%ba%d0%b0%d0%b7%d0%b0%d1%82%d0%b5%d0%bb%d0%b8-%d1%87%d0%b0%d1%81%d1%82%d1%8c-1-%d0%be%d0%b1%d1%89%d0%b8%d0%b5-%d0%bf%d0%be%d0%bd%d1%8f%d1%82%d0%b8%d1%8f-%d1%82%d0%b8%d0%bf%d1%8b-%d1%83/);
* передача параметра **по ссылке** (Call-By-Reference). Передается ссылка (указатель) на объект (переменную), что позволяет синтаксически использовать эту ссылку как указатель, и как значение. Изменения, внесенные в параметр, который передается по ссылке, изменяют исходную копию параметра вызывающей функции.

*Прототип функции* – это соообщение компилятору и другим функциям о том, что такая функция существует. Иными словами, это сообщение компилятору и другим функциям о том, что в программе существует функция с указанной сигнатурой. Прототип функции состоит из сокращенного описания без тела функции.

Прототип функции содержит:

* имя функции;
* параметры функции или типы этих параметров;
* тип значения, возвращаемого функцией.

Общая форма объявления прототипа функции следующая:

return\_type FuncName(parameters);

## Использование функций. Использование указателей для связи между функциями. Рекурсия. Равноправность функций в языке Си.

В функцию можно передавать не только значения переменных, но и их адреса. В этом случае в вызываемой функции можно изменять значение локальных переменных, определенных в вызывающей функции.

void swap(int\*,int\*);

void main(void){

int x=5, y=10;

printf (Прервичные значения х=%d, y=%d\n,x, y);

swap (x, y);

printf (Новые значения х=%d, y=%d\n, x, y);

}

void swap (int\*v, int\*z){

int u;

u=\*v;

\*v=\*z; //x=y

\*z=u;

}

Мы передали в функцию адреса переменных, поэтому при выходе х, у имееют новые значения. При передаче значений этого не происходило.

Вызов swap (x, y); swap (x, y);

Определение функции swap (int v, int z); swap (int\*v, int\*z);

Переменные х, y являются локальными в функции main, но мы, таким образом, можем на них воздействовать в другой функции.

**Рекурсивная функция** (или просто ***«рекурсия»***) в языке C++ — это функция, которая вызывает сама себя. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <iostream>    void countOut(int count)  {      std::cout << "push " << count << '\n';      countOut(count-1); // функция countOut() вызывает рекурсивно сама себя  }    int main()  {      countOut(4);        return 0;  } |

При выполнении рекурсии программа сохраняет в стеке значения всех локальных переменных функции и ее аргументов с тем, чтобы в дальнейшем по возвращении из рекурсивного вызова восстановить их сохраненные значения

Все функции в программе, написанной на языке Си, равноправны: каждая из них может вызывать любую другую функцию и в свою очередь каждая может быть вызвана любой другой функцией.

Нет ли у функции main() какой-то специфики? Безусловно, есть. Она заключается в том, что после «сборки» программы, состоящей из нескольких функций, ее выполнение начинается с первого оператора функции main(). Но этим ее исключительность и ограничивается. Даже функция main() может быть вызвана другими функциями

## Использование функций. Параметры и аргументы функций. Формальные и фактические параметры. Аргумент типа void. Символьные параметры. Целочисленные параметры. Параметры в формате чисел с плавающей точкой. Параметры в формате чисел двойной длины. Массивы в качестве параметров. Аргументы по умолчанию.

**Параметр функции**— это переменная, которая используется в функции, и значение которой предоставляет caller (вызывающий объект). Параметры указываются при объявлении функции в круглых скобках. Если их много, то они перечисляются через запятую, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // Эта функция не имеет параметров  void doPrint()  {     std::cout << "In doPrint()" << std::endl;  }    // Эта функция имеет один параметр типа int: a  void printValue(int a)  {     std::cout << a << std::endl;  } |

Параметры каждой функции действительны только внутри этой функции. Поэтому, если printValue() и add() имеют параметр с именем a, то это не означает, что произойдет конфликт имен. Эти параметры считаются независимыми и никак не взаимодействуют друг с другом.

**Аргумент функции** — это значение, которое передается из caller-а в функцию и которое указывается в скобках при вызове функции в caller-е:

|  |  |
| --- | --- |
| 1  2 | printValue(7); // 7 – это аргумент функции printValue()  add(4, 5); // 4 и 5 – это аргументы функции add() |

Обратите внимание, аргументы тоже перечисляются через запятую. Количество аргументов должно совпадать с количеством параметров, иначе компилятор выдаст сообщение об ошибке.

Формальные параметры – это переменные, принимающие значение аргументов (параметров) функции. Если функция имеет несколько аргументов (параметров), их тип и имена разделяются запятой ‘ , ‘.

При вызове функции с параметрами, компилятор осуществляет копирование копий формальных параметров в стек.

**Пример.** Функция MyAbs(), находящая модуль числа имеет один формальный параметр x.

// функция, которая находит модуль вещественного числа

float MyAbs(float x) // x - формальный параметр

{

if (x<0) return (float)(-x);

else return x;

}

Вызов функции из другого программного кода (другой функции)

// вызов функции из другого программного кода

float res, a;

a = -18.25f;

res = MyAbs(a); // res = 18.25f; переменная a - фактический параметр

res = MyAbs(-23); // res = 23; константа 23 - фактический параметр

При вызове функции из другого программного кода имеет место фактический параметр. В данном примере фактический параметр это переменная a и константа 23.

При вызове функции фактические параметры копируются в специальные ячейки памяти в стеке (стек – часть памяти). Эти ячейки памяти отведены для формальных параметров. Таким образом, формальные параметры (через использование стека) получают значение фактических параметров.

Поскольку, фактические параметры копируются в стек, то изменение значений формальных параметров в теле функции не изменит значений фактических параметров (так как это есть копия фактических параметров).

В соответствии с ANSI С, отсутствие списка аргументов функции должно быть указано явно при помощи ключевого слова void. В C++ использование void пока не обязательно, но считается целесообразным

Функции можно передавать символьные значения. В следующем примере в функции main() одиночный символ считывается с клавиатуры и передается функции voutput(). Символ считывается функцией getch().

void voutput(char c);

main()

{

char cyourchar;

/\* Введите один символ с клавиатуры \*/

printf("Enter one character from the keyboard. \n");

cyourchar=getch();

voutput(cyourchar);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

В следующем примере одно целое число вводится с клавиатуры при помощи функции С scanf() и передается функции vside()

main()

{

int iyourlength=0;

/\* Введите с клавиатуры длину как целое число \*/

printf("Enter the length, as an integer from the keyboard. \n");

scanf("%d",&iyourlength);

vside(iyourlength);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

Числа с плавающей точкой так же легко передавать в качестве параметров функции, как и целые значения.

Тип чисел двойной длины double обеспечивает очень большую точность чисел с плавающей точкой.

Если функция принимает в качестве параметра массив, то фактически в эту функцию передается указатель на первый элемент массива. То есть как и в случае с указателями нам доступен адрес, по которому мы можем менять значения. Поэтому следующие объявления функции будут по сути равноценны:

void print(int numbers[]);

void print(int \*numbers);

**Параметр по умолчанию** (или ***«необязательный параметр»***) — это параметр функции, который имеет определенное (по умолчанию) значение. Если пользователь не передает в функцию значение для параметра, то используется значение по умолчанию. Если же пользователь передает значение, то это значение используется вместо значения по умолчанию. Например:

|  |  |
| --- | --- |
|  | void printValues(int a, int b=5) |

## Использование функций. Возвращение значения функцией: оператор return. Типы функций. Функции типа void. Функции типа char. Функции типа int. Функции типа long. Функции типа float. Функции типа double.

Для возвращения результата из функции применяется оператор **return**. Этот оператор имеет две формы:

|  |  |
| --- | --- |
| 1  2 | return;  return выражение; |

Первая форма используется, если в качестве возвращаемого типа функции применяется тип **void**.

Если функция имеет в качестве возвращаемого типа любой тип, отличный от void, то такая функция обязятельно должна возвратить некоторое значение с помощью оператора return. Причем значение, которое возвращается оператором return, должно соответствовать возвращаемому типу функции, либо допускать неявное преобразование в этот тип.

Тип функции определяется типом возвращаемого ею значения, а не типом ее аргументов. Если указание типа отсутствует, то по умолчанию считается, что функция имеет тип int. Если значения функции не принадлежат типу int, то необходимо указать ее тип в двух местах.

* Описать тип функции в ее определении:

char pun(ch, n) /\* функция возвращает символ \*/

int n;

char ch;

float raft(num) /\* функция возвращает величину типа float \*/

int num;

* Описать тип функции также в вызывающей программе. Описание функции должно быть приведено наряду с описаниями переменных программы; необходимо только указать скобки (но не аргументы) для идентификации данного объекта как функции.

## Использование функций. Важные возможности C++. Встраивание (inline). Перегрузка (overloading).

Одним из его достоинств является возможность написания встроенных функций. Код некоторой встроенной функции воспроизводится в том месте программы, в котором она вызывается. Поскольку компилятор располагает этот код в точке вызова функции, то при использовании коротких, часто вызываемых функций, сокращается время выполнения программы.

Помимо этого, в C++ функции можно перегружать. Перегрузка позволяет использовать одно и то же имя функции для нескольких ее прототипов. Затем различные прототипы распознаются не по имени, а по типу и списку аргументов.

если тело функции небольшого размера и обращение к ней в программе происходит довольно часто, на практике можно указать компилятору вместо вызовов функции в соответствующих местах генерировать все ее тело. Осуществляется это с помощью ключевого слова inline.

Ключевое слово inline можно рассматривать как директиву или, лучше сказать, как указание компилятору C++ на встраивание этой функции. По разным причинам компилятор может проигнорировать это указание. Например, функция может быть слишком длинной.

Ключевое слово inline указывает компилятору заменить код в определении функции для каждого экземпляра вызова функции. Применение встраиваемых функций может ускорить выполнение программ, поскольку устраняет нагрузку на вызов функций. Компилятор может оптимизировать функции, развернутые встроенными способами, которые недоступны обычным функциям. Подстановка встроенного кода выполняется по усмотрению компилятора. Например, компилятор не будет встраивать функцию, если используется его адрес или слишком большой для встроенной функции.

**Перегрузка функций** — это возможность определять несколько функций с одним и тем же именем, но с разными параметрами.

int subtract(int a, int b); // целочисленная версия

double subtract(double a, double b); // версия типа с плавающей запятой

## Использование функций. Функции с переменным числом параметров. Многоточие (...). Задание числа дополнительных параметров с помощью первого параметра. Определение конца списка параметров с помощью параметра индикатора. Использование специального набора макроопределений. Список указателей переменной длины на char (конкатинация строк). Изменение параметров по числу и по типу. Дополнительные примеры функций с произвольным числом параметров.

Язык программирования Си допускает использование функций, которые имеют нефиксированное количество параметров. Более того может быть неизвестным не только количество, но и типы параметров. То есть точное определение параметров становится известным только во время вызова функции.

Для определения параметров неопределенной длины в таких функциях используется многоточие:

|  |  |
| --- | --- |
| 1 | тип имя\_функции(обязательные параметры, ...) |

При этом надо учитывать, что функция должна иметь как минимум один обязательный параметр.

каждая функция с переменным списком параметров должна иметь механизм определения их количества и их типов. Принципиально различных подходов к созданию этого механизма всего два. Первый подход предполагает добавление в конец списка реально использованных необязательных фактических параметров специального параметра-индикатора с уникальным значением, которое будет сигнализировать об окончании списка. В теле функции параметры последовательно перебираются, и их значения сравниваются с заранее известным концевым признаком. Второй подход предусматривает передачу в функцию значения реального количества фактических параметров. Значение реального количества используемых фактических параметров можно передавать в функцию с помощью одного из явно задаваемых (обязательных) параметров.

Для доступа к параметрам, которые один за другим попали в стек, используется адрес первого параметра **(\*p = &n;)**, таким образом, указатель устанавливается на начало списка параметров в стеке (в памяти). Затем в цикле при помощи указателя **p** перемещаемся по параметрам и суммируем их, извлекая **(\*p)** из памяти. Все параметры должны иметь **одинаковый тип**.

Для использования в программе функций с переменным числом параметров, при вызове которых можно использовать параметры разных типов, удобно использовать специальный набор макроопределений (заголовочный файл **stdarg.h**). Макрокоманды для доступа к списку фактических параметров переменной длины, имеют следующий формат:

//связывание переменной param с первым параметром

**void va\_start(va\_list param,<последний явный параметр>);**

//получение значения очередного параметра типа type

**type va\_arg(va\_list param, type);**

//организации корректного выхода из функции

**void va\_end(va\_list param);**

Они объявлены в загаловке <stdarg.h>

*Строковая константа*, написанная в виде

"Я строка"

есть массив символов. Во внутреннем представлении этот массив заканчивается нулевым символом '\0', по которому программа может найти конец строки. Число занятых ячеек памяти на одну больше, чем количество символов, помещенных между двойными кавычками.

Чаще всего строковые константы используются в качестве аргументов функций, как, например, в

printf("здравствуй, мир\n");

Когда такая символьная строка появляется в программе, доступ к ней осуществляется через символьный указатель; *printf* получает указатель на начало массива символов. Точнее, доступ к строковой константе осуществляется через указатель на ее первый элемент.

Строковые константы нужны не только в качестве аргументов функций. Если, например, переменную *pmessage* объявить как

char \*pmessage;

то присваивание

pmessage = "now is the time";

поместит в нее указатель на символьный массив, при этом сама строка не копируется, копируется лишь указатель на нее. Операции для работы со строкой как с единым целым в Си не предусмотрены.

Для изменения значения параметра функция должна знать адрес памяти параметра. Чтобы сообщить функции адрес параметра, ваши программы должны использовать оператор адреса C++ (&). Следующий вызов функции иллюстрирует, как программа будет использовать оператор адреса для передачи адресов переменных *big* и *small* в функцию *change\_values:*

change\_values (&big, &small); ***-->Передача параметров по адресу***

Внутри функции вы должны сообщить C++ , что программа будет передавать параметры с помощью адреса. Для этого вы объявляете *переменные-указатели,*предваряя имя каждой переменной звездочкой, как показано ниже:

void сhange\_values (int. \*big, int. \*small)***---> Указатель на тип int***

## Использование функций. Аргументы функции main(). Строки. Целые числа. Числа с плавающей точкой.

В языке C заданы два встроенных аргумента функции main: argc и argv.

Выглядит это так:

int main(int argc, char \*argv[]) {...}

Аргумент argc типа integer содержит в себе количество аргументов командной строки.

Аргумент argv типа char - указатель на массив строк. Каждый элемент массива указывает на аргументы командной строки. Один параметр отделяется от другого пробелами.

* argv[0] - полное имя запущенной программы
* argv[1] - первая строка записаная после имени программы
* argv[2] - вторая строка записаная после имени программы
* argv[argc-1] - последняя строка записаная после имени программы
* argv[argc] - NULL
* В средах разработки фирмы Borland (Borland Builder C++, Borland C++ и т.п.) предусмотрен еще и третий аргумент env, который, так же как и argv является указателем на массив строк, но содержит установки среды:
* int main(int argc, char \*argv[], char \*env[]) {...}

Аргумент командной строки — это информация, которая вводится в командной строке операционной системы вслед за именем программы.

Аргументы командной строки передаются как символьные строки, что облегчает работу с ними.

Если из командной строки вводятся цифровые значения, то они интерпретируются как ASCII-строки отдельных символов и должны печататься такими как есть.

Во многих программах желательно иметь возможность ввода из командной строки целых чисел; это может быть, к примеру, программа для вычисления средних оценок студентов. В таких случаях символьная информация в кодах ASCII должна быть преобразована в целые значения. Если из командной строки вводится одно целое число. Поскольку на самом деле это число является символьной строкой, оно преобразуется в целое при помощи библиотечной функции atoi().

Поскольку вы знаете, как вводить целые числа из командной строки, числа float не создают дополнительных проблем.

## Использование функций. Области видимости. Локальные и глобальные переменные. Сложности в правилах области действия (scope rules). Неопределенные символы в программе на С. Использование переменной с файловой областью действия. Приоритет переменных с файловой и локальной областями действия. Проблемы области действия в C++. Операция уточнения области действия в C++.

Область видимости (scope) представляет часть программы, в пределах которой можно использовать объект. Как правило, область видимости ограничивается блоком кода, который заключается в фигурные скобки В зависимости от области видимости создаваемые объекты могут быть глобальными, локальными или автоматическими.

Глобальные переменные определены в файле программы вне любой из функций и могут использоваться любой функцией.

### Локальные объекты

Объекты, которые создаются внутри блока кода (он может представлять функцию или какую-либо конструкцию типа циклов), называются локальными. Такие объекты доступны в пределах только того блока кода, в котором они определены.

#### Автоматические объекты

Локальные объекты, которые существуют только во время выполнения того блока, в котором они определены, являются **автоматическими**.

При входе в блок для подобных переменных выделяется память, а после завершения работы этого блока, выделенная память освобождается, а объекты удаляются.

**Если используются переменные с различной областью действия, то можно столкнуться с совершенно неожиданными результатами программирования, называемыми побочными эффектами.** Например, как вы уже знаете, может существовать переменная (вернее две переменные с одинаковым именем) как с файловой, так и с локальной областью действия. **Правила области действия констатируют, что переменная с локальной областью действия (называемая локальной переменной) имеет приоритет по сравнению с переменной с файловой областью действия (называемой глобальной переменной).**

В следующем примере четыре переменные имеют локальную область действия внутри функции main(). Копии переменных il и im передаются в функцию iproduct(). Это не нарушает правила области действия. Однако, когда функция iproduct() пытается обратиться к переменной in, она ее не находит. Почему? Потому что область действия этой переменной локальна для функции main().

В следующем примере переменной in определена файловая область действия. Глобальная видимость переменной in во всем файле позволяет ее использовать как функции main(), так и функции iproduct(). Также надо заметить, что обе функции, main() и iproduct(), могут изменять значение переменной. Существует хорошее программистское правило: если функции должны быть действительно переносимыми, не следует позволять им менять глобальные переменные.

**Правила области действия констатируют, что у переменной, имеющей как локальную, так и файловую область действия, используется ее локальное, а не глобальное значение.**

Операция разрешения области действия обозначается двумя символами двоеточие без пробела – “::” Узнали? Рассмотрим ее назначение.

Она бывает двух типов: унарная и бинарная.  
Унарная операция разрешения области действия позволяет получить доступ к глобальной переменной (область видимости – вся программа) из блока, в котором может быть объявлена локальная переменная (область видимости начинается ее объявлением и заканчивается закрытием текущего блока) с таким же именем.  
Пример:

1.int a; //глобальная переменная

2.void func1(){

3.int a; //локальная переменная

4.a=10; //обращение к локальной переменной

5.::a=100; //обращение к глобальной переменной

6.}

Бинарная операция разрешения области действия позволяет ссылаться на элемент класса при наличии одноименных переменных или функций, объявленных вне его или в нескольких классах. Также данная операция используется при описании функции-метода вне класса.  
Операция разрешения области действия используется в следующих случаях:

1) Локальная переменная, объявленная в теле функции, имеет то же имя, что и некая глобальная переменная. По умолчанию компилятор считает, что все упоминания этого имени в теле функции относятся к локальной переменной, а доступ к глобальной переменной осуществляется через операцию :: (унарную)

2) Имя в пространстве имен не разрешено оператором using.

3) Для доступа к статическим свойствам и методам классов.

4) Если метод класса реализован вне его определения.

## Использование функций. Математические функции.

В С++ определены в заголовочном файле <cmath> функции выполняющие некоторые часто используемые математические задачи.

| **Функция** | **Описание** |
| --- | --- |
| **abs( a )** | модуль или абсолютное значение от **а** |
| **sqrt(a)** | корень квадратный из **а,**причём **а**не отрицательно |
| **pow(a, b)** | возведение **а**в степень **b** |
| **ceil( a )** | округление **а** до наименьшего целого, но не меньше чем **а** |
| **floor(a)** | округление **а** до наибольшего целого, но не больше чем **а** |
| **fmod(a, b)** | вычисление остатка от  a/b |
| **exp(a)** | вычисление экспоненты **еа** |
| **sin(a)** | **a** задаётся в радианах |
| **cos(a)** | **a** задаётся в радианах |
| **log(a)** | натуральный логарифм **a**(основанием является экспонента) |
| **log10(a)** | десятичный логарифм **а** |
| **asin(a)** | арксинус **a**, где **-1.0 < а < 1.0** |

Необходимо запомнить то, что операнды данных функций всегда должны быть вещественными, то есть  a и b числа с плавающей точкой. Это связано с тем, что существует несколько экземпляров перегруженных функций, соответствующих списку аргументов.

## Использование функций. Указатель на функцию. Указатели на функции. Указатели на методы. Указатель на функцию и динамическое связывание. Таблицы функций, вызов по имени. Указатель на функцию как средство параметризации алгоритма.

Указатель на функцию (function pointer) хранит адрес функции. По сути указатель на функцию содержит адрес первого байта в памяти, по которому располагается выполняемый код функции.

Самым распространенным указателем на функцию является ее имя. С помощью имени функции можно вызывать ее и получать результат ее работы.

Но также указатель на функцию мы можем определять в виде отдельной переменной с помощью следующего синтаксиса:

|  |  |
| --- | --- |
| 1 | тип (\*имя\_указателя) (параметры); |

Здесь *тип* представляет тип возвращаемого функцией значения.

*имя\_указателя* представляет произвольно выбранный идентификатор в соответствии с правилами о наименовании переменных.

И *параметры* определяют тип и название параметров через запятую при их наличии.

Например, определим указатель на функцию:

|  |  |
| --- | --- |
| 1 | void (\*message) ();> |

В данном случае определен указатель, который имеет имя **message**. Он может указывать на функции без параметров, которые возвращают тип void (то есть ничего не возвращают).

Указателю на функцию можно присвоить функцию, которая соответствует указателю по возвращаемому типу и спецификации параметров:

|  |  |
| --- | --- |
| 1 | message=hello; |

То есть в данном случае указатель message теперь хранит адрес функции hello. И посредством обращения к указателю мы можем вызвать эту функцию:

|  |  |
| --- | --- |
| 1 | message(); |

В качестве альтернативы мы можем обращаться к указателю на функцию следующим образом:

|  |  |
| --- | --- |
| 1 | (\*message)(); |

Впоследствии мы можем присвоит указателю адрес другой функции, как в данном случае.

Когда программа компилируется, компилятор преобразует каждую инструкцию в вашей программе на C++ в одну или несколько строк машинного кода. Каждой строке машинного кода дается собственный уникальный адрес в последовательности адресов. То же самое и с функциями – когда встречается функция, она преобразуется в машинный код и получает следующий доступный адрес. Таким образом, каждая функция получает уникальный адрес.

**Связывание** относится к процессу, который используется для преобразования идентификаторов (например, имен переменных и функций) в адреса. Хотя связывание используется как для переменных, так и для функций, в этом уроке мы сосредоточимся на связывании функций.

## Раннее связывание

Большинство вызовов функций, с которыми сталкивается компилятор, будут прямыми вызовами функций. Прямой вызов функции – это инструкция, которая вызывает функцию напрямую.

Прямые вызовы функций могут быть разрешены с помощью процесса, известного как раннее связывание. **Раннее связывание** (также называемое **статическим связыванием**) означает, что компилятор (или компоновщик) может напрямую связать имя идентификатора (например, имя функции или переменной) с машинным адресом. Помните, что каждая функция имеет уникальный адрес. Поэтому, когда компилятор (или компоновщик) сталкивается с вызовом функции, он заменяет вызов функции инструкцией машинного кода, которая сообщает процессору перейти к адресу функции.

## Позднее связывание

В некоторых программах до момента выполнения (когда программа запущена) невозможно узнать, какая функция будет вызвана. Это называется **поздним связыванием** (или **динамическим связыванием**). В C++ один из способов получить позднее связывание – использовать указатели на функции. Вкратце, [указатель на функцию](https://radioprog.ru/post/1196) – это тип указателя, который указывает на функцию, а не на переменную. Функция, на которую указывает указатель, может быть вызвана с помощью применения оператора вызова функции (()) к указателю.

Вызов функции через указатель на функцию также известен как косвенный вызов функции. Следующая программа калькулятора функционально идентична примеру калькулятора, приведенному выше, за исключением того, что в ней вместо прямого вызова функций используется указатель на функцию

Позднее связывание немного менее эффективно, поскольку предполагает дополнительный уровень косвенного обращения. При раннем связывании CPU может напрямую перейти к адресу функции. При позднем связывании программа должна сначала прочитать адрес, содержащийся в указателе, а затем перейти к этому адресу. Это включает в себя один дополнительный шаг, что делает этот тип связывания немного медленнее. Однако преимущество позднего связывания состоит в том, что оно гибче, чем раннее связывание, поскольку решения о том, какую функцию вызывать, не нужно принимать до времени выполнения.

Тип данных вида **void (\*pp[])()** расшифровывается в соответствии с контекстным определением типа данных как массив указателей на функции с общим прототипом (схемой передачи параметров и результата) – последовательность операций в контексте – массив – указатель – вызов функции. Образно, хотя и не совсем точно этот тип можно назвать таблицей функций, вызов которых может производиться по номеру (индексу).

Оригинальность и обособленность указателя на функцию заключается в том, что указуемым объектом является не переменная (компонента данных программы), а функция (компонента алгоритма). Но сущность указателя при этом не меняется: если обычный указатель позволяет параметризовать алгоритм обработки данных, то указатель на функцию позволяет параметризовать сам алгоритм. Это значит, что некоторая его часть может быть заранее неизвестна (не определена, произвольна) и будет подключаться к основному алгоритму только в момент его выполнения (**динамическое связывание**).

Для реализации указанного принципа основная функция должна получать необходимый для ее работы «довесок» в виде **формального параметра – указателя на функцию**.

## Использование функций. Массив указателей на функции. Шаблоны функций в С++. Основные понятия. Параметры шаблонов функций.

Можно объявить массив указателей на функции. Например:

int (\*testCases[10])();

testCases – это массив из десяти элементов, каждый из которых является указателем на функцию, возвращающую значение типа int и не имеющую параметров.

Подобные объявления трудно читать, поскольку не сразу видно, с какой частью ассоциируется тип функции.

В этом случае помогает использование имен, определенных с помощью директивы typedef:

// typedef делает объявление более понятным

typedef int (\*PFV)(); // typedef для указателя на функцию

PFV testCases[10];

Данное объявление эквивалентно предыдущему.

Шаблоны функций, своими словами,— это инструкции, согласно которым создаются локальные версии шаблонированной функции для определенного набора параметров и типов данных.

На самом деле, шаблоны функций -это мощный инструмент в С++, который намного упрощает труд программиста. Например, нам нужно запрограммировать функцию, которая выводила бы на экран элементы массива. Задача не сложная! Но, чтобы написать такую функцию, мы должны знать тип данных массива, который будем выводить на экран. И тут нам говорят — тип данных не один, мы хотим, чтобы функция выводила массивы типа int, double, float и char.

Как оказалось, задача усложнилась. И теперь мы понимаем, что нам нужно запрограммировать целых 4 функции, которые выполняют одни и те же действия, но для различных типов данных. Так как мы еще не знакомы с шаблонами функций, мы поступим так: воспользуемся [перегрузкой функций](http://cppstudio.com/uchebniki/yazyk-programmirovaniya-s/peregruzka-funkcij-v-s/).

Получается, что для каждого типа данных придется свою функцию создавать. То есть, сами понимаете, что один и тот же код будет в нескольких экземплярах, нам это ни к чему. Поэтому в С++ придуман такой механизм — шаблоны функций.

Мы создаем один шаблон, в котором описываем все типы данных. Таким образом исходник не будет захламляться никому ненужными строками кода.

template <typename T>

void printArray(const T \* array, int count)

{

    for (int ix = 0; ix < count; ix++)

        cout << array[ix] << "   ";

    cout << endl;

}

**объявление параметров шаблона**:

   Сначала пишем **ключевое слово** **template**, которое сообщает компилятору, что дальше мы будем объявлять параметры шаблона.

   Параметры шаблона функции указываются в угловых скобках (<>).

   Для создания типов параметров шаблона используются **ключевые слова typename и class**. В базовых случаях использования шаблонов функций разницы между typename и class нет, поэтому вы можете выбрать любое из двух. Если вы используете ключевое слово class, то фактический тип параметров не обязательно должен быть классом (это может быть переменная фундаментального типа данных, [**указатель**](https://ravesli.com/urok-80-ukazateli-vvedenie/) или что-то другое).

   Затем называем тип параметра шаблона (обычно T).

Если требуется несколько типов параметров шаблона, то они разделяются запятыми

|  |  |
| --- | --- |
| 1  2 | template <typename T1, typename T2> |

Поскольку тип аргумента функции, передаваемый в тип T, может быть классом, а классы, как правило, не рекомендуется [**передавать по значению**](https://ravesli.com/urok-97-peredacha-argumentov-po-znacheniyu/), то лучше сделать параметры и возвращаемое значение нашего шаблона функции [**константными ссылками**](https://ravesli.com/urok-89-ssylki-i-const/)

## Использование функций. Шаблоны функций. Аргументы по умолчанию. Функции округления. Компиляция программ, состоящих из двух или более функций.

Шаблон функции - это обобщенное описание функции; т.е. он определяет функцию в терминах обобщенного типа, вместо которого может быть подставлен определенный тип данных, такой как iпt или douЫe. Передавая шаблону тип в качестве параметра, можно заставить компилятор сгенерировать функцию для этого конкретного типа. Поскольку шаблоны позволяют программировать в терминах обобщенного, а не специфического типа, этот процесс иногда называют обобщенным программированием. Поскольку типы представлены параметрами, средство шаблонов иногда называют параметризированными типами. Давайте посмотрим, чем это средство полезно, и как оно работает.

Средство шаблонов функций С++ автоматизирует этот процесс, обеспечивая высокую надежность и экономию времени. Шаблоны функций позволяют определять функции в терминах некоторого произвольного типа. Например, можно создать шаблон для осуществления обмена значениями, подобный приведенному ниже:

template void Swap (AпyType &а, АпуТуре &b)

{

AnyType temp;

а = Ь;

Ь = temp ;

}

Шаблоны должны использоваться в тех случаях, когда необходимы функции, применяющие один и тот же алгоритм к различным типам данных. Если перед вами не стоит задача обеспечения обратной совместимости и не затрудняет набор более длинного слова, можете использовать при объявлении параметров типа ключевое слово typename , а не class.

Из-за того, что размеры строк и колонок одинаковые, нам не нужно дополнительно узнавать сколько там у массива получилось в пустых скобках. Мы ведь итак знаем, что у него число строк и колонок одинаковое. Удивительно, но **этот код прекрасно работает благодаря тому, что мы использовали шаблон, в котором указали аргумент по умолчанию**.

**Для округления числа в меньшую сторону используется функция floor()** и ее разновидности для различных типов аргументов и возвращаемых параметров. Данная функция имеет следующий синтаксис:

double floor(double x);

long double floorl(long double x);

**Округление в большую сторону производится с помощью функции ceil()**:

double ceil(double x);

long double ceil(long double x);

Простейший способ использования нескольких функций в одной программе заключается в том, чтобы поместить их в один файл, после чего осуществить компиляцию программы, содержащейся в этом файле так, как будто она состояла из одной функции

Второй способ заключается в применении директивы #include Если одна функция содержится в файле с именем filel.c, а вторая в файле file2.c, поместите эту директиву в файл filel.c

## Классы памяти. "Зоопарк" классов памяти. Автоматические переменные. Регистровые переменные. Статические переменные. Внешние переменные. Внешние статические переменные.

Переменные могут либо существовать на протяжении всего выполнения программы (статические переменные) , либо только в период выполнения функции или блока (автоматические переменные).

Когда происходит вызов функции , ее автоматические переменные добавляются в стек, а указатель вершины устанавливается на свободную ячейку памяти , следующую за только что размещенными переменными. После завершения функции указатель вершины снова принимает значение, которое он имел до вызова функции. В результате эффективно освобождается память, которая использовалась для хранения новых переменных.

автоматические переменные, разделяющие между собой одно и то же имя, но определенные в разных функциях, являются отдельными переменными, не зависящими друг от друга, и каждая из них обладает собственным адресом.

Автоматические переменные могут быть объявлены с помощью спецификатора класса хранения register или вообще без спецификатора; в последнем случае они становятся автоматическими по умолчанию

каждая функция при завершении освобождает свои автоматические переменные.

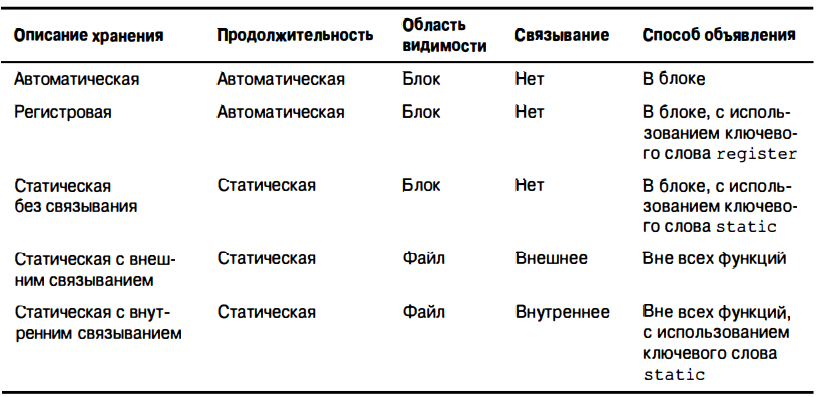
В Borland С++ спецификатор register может применяться к локальным переменным и формальным параметрам функции. Нельзя применять register к глобальным переменным. Также, поскольку регистровая переменная может быть сохранена в регистре процессора, нельзя получить адрес регистровой переменной. (Данное ограничение присутствует только в С, но не в С++)

В целом операции с регистровыми переменными выполняются гораздо быстрее, чем с переменными, сохраненными в памяти. Фактически, когда значение переменной содержится в процессоре, не требуется доступа к памяти для определения или модификации значения. Это делает регистровые переменные идеальным средством для управления циклами.

все переменные со статической продолжительностью хранения (в приведенном примере globa l, one\_file и count) существуют с момента начала выполнения программы и до ее завершения.

Переменная count, объявленная внутри функции functl {) , характеризуется локальной областью видимости и отсутствием связывания. Это означает, что она может использоваться только в рамках функции fun ctl ( ) , точно так же как автоматическая переменная llama. Но, в отличие от llama, переменная count остается в памяти, даже когда функция functl ( ) не выполняется. Переменные global и one\_file имеют область видимости файла, а это значит, что они могут использоваться, начиная с точки объявления и до конца файла.

се статические переменные обладают следующей особенностью инициализации: все биты неинициализированной статической переменной устанавливаются в О. Такая переменная называется инициализированной нулями.



Переменные с внешним связыванием часто называются просто внешними переменными. Они обязательно имеют статическую продолжительность хранения и область видимости файла. Внешние переменные определяются вне всех функций и поэтому являются внешними по отношению к любой функции. Например, они могут быть объявлены до описания функции main () или в заголовочном файле. Внешнюю переменную можно использовать в любой функции, которая следует в файле после определения переменной. Поэтому внешние переменные также называются глобальными - в отличие от автоматических переменных, которые являются локальными.

Если внешняя переменная используется в нескольких файлах, только один из них может содержать определение этой переменной (согласно правилу одного определения). Но во всех прочих файлах, где эта переменная используется, она должна быть объявлена с указанием ключевого слова extern.

Применение модификатора static к переменной с областью видимости файла обеспечивает для нее внутреннее связывание. Различие между внутренним и внешним связыванием становится значимым в многофайловых программах. В таком контексте переменная с внутренним связыванием является локальной для файла, который ее содержит. При этом обычная внешняя переменная обладает внешним связыванием, что означает возможность ее применения в различных файлах.

## Классы памяти. Объявление переменных на внутреннем уровне. Объявление переменных на внешнем уровне.

Любой из четырех спецификаторов класса памяти можно использовать для объявления переменных на внутреннем уровне. (По умолчанию используется **auto**.) Спецификатор **auto** объявляет переменную с локальным временем жизни. Она видна только в том блоке, в котором описана, и может иметь инициирующее значение.

Спецификатор класса памяти register указывает компилятору на необходимость размещения переменной в регистре (если это возможно). При использовании этого спецификатора увеличивается скорость доступа и уменьшается размер кода программы. Видимость переменной такая же, как при использовании спецификатора **auto**. Если компилятор встречает объявление **register**, но отсутствуют доступные регистры, переменной назначается класс памяти **auto** и она запоминается в оперативной памяти.

Переменная, объявленная на внутреннем уровне со спецификатором класса памяти **static**, имеет глобальное время жизни, но видима только в том блоке, в котором она объявлена. В отличие от переменных **auto**, переменные **static** сохраняют свои значения при выходе из блока. Можно инициализировать переменную **static** при помощи констант. По умолчанию начальное значение — ноль.

Переменная, объявленная со спецификатором класса памяти **extern**, является ссылкой на переменную с таким же именем, описанную на внешнем уровне в любом исходном файле программы. Внутреннее объявление **extern** используется для того, чтобы сделать видимым внутри блока описание переменной, сделанное на внешнем уровне.

В объявлении переменных на внешнем уровне можно использовать только класс памяти static или extern (auto или register использовать нельзя). Это могут быть либо описания переменных, либо ссылки на переменные, описанные в другом месте. Внешнее объявление переменной, инициализирующее переменную (неявно или явно), является объявлением-описанием

Если переменная описывается на внешнем уровне, она видима в оставшейся части того исходного файла, в котором она объявлена, и не видима в этом файле до точки ее описания. Кроме того, она не видима в других исходных файлах программы, если ее видимость не обеспечена ссылочным объявлением.

Внутри исходного файла можно только один раз описать переменную на внешнем уровне. Если использовать спецификатор класса памяти static, в другом исходном файле можно описать еще одну переменную с таким же именем и спецификатором static. Поскольку каждое статическое описание видимо только в соответствующем исходном файле, конфликтов не будет. При помощи спецификатора extern объявляется ссылка на переменную, которая описана в другом месте. Объявление external можно использовать для обеспечения видимости описания из другого файла или для того, чтобы расширить видимость переменной в том же самом файле. Переменная видима на протяжении оставшейся части того исходного файла, в котором находится объявленная ссылка.

Для того чтобы внешняя ссылка была действительной, связанная с ней переменная должна быть объявлена только один раз на внешнем уровне. Описание может располагаться в любом исходном файле программы.

## Классы памяти. Переменные класса volatile. Ключевое слово mutable. Классы памяти и область действия. Правила области действия переменной. Операция уточнения области действия в C++. Выбор класса памяти.

В тех случаях, когда необходимо предусмотреть возможность модификации переменной периферийным устройством или другой программой, используют модификатор volatile. В связи с этим компилятор не пытается оптимизировать программу путем размещения переменной в регистрах.

Несмотря ни на что, **volatile** таки используется для доступа к переменной из разных потоков.

Итак, **volatile** в языке Си - это квалификатор переменной, говорящий компилятору, что значение переменной может быть изменено в любой момент и что часть кода, которая производит над этой переменной какие-то действия (чтение или запись), не должна быть оптимизирована.

Есть три основных типа ошибок, касающихся квалификатора **volatile**:

* **неиспользование volatile там, где нужно**

обычно совершается программистами, которые не знают про существование **volatile**, или видели, но не понимают, что это такое;

* **использование volatile там, где нужно, но не так, как нужно**

присуща программистам, знающим, насколько важен **volatile** при программировании параллельных процессов или при доступе к периферийным регистрам, но не учитывающие некоторые его нюансы;

* **использование volatile там, где не нужно (бывает и такое)**

такое делают те, кто однажды обжегся на первых двух ошибках. Это не ошибка и она не приведет к неправильному поведению программы, но создаст свои неприятности.

С++ определяет ключевое слово mutable. Если член объекта объявлен с ключевым словом mutable, то он может быть переопределен. Это означает, что члены с ключевым словом mutable константного объекта не являются константными и могут быть модифицированы.

Четыре области действия для переменной следующие: блок, функция, файл и программа. Переменная, объявленная внутри блока или функции известна только в этом блоке и функции. Правила области действия для переменных, используемых с функциями, в С и C++ аналогичны. Переменная, объявленная внешней по отношению к функции, известна в том файле, в котором она описана, начиная с момента объявления и до конца файла. Переменная, объявленная на внешнем уровне в одном исходном файле и объявленная как внешняя в других файлах, имеет в качестве области действия всю программу.

. Правила области действия для переменных, используемых с функциями, в С и C++ аналогичны. Переменные могут иметь следующие области действия: локальную, файловую или внутри класса.

Локальную переменную можно использовать в описании функции. Тогда ее область действия ограничена этой функцией. Эта переменная считается доступной или видимой только внутри функции и имеет локальную область действия.

Переменные с областью действия внутри файла объявляются вне отдельных функций или классов. Эти переменный доступны или видимы во всем файле, где они объявлены и являются глобальными.

Можно задать переменную с областью действия внутри файла, а затем включить ее в описание функции, сделав область ее действия локальной. В этом случае локальная область действия имеет приоритет над файловой. В C++ имеется новое программное средство, называемое операцией уточнения области действия (::). Когда используется эта операция, выполняется обращение не к локальной переменной, а к переменной с областью действия внутри всего файла. Тогда переменная становится глобальной.

Выбор класса памяти.

Ответ на вопрос почти всегда один — «автоматический» на первый взгляд использование внешних переменных очень соблазнительно. Опишите все ваши переменные как внешние, и у вас никогда не будет забот при использовании аргументов и указателей для связи между функциями в прямом и обратном направлениях. К сожалению, у вас возникнет проблема с функцией С, коварно изменяющей переменные в функции А, а это совсем не входит в ваши интересы. Неоспоримый совокупный опыт использования машин, накопленный в течение многих лет, свидетельствует о том, что такая проблема значительно перевешивает кажущуюся привлекательность широкого использования внешних переменных.

## Классы памяти. Пространства имен. Функции и классы памяти. Объявления функций на внешнем уровне.

Пространство имен — это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных, и т. д.). Пространства имен используются для организации кода в виде логических групп и с целью избежания конфликтов имен, которые могут возникнуть, особенно в таких случаях, когда база кода включает несколько библиотек. Все идентификаторы в пределах пространства имен доступны друг другу без уточнения. Идентификаторы за пределами пространства имен могут получить доступ к членам с помощью полного имени для каждого идентификатора, например std::vector<std::string> vec;, с помощью объявления для одного идентификатора (using std::string) или директивы using для всех идентификаторов в пространстве имен (using namespace std;). Код в файлах заголовков всегда должен содержать полное имя в пространстве имен.

**Функции также могут иметь класс памяти extern и static. Функция объявляется с классом памяти extern, если она описана в другом файле, внешнем по отношению к данной программе. Невозможно объявить функцию static так, чтобы к ней можно было обращаться извне, кроме как из файла, где она описана.**

При объявлении функции на внешнем или внутреннем уровнях можно использовать спецификаторы класса памяти **static** или **extern**. *Функции, в отличие от переменных, всегда имеют глобальное время жизни. Правила видимости для функций немного отличаются от правил видимости переменных.*

**Функции, объявленные как static, видимы только в том исходном файле, в котором они описаны. Функции, находящиеся в том же файле, могут вызывать статическую функцию, а функции из других файлов — не могут. Кроме того, в другом исходном файле можно объявить другую статическую функцию с таким же именем, и не будет конфликта.**

**Функции, объявленные как external, видимы во всех исходных файлах, образующих программу (если вы где-то не переопределите функцию как static). Внешнюю функцию могут вызывать все функции.**

Объявления функций, в которых опущен спецификатор класса памяти, являются по умолчанию external.

## Дополнительные приемы программирования. Совместимость типов. Определение совместимости типов в ANSI С.

С — слабо типизированный язык, а C++ имеет чуть большую типизацию (например, перечисляемые типы). Вы видели, как язык С может выполнять автоматические и явные преобразования типов, используя оператор явного приведения типа.

Комитет ANSI С установил, что для того, чтобы два типа были совместимыми, они должны либо быть одного и того же типа, либо быть указателями, функциями или массивами, обладающими некоторыми свойствами.

Существуют две разновидности совместимости типов: совместимость имен типов и совместимость структур типов. *Совместимость имен типов* означает, что две переменные относятся к совместимым типам только в том случае, если они были описаны в объявлениях с одним и тем же именем типа. *Совместимость структур типов* означает, что две переменные имеют совместимые типы в том случае, если у их типов одинаковые структу­ры.

## Дополнительные приемы программирования. Идентичный тип. Перечисляемые типы. Типы массивов. Типы функций. Типы структур и объединений. Типы указателей. Совместимость нескольких исходных файлов.

Составной тип — это общий тип, образованный двумя совместимыми типами. Два любых совпадающих типа являются совместимыми, и их составной тип принадлежит к тому же типу.

Два арифметических типа идентичны, если они — одного типа. Краткие объявления одного и того же типа также идентичны. Обе переменные, shivalue1 и shivalue2, имеют идентичный тип:

short shivalue 1;

short int shivalue2;

В следующем примере тип int совпадает с типом unsigned int:

int sivalue1;

unsigned int sivalue2;

Однако, типы int, short и unsigned отличаются друг от друга; при работе с символьными данными большое различие существует между типами char, signed char и unsigned char.

Как постановил комитет ANSI С, любой тип, которому предшествует какой-либо модификатор доступа, порождает несовместимые типы. Например, следующие два объявления относятся к несовместимым типам:

int ivalue1;

const int ivalue2;

компилятор рассматривает объявления typedef как синонимы типов, но не как совершенно новые типы.

комитет ANSI С установил, что каждый перечисляемый тип совместим с зависящим от реализации интегральным типом; однако, в C++ перечисляемые типы с интегральными несовместимы.

Если два массива содержат элементы совместимых типов, то сами массивы считаются совместимыми. Если размер указан только для одного массива или же не указан совсем, то совместимость типов остается. Однако в случае, когда размеры указаны для обоих массивов, для того, чтобы они были совместимыми, размеры должны совпадать.

Для того чтобы считать совместимыми две функции, имеющие прототипы, должны выполниться три условия. Две функции должны иметь один и тот же возвращаемый тип и одинаковое число параметров; при этом соответствующие параметры должны быть совместимых типов. Однако, имена параметров могут не совпадать.

При объявлении в программе новой структуры или объединения вводится новый тип, отличный (не совместимый) от любого другого типа в том же исходном файле.

Однако же, ссылка на спецификатор типа, являющегося структурой, объединением или перечисляемым типом, принадлежит к тому же типу. Для связи ссылки и объявления типа используется поле тега, поэтому поле тега можно рассматривать как название типа.

Два типа указателей считаются совместимыми, если они оба указывают на совместимые типы. Составной тип двух совместимых указателей такой же, как составной тип, на который осуществляется ссылка.

каждое объявление структуры, объединения или перечисляемого типа рассматривается, как новый несовместимый тип, может возникнуть вопрос: что произойдет, если возникнет необходимость сослаться на эти типы из других файлов этой же программы.

Повторяющиеся объявления структур, объединений и перечисляемых типов совместимы в исходных файлах, если в них объявляются одни и те же элементы, в одном и том же порядке и элементы имеют совместимые типы. Для перечисляемых типов, однако, не требуется, чтобы константы перечисления объявлялись в том же порядке, хотя каждая константа должна иметь то же самое перечисляемое значение.

## Дополнительные приемы программирования. Макроопределения. Определение макросов. Макросы и параметры. Сложности при раскрытии макросов. Создание и использование собственных макросов. Макросы, поставляемые вместе с компилятором. Выбор макроопределения или функции.

Макрос — это фрагмент кода, который выглядит и работает так же, как функция.

Преимущество правильно написанного макроса заключается в скорости выполнения. Макрос раскрывается (заменяется его определением #define) во время работы препроцессора, при этом создается так называемый встраиваемый код. Именно поэтому макросы не вызывают дополнительных затрат времени, как это обычно случается при вызове функций. Однако, при каждой подстановке макроса увеличивается общий размер исполняемой программы.

Обратная ситуация — определения функций раскрываются только один раз и не имеет значения сколько раз они вызываются. Оптимальный выбор между скоростью выполнения и общим размером программы

В С имя функции имеет значение адреса, по которому эта функция располагается. Поскольку макрос является встраиваемым средством и может раскрываться много раз, не существует единого адреса, связанного с ним. Поэтому макрос нельзя использовать в контексте, где требуется указатель на функцию. Кроме того, можно объявить указатель на функцию, но невозможно объявить указатель на макрос.

В результате этого при передаче макросу неправильного количества аргументов или при ошибке в типе аргумента никаких сообщений от компилятора получено не будет.

Так как макросы раскрываются до начала фактической компиляции программы, некоторые макросы могут неправильно обрабатывать аргументы, если этот аргумент вычисляется в макросе несколько раз.

Макрос это фрагмент кода, который выглядит и работает так же, как функция. Однако функцией он не является. Имеется несколько различий между макросами и функциями:

* макрос заменяется своим определением во время работы препроцессора, то есть еще до компиляции программы. Поэтому макросы не вызывают дополнительных затрат времени, как при вызове функции
* использование макросов приводит к разрастанию исходного кода и увеличения размера исполняемой программы. Функции являются в некотором смысле антиподом макросов. Код, который они представляют, встраивается в программу только один раз, что приводит к сокращению кода программы. С другой стороны, при выполнении программы требуется дополнительное время для организации вызова функции;
* компилятор не выполняет никаких проверок типов в макросе. Поэтому при передаче макросу аргумента, не соответствующего подразумеваемому типу, или неверного числа аргументов, никаких сообщений об ошибке не будет.
* поскольку макрос является средством для встраивания фрагмента кода, не существует единого адреса, связанного с ним. Поэтому нельзя объявить указатель на макрос или использовать его адрес

Для определения макросов используется директива #define. Как и функции, макросы могут иметь параметры.

Макросы определяются аналогично символьным константам. Единственное отличие в том, что строка подстановки substitution\_string обычно содержит несколько значений:

#define search\_string substitution\_string

Следует обратить внимание на то, что в конце макроса отсутствует точка с запятой. Макроопределение не должно содержать внутри себя пробелы Препроцессор не выполняет вычислений; он только очень точно делает предложенные подстановки.

В языке С макросы могут иметь аргументы. Такие макросы необходимо определять вместе с параметрами, назначение которых аналогично аргументам функции: они определяют места подстановки фактических параметров

Внешне использование макроса похоже на использование функции (из-за чего их иногда называют псевдофункциями). Поскольку, как отмечено выше, это все-таки разные объекты, принято имена макросов записывать прописными буквами.

По этой причине в качестве общего правила рекомендуется всегда заключать в скобки каждый параметр макроса. Кроме того, если вызов макроса может появляться в инструкциях, содержащих операторы приведения типа

Если определение макроса не умещается на одной строке, оно может быть продолжено в последующих строках. Для продолжения макроопределения в следующей строке достаточно в конце текущей строки поставить знак '\'.

При использовании макросов одни символы или лексемы буквально заменяются на другие. Фактический синтаксический анализ объявления макроса, выражения внутри него и оператора, где он вызывается, выполняется после процесса раскрытия макроса

В качестве общего правила можно посоветовать всегда заключать в скобки каждый параметр, появляющийся в теле макроопределения

При использовании макросов читаемость текста программы улучшается, но нужно, однако, помнить о том, что при этом отключается автоматическая проверка типов, выполняемая компилятором, и возможны побочные эффекты, вызванные ошибками в синтаксисе вызывающих операторов.

Комитет ANSI С рекомендовал, чтобы во всех компиляторах С были описаны пять специальных макросов без параметров.

|  |  |
| --- | --- |
| **Макрос** | **Описание** |
| \_\_DATE\_\_ | Строка, представляющая в форме mmm.dd.yyyy дату создания данного файла |
| \_\_FILE\_\_ | Имя текущего обрабатываемого файла |
| \_\_LINE\_\_ | Номер текущей строки обрабатываемого файла |
| \_\_STDC\_\_ | Определен, если установлен режим совместимости с ANSI С |
| \_\_TIME\_\_ | Время начала обработки текущего файла в формате hh.mm.ss |
| \_\_TIMESTAMP\_\_ | Строковая константа, представляющая собой дату и время последней модификации исходного файла в виде Ddd Mmm hh:mm:ss уууу (день месяц часы:минуты:секунды год) |

Макроопределения должны использоваться скорее как хитрости, а не как обычные функции: они могут иметь нежелательные побочные эффекты, если вы будете неосторожны. Некоторые компиляторы ограничивают макроопределения одной строкой, и, по-видимому, лучше соблюдать такое ограничение, даже если ваш компилятор этого не делает.

Выбор макроопределения приводит к увеличению объема памяти, а выбор функции — к увеличению времени работы программы

Преимущество макроопределений заключается в том, что при их использовании вам не нужно беспокоиться о типах переменных

## Дополнительные приемы программирования. Директивы препроцессора. Директива #define. Директива #include. Заголовочные файлы. Правильное использование заголовочных файлов. Более эффективное использование заголовочных файлов. Новый стиль заголовков. Предварительная компиляция заголовочных файлов. Файлы limits.h и float.h.

Директивы препроцессора представляют собой инструкции, записанные в тексте программы на СИ, и выполняемые до трансляции программы. Директивы препроцессора позволяют изменить текст программы, например, заменить некоторые лексемы в тексте, вставить текст из другого файла, запретить трансляцию части текста и т.п. Все директивы препроцессора начинаются со знака #. После директив препроцессора точка с запятой не ставятся.

Директива #include включает в текст программы содержимое указанного файла. Эта директива имеет две формы:

#include "имя файла"

#include

Директива #include широко используется для включения в программу так называемых заголовочных файлов, содержащих прототипы библиотечных функций, и поэтому большинство программ на СИ начинаются с этой директивы.

Директива #define служит для замены часто использующихся констант, ключевых слов, операторов или выражений некоторыми идентификаторами. Идентификаторы, заменяющие текстовые или числовые константы, называют именованными константами. Идентификаторы, заменяющие фрагменты программ, называют макроопределениями, причем макроопределения могут иметь аргументы.

Директива #define имеет две синтаксические формы:

#define идентификатор текст

#define идентификатор (список параметров) текст

Если программа содержит много кода, то более оптимально было бы разнести отдельные части кода по отдельным файлам. Например, одни функции могут храниться в одном файле исходного кода, другие функции - в другом файле. При компиляции в Visual Studio все файлы автоматически компилируются в один.

Поскольку заголовочные файлы состоят из синтаксически правильных символьных (ASCII) текстов программ на C/C++ и могут включаться в другие файлы в тех местах, где использована директива #include, многие начинающие программисты используют их не по назначению. Иногда эти файлы некорректно используются для описания целых функций или наборов функций.

Заголовочные файлы применяются для описания объявлений, используемых совместно несколькими исходными файлами; они централизованно хранят объявления всех внешних переменных, прототипы функций, определения классов, структур, объединений, перечисляемых типов и встраиваемых функций. Переменные, функции и классы должны объявляться в файлах при помощи заголовочных файлов и директив #include.

заголовочный файл никогда не должен содержать нестатических определений. Если два файла в одной программе будут включать некий заголовочный файл, имеющий внешнее определение, то большинство редакторов связей (компоновщиков) не станут обрабатывать программу из-за наличия многократно описанных символов

Эффективность компиляции заголовочных файлов можно повысить, если использовать комбинации директив препроцессора.

При компиляции проще всего управлять видимостью файла заголовков, если заключить код внутри его в следующие комбинации из трех операторов:

#ifndef \_INC\_myheader

#define \_INC\_myheader /\* начало видимости \_INC\_MYHEADER \*/

...

...

...

#endif /\* окончание условной видимости \_INC\_MYHEADER \*/

Помимо этого, для включения в программу библиотек функций языка С в соответствии с новым стандартом заголовок преобразуется следующим образом: отбрасывается расширение .h и к имени заголовка добавляется префикс с. Таким образом, например, заголовок <string.h> заменяется заголовком <cstring>. Если же используемый компилятор не поддерживает объявления заголовков в новом стиле, можно по-прежнему использовать заголовки в стиле языка С, хотя это и не рекомендуется стандартом C++.

Правильное использование заголовочных файлов позволяет увеличить скорость компиляции программы; для этой же цели служат предварительно компилированные заголовочные файлы. Предварительная компиляция наиболее удобна тогда, когда оттранслированный и не меняющийся фрагмент программы используется вместе с другим фрагментом, находящимся в стадии проектирования.

Для того, чтобы создаваемые программы были переносимыми, согласно требованиям комитета ANSI C все компиляторы С документируют системно-зависимые диапазоны представления целых чисел и чисел с плавающей точкой.

|  |  |  |
| --- | --- | --- |
| **#define** | **Значение** | **Описание** |
| #define CHAR\_BIT | 8 | Количество разрядов в типе char |
| #define SCHAR\_MIN | (-127) | Минимальное значение signed char |
| #define SCHAR\_MAX | 127 | Максимальное значение signed char |
| #define UCHAR\_MAX | Oxff | Максимальное значение unsigned char |
| #define SHRT\_MIN | (-32767) | Минимальное значение (signed) short |
| #define SHRT\_MAX | 32767 | Максимальное значение (signed) short |
| #define USHRT\_MAX | Oxfff | Максимальное значение unsigned short |
| #define INT\_MIN | (-32767) | Минимальное значение (signed) int |
| #define INT\_MAX | 32767 | Максимальное значение (signed) int |
| #define UINT\_MAX | Oxfff | Максимальное значение unsigned int |
| #define LONG\_MIN | (-2147483647) | Минимальное значение (signed) long |
| #define LONG\_MAX | 2147483647 | Максимальное значение (signed) long |

## Дополнительные приемы программирования. Директивы #ifdef и #endif. Директива #undef. Директива #ifndef. Директива #if. Директива #else. Директива #elif.

Директива #ifdef позволяет проверить, определена ли символическая константа и, если – да, следующие за ней строки будут направлены для обработки компилятором. Она относится к так называемым условным директивам, поскольку проверяет выполнение некоторого условия и в зависимости от результата проверки изменяет процесс компиляции. Используемая совместно с данной (и другими условными директивами), директива #endif сообщает препроцессору о конце условного блока кода.

Директива #undef сообщает препроцессору об отмене всех предыдущих определений для указанного идентификатора

компилятор не выдаст ошибки, если будет попытка отменить имя, ранее не определявшееся; если некоторое имя было отменено, то его можно определить заново при помощи другой директивы #define.

Директива #undef отменяет самое последнее определение поименованного макроопределения.

является директива #ifndef. Эта директива также проверяет существование указанного в ней идентификатора, однако следующие за ней строки кода передаются компилятору, если этот идентификатор не определен.

Директива #if вместе с директивами #elif , #else и #endif управляет компиляцией частей исходного файла. Если выражение, которое вы пишете (после #if ), имеет ненулевое значение, группа строк, следующая непосредственно за директивой #if , сохраняется в единице перевода.

Директива #else применяется совместно с условными директивами и позволяет отделить часть кода, которая будет обрабатываться, если предыдущее условие не выполняется.

директивы - #if и #elif - также используются для проверки условия. Имя последней из них является сокращением английских слов “else if”. Она позволяет проверить условие, альтернативное установленному в директиве #if.

При использовании условных директив нужно иметь в виду следующие правила:

* для каждой директивы #if должна присутствовать соответствующая директива #endif;
* директивы #elif и #else являются необязательными; если ни одно из выражений не истинно, компилируется секция кода, следующая за #else;
* значение выражения должно быть целой константой;
* в выражениях могут применяться операции сравнения =, >, >=, <,<=.

## Дополнительные приемы программирования. Директива #line. Директива #error. Директива #pragma.

Директива #line изменяет внутренний счетчик строк компилятора.

Директива **#line** изменяет содержимое **\_\_LINE\_\_** и **\_\_FILE\_\_**, которые являются зарезервированными идентификаторами (макросами) в компиляторе. В первом из них содержится номер компилируемой в данный момент строки программного кода программы [3], а второй идентификатор – это строка, содержащая имя компилируемого исходного файла.

Директива **#line** выглядит следующим образом:

**#line номер "имя\_файла"**

В определении директивы **#line** обязательным является номер строки, относительно которой будет выполняться подсчет следующих строк. Второй параметр **"имя\_файла"** является не обязательным. Если его не будет, то идентификатор **\_\_FILE\_\_** будет содержать путь и имя программы. Если указать в качестве параметра новое имя файла, то **\_\_FILE\_\_** будет содержать его.

Директива **#line** в основном используется для отладки и специальных применений

Директива #error указывает компилятору, что нужно напечатать сообщение об ошибке и прекратить компиляцию. Обычно ее используют внутри условных директив.

Директива #еrrоr указывает компилятору на сообщение об ошибке, определенное пользователем и которое можно использовать для расширения собственных возможностей компилятора по расшифровке ошибок и выдаче сообщений. После того, как компилятор встречает директиву #error, он ищет в остальной части программы синтаксические ошибки, но объектный файл не создает.

Директива #pragma определяется стандартом ANSI С для реализации директив, предоставляю­щих компилятору различные инструкции. Стандартный вид #pragma следующий:  
  
#pragma имя

где имя — это имя директивы #pragma. Borland определяет 14 директив #pragma:  
  
argused  
anon\_struct  
codeseg  
comment  
exit  
hdrfile  
hdrstop  
inline  
intrinsic  
message  
option  
saveregs  
startup  
warn

Директива argused должна стоять перед функцией. Она используется для устранения предупреж­дений, если аргумент функции, перед которой стоит #pragma, не используется в теле функции.

Чтобы разрешить использовать анонимные структуры, укажите директиву anon\_struct. С помо­щью директивы codeseg можно указать сегмент, класс или группу, используемую функцией.

Используя директиву comment, в выходной файл, например, в файл с расширением obj или ехе можно поместить комментарии.

Директива exit определяет одну или несколько функций, вызываемых при завершении програм­мы. Директива startup определяет одну или несколько функций, вызываемых при запуске про­граммы.

С помощью директивы hdrfile можно определить имя файла, используемого для хранения пред- компилированных заголовочных файлов.

Директива hdrstop сообщает Borland С++ о необходимости остановить предкомпиляцию заго­ловочных файлов.

Воспользовавшись директивой intrinsic, можно отменить функции, которые станут внутренними: #pragma intrinsic имя\_функции

Директива message позволяет определить сообщение, выводимое в качестве предупреждения при компиляции программы.

Директива option позволяет определить опции командной строки вместо того, чтобы данные опции явно указывать в командной строке.

## Дополнительные приемы программирования. Оператор defined. Условная компиляция.

В директивах #if и #elif может применяться оператор defined. Он позволяет проверить, был ли перед этим определен идентификатор или макрос с указанным в этом операторе именем. Можно также применять операцию логического отрицания (!) для проверки того, что идентификатор или макрос не определен.

Существует несколько директив условной компиляции, позволяющих изменять порядок компиляции программы в зависимости от состояния системы. Данный процесс называется условной компиляцией и широко используется при разработке коммерческого программного обеспечения, предоставляющего и поддерживающего много различных версий одной и той же программы.

Не всегда команды препроцессора располагаются только в заголовочных файлах; их можно использовать в исходном коде для более эффективной компиляции.

Директивы условной компиляции позволяют в зависимости от условий добавить добавить в файл определенный код.

Прежде всего это такие директивы как **#if/#else/#endif**, действие которых напоминает условную конструкцию if:

|  |  |
| --- | --- |
| 1  2  3  4  5 | #if условие  текст\_1  #else  текст\_2  #endif |

Если условие возвращает ненулевое значение (то есть оно истинно), то в итоговый исходный файл вставляется *текст\_1*. Если же условие возвращает 0 (то есть оно ложно), то вставляется *текст\_2*.

## Дополнительные приемы программирования. Дополнительные операции препроцессора. Операция подстановки строки (#). Операция конкатенации (##). Операция подстановки символа (#@).

Существуют три операции, которые можно использовать только в директивах препроцессора: подстановка строки (#), конкатенация (##) и подстановка символа (#@).

Если перед параметром макроса используется символ #, то компилятор вместо значения этого параметра подставляет его имя. В результате имя аргумента преобразуется в строку. Эта операция необходима потому, что параметры не заменяются, если они входят в литеральную строку, записанную в макросе в явном виде.

Если перед параметром макроса указана операция подстановки символа (#), то компилятор при раскрытии макроса вместо параметра подставит его имя. Как и любая строка, результат этой операции может объединяться со смежными строками, если отделяется он них только пробелами.

ещё одна специальную операцию для упревления препроцессором - операцию конкатенации ##. Обычно эта операция используется в контексте функциональных макроопределений.

Если в замещающей последовательности лексем между двумя лексемами встречается операция препоцессирования ## и, возможно, что одна или обе лексемы являются параметрами, то препроцессор вначале обеспечивает подстановку фактических значений в параметры, после чего сама операция конкатенации и окружающие её пробельные символы убираются.

Операция подстановки символа предшествует формальным параметрам в макроопределении. При этом фактический параметр рассматривается как отдельный символ, заключенный в простые кавычки.

## Дополнительные приемы программирования. Обработка ошибок: perror(). Модели памяти. Модель tiny. Модель small. Модель medium. Модель compact. Модель large. Модель huge.

Что есть в языке С для обработки ошибочных ситуаций, которые могут возникнуть при работе стандартных функций? Есть глобальная переменная **errno**, в которую записывается число - номер ошибки.

Это число можно распечатать. Стала диагностика лучше? Не намного. Мы теперь заставили пользователя заучивать наизусть номера ошибок. Более того, ситуации "нет файла" в разных ОС могут соответствовать разные номера ошибок.

Хочется сообщений об ошибке в удобном для человека формате. Еще лучше - на его родном языке.

Для этого существует массив строк **sys\_errlist[ ]**, в котором записаны сообщения об ошибке. Но чем писать в коде sys\_errlist[errno] лучше воспользоваться стандартными функциями **perror**

**Perror** Заголовочный файл **stdio.h** Это функция вывода ошибок, которая выводит сообщение об ошибке на стандартном устройстве вывода.

Компилятор Visual C/C++ поддерживает шесть стандартных моделей памяти — tiny (миниатюрная), small (малая), medium (средняя), compact (компактная), large (большая) и huge (гигантская) — а также пользовательскую модель, которая используется в тех приложениях, где имеются особые требования к хранению данных и кода. Тщательно выбирая подходящую модель памяти для приложения, можно улучшить степень использования системных ресурсов и скорость выполнения программы.

При использовании модели памяти tiny создаются программные файлы с расширением .СОМ. Такие программы содержат один 64 Кб сегмент для кода и данных. Доступ ко всем объектам кода и данных осуществляется при помощи near (ближних) адресов. Эти программы не могут использовать библиотеки, содержащие far-функции, например, библиотеки графики. Файлы с расширением .СОМ можно запускать только под DOS. Кроме этого, компилятор Microsoft Visual C/C++ не создает р-код для модели tiny. Программы с этой моделью памяти используют память аналогично описанным в следующем параграфе программам с моделью памяти small. Однако, при компоновке tiny-приложений объектный файл связывается с библиотекой CRTCOM.LIB; в результате исполняемый файл получается с расширением .СОМ, а не .ЕХЕ.

Когда выбрана опция small, программа может иметь два сегмента: один для данных и один для кода. После компиляции программы с моделью памяти small получают расширение .ЕХЕ. Эта опция используется по умолчанию, когда не указана другая модель памяти. Размер обоих сегментов — данных и кода — ограничен 64 Кб. Длина программы с моделью памяти small не может превышать 128 Кб. По умолчанию в таких программах используются near-адреса как для кода, так и для данных.

Программа с моделью памяти medium может иметь один сегмент для данных, но множество сегментов — для кода. Поэтому такие программы могут содержать более 64 Кб кода, но не более 64 Кб данных. По мере необходимости программный код может занимать столько места, сколько необходимо, однако при этом размер всего блока данных не должен превышать 64 Кб. По умолчанию в программах с моделью памяти medium используются far (дальние) адреса для кода и near (ближние) адреса — для данных. Установки по умолчанию можно отменить при помощи ключевого слова \_near.

При использовании модели памяти compact программы могут иметь несколько сегментов данных, но только один сегмент кода. Эта модель памяти наиболее подходит для тех приложений С, у которых большой объем данных, но короткий код. Модель памяти compact позволяет данным занимать столько места и сегментов, сколько нужно. По умолчанию в программах с этой моделью памяти используются near-адреса для кода и far-адреса для доступа к данным. В приложении можно переопределить эти установки при помощи ключевого слова \_near или \_huge для данных и ключевого слова \_far для кода.

Как нетрудно догадаться, приложения с моделью памяти large могут иметь по несколько сегментов для кода и данных. Однако, размер отдельного объекта данных не должен превышать 64 Кб. Эта модель памяти подходит для основных программ, работающих с большими объемами данных. По умолчанию в программах с моделью памяти large far-адреса используются как для кода, так и для данных. Эти установки можно переопределить в приложении при помощи ключевого слова \_near или \_huge для данных и ключевого слова \_far для кода.

Модели памяти huge и large схожи. Главное отличие заключается в том, что при использовании модели huge снимаются ограничения на размер отдельных объектов данных. Однако, имеются ограничения на размер элементов массива, если этот массив больше, чем 64 Кб. Элементы массива не могут пересекать границы сегментов — это гарантирует эффективную адресацию для каждого элемента. Поэтому размер отдельного элемента массива не может превышать 64 Кб. Кроме того, если массив больше 128 Кб, то размер каждого элемента в байтах должен быть равным степени числа 2. Однако, если размер массива равен 128 Кб или меньше, то его элементы могут иметь любую длину до 64 Кб максимум. Необходимую модель памяти можно устанавливать непосредственно в среде разработки. Для этого достаточно выбрать пункт главного меню Option Language Options, а затем указать С or C++ Compiler options.... Следующее диалоговое окно будет иметь пункт Memory Model. Для выбора нужно щелкнуть по стрелке вниз.

## Дополнительные приемы программирования. Модификаторы функций. Модификаторы cdecl и pascal.

В зависимости от того, какая модель памяти применяется, функции и указатели на функцию являются ближними или дальними. Как и в случае с указателями на переменные, размер указателя на функцию можно изменить с помощью модификаторов near и far. Так, например, функции, обращение к которым осуществляется операционной системой или драйвером какого-либо устройства, всегда должны быть объявлены как дальние (far) и, если используется модель памяти с короткими указателями для кода (Tiny, Small или Compact), такие функции должны специфицироваться с помощью модификатора far.

Существует два варианта помещения аргументов в стек; слева направо (соглашение о вызове языка Pascal) и справа налево (соглашение о вызове языка С). По умолчанию компилятор использует соглашение, указанное в параметрах (при вызове из командной строки или в опциях интегрированной среды разработки приложения), однако программист всегда может явно указать ятя каждой функции, какое из соглашений вызова необходимо применять. Это осуществляется с помощью модификаторов cdecl и pascal.

Оба модификатора соответствующим образом влияют на внутреннее имя функции при декорировании имен (будет рассматриваться позже), тем самым сообщая компилятору использовать то или иное соглашение о вызове.

Модификатор cdecl указывает компилятору на то, что параметры вызываемой функции должны помещаться в стек в порядке, обратном следованию при вызове, то есть справа налево (первым передается аргумент, стоящий последним в списке параметров, затем предыдущий и т.д.). При этом не делается никаких предположений об ответственности функции за очистку стека. Внутреннее имя функции эквивалентно объявляемому имени с добавлением символа подчеркивания и использованием соглашения языка С о различении верхнего и нижнего регистров:

Модификатор pascal наоборот, требует прямой передачи параметров в стек, слева направо. Кроме того, данный спецификатор сигнализирует, что именно вызываемая функция ответственна за очистку стека. Данное соглашение используют на практике, если функция вызывается много раз из разных мест. Однако функции с переменным числом параметров не могут использовать данного соглашения. Внутреннее имя при использовании данного вида соглашения совпадает с именем прототипа, но с преобразованием в верхний регистр

## Файлы в C. Файлы и потоки. Связь с файлами. Закрытие потоков. Обработка ошибок в C и C++. Переменная errno и коды ошибок.

**Файл** -- именованная область на жестком диске. С точки зрения UNIX файлом называется "что-нибудь", из чего можно считывать информацию или во что можно записывать информацию.

файл является структурой, и что краткое наименование шаблона файла — FILE.

Самый распространенный связан со структурой FILE (это не класс, потому что сущность языка C). Эта структура определена в заголовочном файле стандартной библиотеки <stdio.h>. Размер этой структуры и ее поля зависят от ОС и от версии компилятора. Поэтому никто не пользуется структурой FILE. Обычно пользуются указателем на эту структуру: FILE\*.

Часто нам бывает нужна программа получения информации от файла или размещения результатов в файле. Один способ организации связи программы с файлом заключается в использовании операций переключения < и >. Этот метод прост, но ограничен. Например, предположим, вы хотите написать диалоговую программу

Си предоставляет и более мощные методы связи с файлами. Один подход заключается в использовании функции fopen(), которая открывает файл, затем применяются специальные функции ввода-вывода для чтения файла или записи в этот файл и далее используется функция fclose() для закрытия файла.

По завершении работы с потоком он должен быть закрыт. Это осуществляется с помощью функции fclose(), которая имеет следующий прототип:

int fclose(FILE \*stream);

Все буферы, связанные с потоком, освобождаются перед закрытием потока. В случае успеха fclose() возвращает 0; в случае ошибки - EOF. Если ваша программа не закрывает поток с помощью явного вызова fclose (), то он закрывается автоматически по ее завершению.

Две функции fclose() и fcloseall() используются для закрытия потока или всех потоков, соответственно. Функция fclose() закрывает отдельный файл, а функция fcloseall() закрывает все открытые потоки, за исключением stdin, stdout, stderr, stdprn и stdaux. Однако, если ваша программа не закрывает поток явно, то он закрывается автоматически по ее завершению. Поскольку количество потоков, открытых одновременно, ограничено, то желательно закрывать поток по окончанию работы с ним.

есть функция с именем реrrоr(). Эта функция выводит в поток stderr системное сообщение о последней ошибке, произошедшей при вызове библиотечной процедуры. Для этого используются переменные errno и \_sys\_errlist, предопределенные в файле stdlib.h. Массив \_sys\_erriist представляет собой массив строк сообщений об ошибках. Переменная errno — это индекс массива строковых сообщений, который автоматически устанавливается равным номеру возникшей ошибки. Число элементов в массиве определяется другой константой, \_sys\_nerr, также определенной в файле stdlib.h.

Функция реrrоr() имеет единственный параметр — строку символов. Обычно передаваемый аргумент — это строка, идентифицирующая файл или функцию, где произошла ошибка.

Стандарт ISO C определяет следующие коды:

* **EDOM** – (**E**rror **dom**ain) ошибка области определения.
* **EILSEQ** – (**E**rror **i**nva**l**id **seq**uence) ошибочная последовательность байтов.
* **ERANGE** – (**E**rror **range**) результат слишком велик.

## Файлы в C. Переключение и работа с файлами. Переключение вывода. Переключение ввода. Комбинированное переключение. Конвейерная пересылка. Текстовые и бинарные (двоичные) файлы.

Понятие ввода-вывода включает в себя функции, данные и устройства.

**Существуют два способа написания программ, работающих с файлами.**

Первый способ заключается в явном использовании специальных функций, которые открывают и закрывают файлы, организуют чтение и запись данных и т.д.

Второй способ состоит в том, чтобы использовать программу, спроектированную первоначально в предположении, что данные в нее вводятся с клавиатуры и выводятся на экран, но переключить ввод и вывод на другие информационные каналы, например, из файла в файл. Этот способ в некоторых отношениях обладает меньшими возможностями, чем первый, но зато гораздо проще в использовании. **Операция переключения**- это средство OC UNIX, а не самого языка Си. Но она оказалась настолько полезной, что при переносе компилятора с языка Си на другие вычислительные системы часто вместе с ним переносится и эта операция.

Символ < служит обозначением операции переключения, используемой в OC UNIX. Выполнение указанной операции приводит к тому, что содержимое файла words будет направлено в файл с именем get\_put. Сама программа "ввод-вывод\_ф" не знает, что входные данные поступают из некоторого файла, а не с терминала. На ее вход просто поступает поток символов, она читает их, и последовательно, по одному, выводит на печать до тех пор, пока не встретит признак EOF.

Предположим, что мы хотим создать копию *файла* my\_words и назвать его my\_words2. Нужно ввести для этого команду

get\_put < my\_words > my\_words2

и требуемое задание будет выполнено.

Команда

get\_put > my\_words2 < my\_words

приведет к такому же результату, **поскольку порядок указания операций переключения не имеет значения. Нельзя в одной команде использовать один и тот же файл и для ввода, и для вывода**.

Можно также осуществить соединение выходного потока одной программы с входным потоком другой. Это называется конвейерной пересылкой. Если имеются две выполнимые программы example1 и example2, то конвейерная пересылка между ними организуется с помощью символа вертикальной черты ‘|’.

Язык С++ унаследовал от языка C библиотеку стандартных функций ввода-вывода. Функции ввода-вывода объявлены в заголовочном файле <stdio.h>. Операции ввода-вывода осуществляются с файлами. Файл может быть текстовым или бинарным (двоичным). Различие между ними заключается в том, что в текстовом файле последовательности символов разбиты на строки. Признаком конца строки является пара символов CR (возврат каретки) и LF (перевод строки) или, что то же самое - '\г' + '\n'. При вводе информации из текстового файла эта пара символов заменяется символом CR, при выводе, наоборот, - символ CR заменяется парой символов CR и LF. Бинарный (или двоичный) файл - это просто последовательность символов. Обычно двоичные файлы используются в том случае, если они являются источником информации, не предполагающей ее непосредственного представления человеку. При вводе и выводе информации в бинарные файлы никакого преобразования символов не производится.

## Файлы в C. Потоковый ввод-вывод. Соединение и отсоединение потока от файла. Функция fopen. Функция fclose. Функция freopen.

Многие широко используемые языки высокого уровня имеют механизмы ввода/вывода, позволяющие создавать нетривиальные алгоритмы для получения и отображения сложных структур данных.

Библиотека стандартных функций ввода/вывода С позволяет считывать данные и записывать их в файлы и устройства. Однако в самом языке С отсутствуют какие-либо предопределенные файловые структуры. В С все данные обрабатываются как последовательность байт. Имеется три основных типа функций ввода/вывода: потоковые, работающие с консолью и портами, низкоуровневые.

В потоковых функциях ввода/вывода файлы или объекты данных рассматриваются как поток отдельных символов. Выбирая соответствующую потоковую функцию, вы можете обрабатывать данные любого необходимого размера или формата, начиная от отдельных символов и заканчивая большими, сложными структурами данных.

На техническом уровне, когда программа открывает файл для ввода/вывода при помощи потоковых функций, открытый файл связывается с некоторой структурой типа FILE (предопределенной в stdio.h), содержащей базовую информацию об этом файле. После открытия потока возвращается указатель на файловую структуру.

Все потоковые функции ввода/вывода обеспечивают буферизированный, форматированный или неформатированный ввод и вывод. Буферизированный поток обеспечивает место для промежуточного хранения всей информации, вводимой из потока или записываемой в поток.

Поскольку дисковый ввод/вывод занимает довольно много времени, буферизация потока разгружает приложение. Вместо того чтобы вводить данные из потока по одному символу или по одному элементу данных, потоковые функции ввода/вывода получают данные поблочно. Когда приложению необходимо обработать введенную информацию, оно просто обращается к буферу, что гораздо быстрее. Когда буфер становится пустым, выполняется считывание с диска другого блока.

если ваша программа выполнила несколько операторов вывода, которые не заполнили буфер вывода, и запись на диск не произошла, то по завершении программы эта информация будет потеряна. Для решения этой проблемы обычно выполняется вызов соответствующей функции для очистки буфера. В отличие от других языков высокого уровня, в языке С данная проблема с буферизированным вводом/выводом решается путем автоматической очистки содержимого буфера по завершении программы.

если вы используете потоковый ввод/вывод и приложение заканчивается с аварийным остановом, то буферы вывода могут оказаться неочищенными, что приведет к потере данных.

Функции низкоуровнего ввода/вывода не выполняют никакой буферизации и форматирования; они непосредственно обращаются к средствам ввода и вывода операционной системы. Эти функции позволяют обращаться к файлам и периферийным устройствам на более низком уровне, чем это делают потоковые функции. При открытии файла на этом уровне возвращается описатель файла (file handle), представляющий собой целое число, использующееся затем для обращения к этому файлу при последующих операциях. В общем случае не рекомендуется смешивать функции потокового ввода/вывода с низкоуровневыми. Поскольку потоковые функции являются буферизированными, а низкоуровневые — нет, при обращении к файлу или устройству при помощи двух разных способов возможны рассогласование или даже потеря данных в буферах. Поэтому для каждого конкретного файла необходимо использовать либо потоковые, либо низкоуровневые функции.

Можно также осуществить соединение выходного потока одной программы с входным потоком другой. Это называется конвейерной пересылкой. Если имеются две выполнимые программы examplel и example2. то конвейерная пересылка между ними организуется с помощью символа вертикальной черты " |".

fopen -- функция из стандартной библиотеки. Первый параметр -- имя файла (в текущем каталоге). Второй параметр задает режим открытия файла; в данном случае "r" означает, что файл будет открыт только для чтения. Эта функция возвращает ненулевой указатель, если открытие прошло успешно; и возвращает NULL, если произошла ошибка. Ошибка может возникать в следующих ситуациях:

* не существует файла;
* у программы недостаточно прав доступа для работы с файлом;
* По завершении работы с потоком он должен быть закрыт. Это осуществляется с помощью функции fclose(), которая имеет следующий прототип:
* int fclose(FILE \*stream);
* Все буферы, связанные с потоком, освобождаются перед закрытием потока. В случае успеха fclose() возвращает 0; в случае ошибки - EOF. Если ваша программа не закрывает поток с помощью явного вызова fclose (), то он закрывается автоматически по ее завершению.

Две функции fclose() и fcloseall() используются для закрытия потока или всех потоков, соответственно. Функция fclose() закрывает отдельный файл, а функция fcloseall() закрывает все открытые потоки, за исключением stdin, stdout, stderr, stdprn и stdaux. Однако, если ваша программа не закрывает поток явно, то он закрывается автоматически по ее завершению. Поскольку количество потоков, открытых одновременно, ограничено, то желательно закрывать поток по окончанию работы с ним.

## Файлы в C. Работа с индикаторами ошибки, позиции и конца файла (ferror, clearerr, feof, rewind, fseek, fsetpos, ftell, fgetpos). Блочный ввод-вывод (fwrite, fread).

Функция ferror() проверяет, имеются ли файловые ошибки в данном потоке stream. Возврат 0 означает отсутствие ошибок, а ненулевая величина указывает на наличие ошибки.

Флаги ошибок, связанные с потоком stream, остаются установленными до закрытия файла или до вызова rewind() или clearerr().

Для точного определения природы ошибки используется perror().

Функция clearerr() используется для сброса в 0 флага файловой ошибки, на который указывает stream. Указатель конца файла также сбрасывается.

Флаги ошибки для каждого потока при успешном вызове функции fopen() изначально устанав­ливаются в 0. После возникновения ошибки флаги остаются установленными до явного вызова функции clearerr() или rewind().

Сигнал о файловой ошибке может выдаваться по самым разным причинам, многие из которых зависят от системы. Точная природа ошибки может быть определена вызовом функции perror(), которая отображает информацию о происшедшей ошибке.

Макросом feof() контролируется указатель положения в файле — для того, чтобы установить, достигнут ли конец файла, связанный с потоком stream. Если указатель положения в файле нахо­дится в конце файла, то возвращается ненулевое значение; в противном случае возвращается нуль.

Если конец файла был уже достигнут, то последующие операции чтения будут возвращать EOF до одного из двух событий: вызова rewind() или смещения указателя положения в файле с помо­щью fseek().

Макрос особенно полезен при работе с двоичными файлами, поскольку маркер конца файла — также допустимое двоичное целое число. Для определения момента достижения конца файла необходимо в явной форме вызывать feof(), а не просто проверять переменную, возвращаемую, например, функцией getc().

Функция rewind() перемещает указатель положения в файле на начало указанного потока. Она также сбрасывает флаги конца файла и ошибки, связанные со stream. Она возвращает 0 в случае успеха и ненулевое значение в противном случае.

Функция fseek() устанавливает указатель положения в файле, связанном со stream, в соответ­ствии со значениями offset и origin. Ее основное назначение — поддерживать операции ввода/ вывода по произвольному адресу. Аргумент offset — это выраженный в байтах сдвиг от позиции, определяемой origin, до новой позиции. Аргумент origin может принимать значения 0, 1 или 2, причем 0 означает начало файла, 1 — текущую позицию, а 2 — конец файла. В stdio.h определе­ны следующие макросы для origin

Функция fsetpos() устанавливает указатель положения в файле, связанном с потоком stream, в позицию, указанную аргументом pos. Значение последнего было установлено при предшествую­щем обращении к функции fgetpos(). Тип fpos\_t определен в stdio.h. Он может представлять по­ложение в любом файле.

В случае успеха fsetpos() возвращает 0; при неудаче возвращается ненулевая величина, а пере­менной errno присваивается ненулевое значение.

Функция ftell() возвращает текущее значение указателя положения в файле для указанного потока. Это значение представляет собой количество байт, на которое указатель отстоит от начала файла.

Функция ftell() возвращает — 1L в случае ошибки. Если в данном потоке невозможен поиск по произвольному адресу (в случае, например, консоли), возвращаемое значение не определяется.

Функция fgetpos() хранит текущее значение указателя положения в файле, связанном с указа­телем stream, в переменной, на которую указывает pos. Тип fpos\_t определен в stdio.h.

В случае успеха fgetpos() возвращает 0; при неудаче возвращается ненулевая величина и errno устанавливается в одно из следующих значений:

|  |  |
| --- | --- |
| EBADF | Недействительный поток файла |
| EINVAL | Недействительный аргумент |

Функция fwrite() записывает count объектов — каждый объект по size символов в длину — в поток, указанный stream, из символьного массива, указанного buf. Указатель позиции в файле продвигается вперед на количество записанных символов.

Функция fwrite() возвращает количество действительно записанных объектов, которое в случае успеха равно затребованному количеству. Если количество записанных объектов меньше, чем это указано при вызове, то произошла ошибка.

Если поток открыт для текстовых операций, то при записи в файл символ новой строки авто­матически транслируется в последовательность возврат каретки — перевод строки.

Функция fread() считывает count объектов — каждый объект по size символов в длину — из потока, указанного stream, и помещает их в символьный массив, указанный в buf. Указатель пози­ции в файле продвигается вперед на количество считанных символов.

Функция fread() возвращает количество действительно считанных объектов. Если количество считанных объектов меньше, чем это указано при вызове, то либо произошла ошибка, либо был достигнут конец файла. Чтобы определить, что именно имело место, нужно использовать feof() или ferror().

Если поток открыт для текстовых операций, то возврат каретки и последовательности перевода строки автоматически транслируются в символы новых строк.

## Файлы в C. Символьный ввод-вывод. Функции fputc, putc, fgetc, getc, ungetc. Функции fputs и fgets. Функции fprintf, fscanf.

Функция fputc() записывает символ ch в указанный поток в позицию, соответствующую текуще­му значению указателя положения в файле, а затем дает приращение указателю положения в файле. Даже если ch был объявлен как int, он конвертируется функцией fputc() в unsigned char. Поскольку аргументы символьного типа при вызове приводятся к целым, обычно в качестве аргу­ментов используются символьные переменные. Если используется целое число, то старший байт попросту отбрасывается.

Функция fputc() возвращает значение записанного символа. В случае ошибки она возвращается EOF. Для файлов, открытых для двоичных операций, значение EOF может быть определенно с помощью функции ferror().

Макрос putc() записывает символ, содержащийся в младшем байте ch, в поток вывода, на кото­рый указывает stream. Поскольку во время обращения к функции аргументы символьного типа приводятся к целому типу, можно использовать символьные переменные как аргументы putc().

В случае успеха putc() возвращает записанный символ; в случае ошибки — EOF. Если выходной поток открыт в двоичном режиме, то EOF является допустимым значением для ch. Это значит, что придется использовать ferror(), чтобы выяснить, действительно ли произошла ошибка.

Функция fgetc() возвращает следующий за текущей позицией символ из входного потока и дает приращение указателю положения в файле.

При достижении конца файла функция fgetc() возвращает EOF. Однако поскольку EOF — дей­ствительное целое число, то при работе с двоичными файлами для обнаружения конца файла необходимо использовать feof(). EOF возвращается и в том случае, если fgetc() встречает ошибку. И здесь тоже, работая с двоичными файлами, следует использовать функцию ferror() для обнару­жения файловых ошибок.

Макрос fgetchar() определен как fgetc(stdin).

Макрос getc() возвращает следующий за текущей позицией символ во входном потоке stream и дает приращение указателю положения в файле. Символ считывается как unsigned char, преобра­зованный к переменной целого типа.

При достижении конца файла getc() возвращает EOF. Однако поскольку EOF имеет значение целого типа, при работе с двоичными файлами для контроля достижения конца файла необхо­димо использовать feof(). Если getc() обнаруживает ошибку, то также возвращается EOF. Следует помнить, что при работе с двоичными файлами для проверки ошибок в файле следует использо­вать ferror().

Функция ungetc() возвращает символ, заданный младшим байтом аргумента ch, обратно во входной поток stream. При очередной операции чтения из stream этот символ считывается снова. Обращение к функциям fflush() или fseek() прекращает действие функции ungetc() и отбрасывает возвращаемый символ.

Между двумя последовательными операциями чтения может быть возвращен обратно только один символ.

Возвратить описанным образом EOF невозможно.

Вызов ungetc() сбрасывает флаг конца файла, связанного с указанным потоком. Значение ука­зателя положения в файле для текстового потока является неопределенным до тех пор, пока все возвращенные символы не будут считаны, после чего указателю возвращается то значение, кото­рое он имел до первого вызова ungetc(). В случае двоичного потока при каждом вызове ungetc() значение указателя положения в файле уменьшается на единицу.

Возвращаемое значение равно ch в случае успеха и EOF при неудаче.

Функция fputs() во многом подобна puts(), за тем исключением, что она записывает строку в указанный поток. Функция fgets() читает строку из указанного потока, пока не встретится сим¬вол новой строки или не будет прочитано (длина - 1) символов. Если прочитан символ новой строки, то он станет частью строки (в противоположность gets()). В обоих случаях результирую¬щая строка завершается нулевым символом. Функция возвращает str в случае успеха и нулевой указатель - в случае ошибки.  
Можно использовать fgets() как альтернативу gets().

Преимущество использования fgets() над gets() состоит в том, что можно предотвратить перепол¬нение массива ввода. Массив может содержать символ новой строки.

Функции fprintf( ) и fscanf( )

     Эти функции ввода-вывода работают почти как **printf( )** и **scanf( )**, но им нужен дополнительный аргумент для ссылки на сам файл. Он является первым в списке аргументов.

## Файлы в C. Работа с буферами (setvbuf, setbuf, fflush). Стандартные потоки (putchar, getchar, puts, gets, perror). Служебные функции для работы с файлами (remove, rename, tmpfile, tmpnam).

Функция setvbuf() позволяет программисту задать буфер, его размер и режим работы для ука­занного потока. Массив символов, на который указывает аргумент buf, используется как буфер потока stream для операций ввода/вывода. Размер буфера устанавливается аргументом size, а mode определяет режим использования буфера. Если buf имеет нулевое значение, то буферизация во­обще не производится.

Допустимые значения для mode — это \_IOFRF, \_IONBF и \_IOLBF. Они определены в stdio.h. Когда аргумент mode установлен в \_IOFBF, имеет место полная буферизация. Этот режим уста­навливается по умолчанию. При установке mode в \_IONBF поток не имеет буферов, независимо от значения buf. Если mode установлен в \_IOLBF, буферизация потока производится построчно. Это значит, что очистка буфера производится каждый раз, когда в буфер для потока вывода записы­вается символ «новая строка»; для потоков ввода запрос на ввод считывает из буфера все буквы до новой строки. В обоих случаях очистка буфера производится также при его заполнении. Значение size должно быть больше нуля и меньше 32768.

Функция setbuf() используется либо для указания буфера, который будет использоваться ука­занным потоком, либо, при вызове с нулевым значением аргумента buf, для отмены буфериза­ции. Если должен быть указан буфер, определенный программистом, он должен иметь длину в BUFSIZ символов. BUFSIZ определен в stdio.h.

Функция setbuf() не возвращает никакого значения.

Если stream связан с файлом, открытым для записи, то вызов fflush() приводит к физической записи содержимого буфера в файл. Если же stream указывает на вводимый файл, то очищается входной буфер. В обоих случаях файл остается открытым.

Возврат 0 означает успех, а возврат ненулевой величины указывает на наличие ошибки по за­писи. Очистка всех буферов производится автоматически при нормальном завершении програм­мы или при заполнении буферов. Закрытие файла также приводит к очистке буферов.

Макрос putcnar() записывает символ, находящийся в младшем байте ch, в файл stdout. Функци­онально он эквивалентен putc(ch, sdout). Поскольку во время обращения к функции аргументы символьного типа приводятся к целому типу, можно использовать символьные переменные как аргументы putchar().

В случае успеха putcahar() возвращает записанный символ; в случае ошибки будет возвращен EOF. Если выходной поток открыт в двоичном режиме, то EOF является допустимым значением для ch. Это значит, что придется использовать ferror(), чтобы выяснить, действительно ли про­изошла ошибка.

Макрос getchar() возвращает очередной символ из файла stdin. Символ считывается как пере­менная типа unsigned char, преобразованная к целому. При считывании маркера конца файла возвращается EOF.

Функционально макрос getchar() эквивалентен getc(stdin).

Функция puts() выводит строку, указанную str, на стандартное устройство вывода. Нуль в кон­це строки транслируется в новую строку.

Функция puts() возвращает символ «новая строка» в случае успеха и EOF — при неудаче.

Функция gets() считывает символы из stdin и помещает их в массив символов, на который ука­зывает str. Символы считываются до тех, пока не встретится новая строка или EOF. Символ «но­вая строка» не делается частью строки, а транслируется в нулевой символ, завершающий строку.

В случае успеха gets() возвращает str, в противном случае она возвращает NULL. В случае ошиб­ки по чтению содержимое массива, на который указывает str, не определено.

Поскольку NULL возвращается как в случае ошибки, так и при достижении конца файла, то следует использовать feof() или ferror(), чтобы выяснить, что именно произошло.

Количество символов, считываемых gets(), не ограничивается. Поэтому программист должен сам следить за тем, чтобы не выйти за границы массива, на который указывает str.

Функция perror() помещает значение глобальной переменной errno в строку и записывает эту строку в файл stderr. Если str имеет ненулевое значение, то сначала выводится строка, а за ней следует двоеточие и сообщение об ошибке, соответствующее значению errno.

Функция remove() удаляет файл, указанный fname. Если файл был благополучно удален, она возвра­щает 0, а в случае ошибки —1. При ошибке errno устанавливается в одно из следующих значений:

|  |  |
| --- | --- |
| ENOENT | Файл не существует |
| EACESS | Доступ запрещен |

Функция rename() изменяет название файла со старого oldfname на новое newfname. Новое имя newfname не должно совпадать с именами, имеющимися в каталоге.

Функция rename() возвращает 0 в случае успеха и ненулевую величину — в случае ошибки. При ошибке errno устанавливается в одно из следующих значений:

|  |  |
| --- | --- |
| ENOENT | Файл не существует |
| EACESS | Доступ запрещен |
| ENOTSAM | Другое устройство |

Функция tmpfile() открывает временный файл для операции обновления данных и возвращает указатель на файл. Функция автоматически использует для файла новое имя, чтобы избежать конфликтов с уже существующими файлами.

В случае неудачи функция tmpfile() возвращает 0, в противном случае она возвращает указа­тель на поток.

Временный файл, созданный функцией tmpfile(), автоматически удаляется при закрытии этого файла или при завершении программы.

Функция tmpnam() определена стандартом ANSI С. Она создает уникальное имя файла и со­храняет его в массиве, на который указывает name.

Функция tmpnam() предназначена для создания временного имени файла, которое отличается от имени любого файла, имеющегося в данном каталоге.

Функция может быть вызвана до ТМР\_МАХ раз (число ТМР\_МАХ определено в stdio.h). При каждом вызове она генерирует новое временное имя файла.

Возвращается указатель на имя name. Если name является пустой строкой, то возвращается ука­затель на внутреннюю строку.

## Файлы в C. Низкоуровневый ввод и вывод в С. Форматированный вывод. Использование функций printf() и fprintf().

Богатый ассортимент средств управления форматом вывода в С позволяет легко создавать печатные графики, отчеты или таблицы. Двумя основными функциями, выполняющими этот форматированный вывод, являются printf() и эквивалентная функция для файлов — fprintf(). Эти функции могут использовать любые символы преобразования

|  |  |  |
| --- | --- | --- |
| **ПОЛЕ “ТИП”** | | |
| **Сим-**  **вол** | **Тип** | **Формат вывода** |
| c | int или wint\_t | printf—означает однобайтный символ  wprintf—означает широкий символ |
| C | int или wint\_t | printf—означает широкий символ  wprintf—означает однобайтный символ |
| d | int | Десятичное целое зо знаком |
| e | double | Число со знаком в виде [-]d.ddd e [знак]ddd,где d-отдельная десятичная цифра, ddd-одна или несколько десятичных цифр, ddd-ровно четыре десятичные цифры и знак “+” или ”-” |
| E | double | То же что и “е”, за исключением того, что перед экспонентой стоит “E” |
| f | double | Число со знаком в виде [-]ddd.ddd, где d-одна или несколько десятичных цифр. Число цифр после десятичной точки зависит от точности |
| g | double | Число со знаком в формате ”f” или ”e”. Используется наиболее компактная форма. Отсутствуют нули в младших разрядах. Нет десятичной точки, если за ней отсутствуют цифры |
| G | double | То же что и “g”, за исключением того, что перед экспонентой стоит “G” |
| i | int | Десятичное целое со знаком |
| n | Указатель на целое | Количество символов, записанных в поток или буфер. Адрес буфера, определяемого целочисленным аргументом |
| o | Int | Восьмеричное число без знака |
| p | Указатель на void | Печатается адрес(определяемый аргументом) |
| s | строка | printf – определяет строку однобайтных символов  wprintf - определяет строку широких символов(печать в NULL или с максимальной точностью) |
| S | строка | printf – определяет строку широких символов  wprintf - определяет строку однобайтных символов (печать в NULL или с максимальной точностью) |
| u | строка | Десятичное целое без знака |
| x | int | Шестнадцатеричное целое без знака (используются символы нижнего регистра) |
| X | int | Шестнадцатеричное целое без знака (используются символы верхнего регистра) |

функция printf() подобно puts(), она использует указатель строки в качестве аргумента. Функция printf() менее удобна, чем puts(), но более гибка. Разница заключается в том, что printf() не выводит автоматически каждую строку текста с новой строки. Вы должны указать, что хотите выводить с новых строк. Так,

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

дает то же самое, что и

puts(string);

Вы можете видеть, что первый оператор требует ввода большего числа символов и большего времени при выполнении на компьютере. С другой стороны, printf () позволяет легко объединять строки для печати их в одной строке. Например:

printf(" Хорошо, %s, %s\n", name, MSG);

Функции ввода-вывода fprintf() и fscanf() работают почти как printf() и scanf(), но им нужен дополнительный аргумент для ссылки на сам файл. Он является первым в списке аргументов.

В отличие от getc() и putc() эти две функции получают указатель типа FILE в качестве первого аргумента. Две другие, описанные ниже, получают его в качестве последнего аргумента.

## Файлы в C++. Структура стандартной библиотеки ввода-вывода. Предопределенные потоки. От файла STREAM.H к файлу IOSTREAM.H. Список классов iostream.

В языке программирования С++ стандартная библиотека ввода-вывода организована как иерархия шаблонов классов. Эта библиотека содержит два стандартных множества конкретизированных шаблонов этой иерархии: одно для работы с символами типа **char**, другое для работы с символами типа **wchar\_t**. Классы для работы с символами типа **wchar\_t** имеют те же имена, что и классы для работы с символами типа **char**, но с префиксом **w**.

Наряду с функциями библиотеки времени выполнения, предназначенными для ввода-вывода, в C++ включен дополнительный набор объектно-ориентированных подпрограмм ввода-вывода. **Главным преимуществом системы ввода-вывода C++ является то, что она может перегружаться для создаваемых вами классов**. В настоящее время используются две версии библиотеки ввода-вывода C++: старая или традиционная, основанная на предварительном варианте стандарта C++, и новая, определенная международным стандартом Standard C++. Внешне эти версии почти не различаются. Однако внутренняя их реализация совершенно различна. Новая версия библиотеки широко использует шаблоны классов. Поскольку мы еще не рассмотрели использование шаблонов классов, эти детали реализации мы будем опускать.

**Поток ввода-вывода** - это логическое устройство, предназначенное для приема и выдачи информации пользователю. Поток связан с физическим устройством с помощью системы ввода-вывода C++. Тем самым поток обеспечивает пользователю единый интерфейс при работе с системой ввода-вывода. Это означает, что, например, для вывода информации на экран монитора и для записи ее в файл используется одна и та же функция. **Когда программа на C++ начинает выполняться, автоматически** **создаются восемь предопределенных стандартных потоков. Эти стандартные потоки связаны со стандартными файлами stdin, stdout и stderr языка С**.

Как и стандартные файлы языка С, все эти потоки по умолчанию связаны с терминалом. Объект **cin** управляет вводом из буфера потока, связанного с объектом **stdin**, объявленным в <cstdio>. По умолчанию эти два потока синхронизированы. Объект **cout** управляет буфером потока, связанным с объектом **stdout**. объявленным в <cstdio>. По умолчанию эти два потока также синхронизированы. Поток **clog** - это просто буферизованная версия потока **cerr**. В буферизованной версии потока запись на реальное внешнее устройство делается, только когда буфер полон. Поэтому **clog** является более эффективным для перенаправления вывода в файл, в то время как **cerr** используется главным образом для вывода на экран терминала.

Система ввода-вывода содержит **две иерархии классов**: **одну**, предназначенную для работы с ASCII-символами, имеющими длину 8 бит, и **другую**, предназначенную для работы с UNICODE-символами, имеющими длину 16 бит. Символы первого набора называются "**узкими**", а второго - "**широкими**".

. В основе иерархии потоковых классов лежит класс **basic\_ios**, который в качестве своего подобъекта абстрактный класс **basic\_streambuf**. Последний является базовым классом для создания буфера потока, который управляет передачей элементов в поток и из него для специализированных видов потоков.

Класс **basic\_ios** является базовым для нескольких производных классов, среди которых классы **basic\_istream**, **basic\_ostream** и **basic\_iostream**. Он содержит наиболее общие функции, необходимые для всех потоков, и обслуживает информацию о состоянии, которая отражает целостность потока и буфер потока. Этот класс также обслуживает связь потоковых классов с классами буферов потоков с помощью функции-члена **rdbuf ()**. Классы, производные от **basic\_ios**, специализируют операции ввода-вывода. В свою очередь, **basic\_ios** использует класс **ios\_base**, который также является базовым классом для всех потоковых классов. Он не зависит от типа символов и инкапсулирует информацию, необходимую для всех потоков. Эта информация включает в себя:

* управляющую информацию для синтаксического анализа и форматирования;
* дополнительную информацию для нужд пользователя (которая предоставляет путь для расширения потоков, как мы увидим позже):
* наполнение потока региональными (или локальными) символами.
* В языке C++ объекты для работы с файлами называются потоками (**streams**). В данном случае слово "поток" означает то же самое, что и "файл" в языке C.

Одним из самых впечатляющих усовершенствований компилятора C++ является новая библиотека ввода/вывода, библиотека **iostream**. Средства ввода/вывода не включены в сам язык C++; они реализованы на этом языке и поставляются как компоненты стандартной библиотеки C++, что позволяет по мере необходимости развивать возможности ввода/вывода.

На нижнем уровне C++ интерпретирует файл как последовательность или как поток байтов. На этом уровне концепция типов данных отсутствует. Одна компонента библиотеки ввода/вывода отвечает за передачу этих байтов. Однако с точки зрения пользователя файл состоит из смешанной последовательности цифр и букв или, что возможно, из объектов класса. Другая компонента библиотеки ввода/вывода обеспечивает интерфейс между двумя этими уровнями. В библиотеке **iostream** предопределен набор операций для выполнения считывания и записи встроенных типов данных. Также библиотека обеспечивает пользовательские расширения для обработки классов.

**Базовые операции ввода обеспечивает класс istream, а операции вывода —** **класс ostream. Двунаправленный ввод/вывод поддерживается классом iostream, который является производным от istream и ostream.** Для пользователя предопределены четыре потоковых объекта:

|  |  |
| --- | --- |
| cin | Объект класса istream, связанный со стандартным вводом |
| cout | Объект класса ostream, связанный со стандартным выводом |
| сегг | Небуферизированный выходной объект класса ostream, связанный со стандартным устройством ошибок |
| clog | Буферизированный выходной объект класса ostream, связанный со стандартным устройством ошибок |

В любую программу, использующую библиотеку **iostream**, должен быть включен заголовочный файл **iostream.h**.

В классе **istream** есть метод **close()**, который закрывает файл (на подобие работы с файлами в языке C). Однако вызывать этот метод необязательно. Дело в том, что в деструкторе класса этот метод вызовется автоматически.

За исключением классов буферизированных потоков, все объекты ввода/вывода, описанные в библиотеке классов iostream, используют один и тот же абстрактный базовый класс потоков, называемый ios.

Классы потокового ввода

|  |  |
| --- | --- |
| istream | Используется как универсальное средство ввода или как родительский класс для других производных классов потокового ввода |
| ifstream | Используется для ввода из файлов |
| istream\_withassign | Используется для ввода из потока cin |
| istrstream | Используется для ввода строк |

Классы потокового вывода

|  |  |
| --- | --- |
| ostream | Используется как универсальное средство потокового вывода или как родительский класс для других производных классов потокового вывода |
| ofstream | Используется для вывода в файл |
| ofstream\_withassign | Используется для вывода в потоки cout, cerr и clog |
| ostrstream | Используется для вывода строк |

Классы потокового ввода/вывода

|  |  |
| --- | --- |
| iostream | Используется как универсальное средство ввода и вывода или как родительский класс для других производных классов потокового ввода/вывода |
| fstream | Класс файлового потокового ввода/вывода |
| strstream | Класс потокового ввода/вывода строк |
| stdiostream | Стандартный класс потокового ввода/вывода |

Классы буферизированных потоков

|  |  |
| --- | --- |
| streambuf | Используется как родительский класс для производных объектов |
| filebuf | Класс буферизированных потоков для дисковых файлов |
| strstreambuf | Класс буферизированнных потоков для строк |
| stdiobuf | Класс буферизированных потоков для стандартного файлового ввода/вывода |

## Файлы в C++. Простота ввода/вывода в С ++. Потоки cin, cout и сегг. Операции выделения ( >> ) и вставки ( << ). Опции ввода/вывода в С/С++.

В C++ по-прежнему можно пользоваться библиотекой стандартного ввода/вывода С, описанной в заголовочном файле stdio.h.

В языке C++ потоковый ввод/вывод описывается как набор классов в iostream.h. Эти классы используют перегруженные операции "занести в" и "получить из" << и >>.

в С отсутствуют встроенные операции ввода или вывода; функции, подобные printf(), являются частью стандартной библиотеки, но не самого языка. В C++ также нет встроенных средств ввода/вывода; это обеспечивает большую гибкость при разработке эффекгивного пользовательского интерфейса со структурами данных проектируемого приложения.

Проблема с вводом и выводом в С заключается в реализации этих функций, плохо согласующихся между собой, если рассматривать возвращаемые значения и списки параметров. По этой причине программисты предпочитают использовать функции форматированного ввода/вывода printf(), scanf() и другие — в особенности, в тех случаях, когда обрабатываемыми объектами являются числа или другие несимвольные величины. Такие функции форматированного ввода/вывода удобны и, в своем большинстве, согласованы по интерфейсу; однако, они громоздки, поскольку должны работать с объектами различных форматов.

В языке C++ концепция класса обеспечивает модульное решение задачи манипуляции с данными. В стандартной библиотеке C++ имеются три класса ввода/вывода, являющиеся альтернативой функциям универсального ввода/вывода в С. Эти классы содержат описания для одной и той же пары операций, << и >>, которые оптимизируются для всех типов данных.

Аналогами потоков stdin, stdout и stderr, имеющих прототипы в файле stdio.h, в C++ являются cin, cout и сеrr, описанные в iostream.h. Эти три потока открываются автоматически при запуске программы и становятся интерфейсом между программой и пользователем. Поток cin связан с клавиатурой терминала. Потоки cout и сегг связаны с видеодисплеем.

Еще в header <iostream> объявлены переменные std::cin и std::cerr для стандартного потока ввода и потока ошибок соответственно. Они являются объектами классов std::istream и std::ostream соотсветственно.

Аналогично тому, как stderr отличается от stdin, в языке C++ std::cerr отличается от std::cout отсутствием буферизации.

В классе std::istream есть перегруженный оператор >>. Можно считывать информацию из стандартного потока ввода (с клавиатуры).

>> ("получить из" или выделение — extraction) и << ("занести в" или вставка — insertion). Одним из главных усовершенствований C++ по сравнению с С является возможность перегрузкиопераций, которая позволяет компилятору на основании информации о типах переменных выбирать одноименную функцию или операцию, которая должна выполняться.

Расширяемость для новых типов. Другим преимуществом стандартных потоков является то, что определённые пользователем типы данных могут быть без труда в них встроены.

Простота и последовательность. Библиотека потоков поддерживает единообразный интерфейс ввода-вывода, благодаря широкому использованию перегруженных функций и операторов. Это приводит к более простому и интуитивно понятному синтаксису.

Операции помещения и извлечения из потока

Вывод в поток выполняется с помощью оператора вставки (в поток), которым является перегруженный оператор сдвига влево <<. Левым его операндом является объект потока вывода. Правым его операндом может являться любая переменная, для которой определен вывод в поток (т.е. переменная любого встроенного типа или любого определенного пользователем типа, для которого она перегружена).

нужно отметить при сравнении исходного текста программы на C++ и полученного результата, это то, что операция вставки << не генерирует автоматически символ перевода строки. При необходимости это можно делать непосредственно, включая в программу символ \n или endl.

Операция выделения >> прекращает считывание информации, когда встречает разделительный символ, который может представлять собой символ пробела, табуляции или перевода строки.

Так же как и в С, в языке С++ отсутствуют какие-либо встроенные процедуры ввода/ввода. Вместо них все компиляторы С++ поставляются с объектно-ориентированными классами iostream.

Библиотека С небуферизированного ввода/вывода — Компилятор С обеспечивает небуферизированный ввод/вывод при помощи функций \_read() и \_write(). Эти функции весьма популярны у программистов на С по причине своей эффективности и легкости их адаптации.

Буферизированный ввод/вывод ANSI С — В С имеются также буферизированные функции fread() и fwrite(). Они описаны в библиотеке stdio.h и выполняют собственную буферизацию перед непосредственным обращением к базовым процедурам ввода/вывода.

Библиотека С ввода/вывода на консоль и в порты — В С имеются дополнительные процедуры ввода/вывода, не имеющие аналогов в С++: \_getch(), \_ungetch() и \_kbhit(). Во всех приложениях, не относящихся к Windows, можно использовать эти функции, обеспечивающие прямой доступ к аппаратному обеспечению.

Библиотека Microsoft классов iostream — Библиотека классов iostream предоставляет программам на С++ возможности объектно-ориентированного ввода/вывода. Ее можно использовать вместо таких функций, как scanf(), printf(), fscanf() и fprintf(). Однако, эти классы iostream не являются обязательными для программ на С++ и многие символьные объекты, например cin, cout, cerr и clog, не совместимы с графическим интерфейсом пользователя Windows.

Библиотека Microsoft Foundation Class — Класс Microsoft CFile, находящийся в библиотеке МFС, обеспечивает приложения С++ и, в особенности, Windows - приложения средствами объектного дискового ввода/вывода. Использование этих библиотечных процедур гарантирует переносимость приложения и легкость сопровождения.

## Файлы в C++. Операции и методы классов. Вывод символов в C++. Преобразование системы счисления в C++. Форматирование строк в C++. Форматирование чисел в C++. Файловый ввод и вывод в C++.

. В новой версии каждый объект класса iostream имеет переменные, управляющие операциями форматирования: например, преобразованием системы счисления для целых чисел или точностью числа с плавающей точкой.

Флагами статуса формата можно манипулировать при помощи функций **setf()** и **unsetf()**. Метод класса setf() используется для установки определенного флага статуса формата. Имеются два перегруженных варианта:

setf(long);

setf(long, long);

Первым аргументом может быть либо битовый флаг формата, либо битовое поле формата. В таблице перечислены флаги формата, которые можно использовать в варианте setf(long) (когда имеется только флаг формата).

|  |  |
| --- | --- |
| **Флаг** | **Значение** |
| ios:: showbase | Отображает числовые константы в формате, который может читать компилятор C++ |
| ios:: showpoint | Отображает числа float с точкой и нулями в конце |
| ios:: dec | Форматирует числа в десятичной системе счисления (система счисления по умолчанию) |
| ios:: oct | Форматирует числа в восьмеричной системе счисления |

Имеются некоторые предопределенные установки по умолчанию. Например, целые числа пишутся и читаются как десятичные. Можно изменить основание системы счисления на восьмеричное, шестнадцатеричное или же обратно на десятичное. По умолчанию число с плавающей точкой выводится с точностью шесть цифр. Точность можно изменить при помощи метода определения точности.

В ранних версиях следующий оператор напечатал бы ASCII-код символа с:

cout << с;

В нынешней библиотеке печатается сама символьная переменная. Если же нужно получить ASCII-код, то используется такой оператор:

cout << (int)c;

Есть два способа для вывода значения в разных системах счисления:

cout << hex << ivalue;

и другой способ

cout.setf(ios::hex, ios::basefield);

cout << ivalue;

Для форматирования строки можно объявить массив символов, а затем выбрать необходимый формат вывода при распечатке буфера строки:

pszpadstring38[38 + INULL\_TERMINATOR];

…

ostrstream(pszpadstring38,sizeof(pszpadstring38)) << "->" **<< psz1;**

Метод класса ostrstream() является частью файла strstream.h и имеет три параметра: указатель на массив символов, размер массива и выводимая информация. Для выравнивания вправо этот оператор в начало строки psz1 добавляет пробелы. Часть строки можно вывести при помощи метода write объекта cout:

cout.write(psz1,5);

Этот оператор выведет первые пять символов строки psz1.

Можно без затруднений форматировать числовые данные: выравнивать вправо или влево, менять точность и формат (с плавающей точкой или экспоненциальный), добавлять в начало или в конец заполняющие символы, управлять знаком. Имеются некоторые установки по умолчанию.

Для осуществления операций с файлами библиотека ввода-вывода предусматривает три класса: ifstream, ofstream и fstream. Эти классы являются производными, соответственно, от классов istream, ostream и iostream. Поскольку эти последние классы, в свою очередь, являются производными от класса ios классы файловых потоков наследуют все функциональные возможности своих родителей (перегруженные операции << и >> для встроенных типов, функции и флаги форматирования, манипуляторы). Для реализации файлового ввода-вывода нужно включить в программу заголовочный файл <fstream>.

Файловый поток должен быть связан с файлом прежде, чем его можно будет использовать. С другой стороны, предопределенные потоки могут использоваться сразу после запуска программы, даже в конструкторах статических классов, которые выполняются даже раньше вызова функции main(). Вы можете позиционировать файловый поток в произвольную позицию в файле, в то время как для предопределённых потоков это обычно не имеет смысла.

Для создания файлового потока эти классы предусматривают следующие формы конструктора:

**создать поток, не связывая его с файлом**:

explicit **ifstream();**

explicit **ofstream();**

explicit **fstream();**

**создать поток, открыть файл и связать поток с файлом**:

explicit **ifstream(**constchar **\*name,**

**os::openmode mode = ios::in);**

explicit **ofstream(**constchar **\* name,**

**ios::openmode mode = ios::out |**

**ios::trunc);**

explicit **fstream(**constchar **\* name,**

**ios::openmode mode = ios::in |**

**ios::out);**

Чтобы открыть файл для ввода или вывода, можно использовать вторую форму нужного конструктора

fstream fs("FileName.dat");

или вначале создать поток с помощью первой формы конструктора, а затем открыть файл и связать поток с открытым файлом, вызвав функцию-член open (). Эта функция определена в каждом из классов потокового ввода-вывода и имеет следующие прототипы:

void **ifstream::open(**constchar **\*name,**

**ios::openmode mode = ios::in);**

void **ofstream::open(**constchar **\* name,**

**ios::openmode mode = ios::out |**

**ios::trunc);**

void **fstream::open(**constchar **\* name,**

**ios::openmode mode = ios::in |**

**ios::out);**

**Закрытие файла происходит автоматически при выходе объекта потока из области видимости, когда вызывается деструктор потока.**

Следует иметь в виду, что стандартная библиотека ввода-вывода отличается от традиционной. Этот факт нужно учитывать при переносе старых программ в современные системы программирования. Во-первых, в традиционной библиотеке функция open() имела третий параметр, задававший режим защиты файла. Во-вторых, конструктор потока fstream и функция open() для него не предусматривали установки по умолчанию режимов открытия ios:: in | ios :: out. В-третьих, стандартная библиотека не поддерживает режимы открытия ios:: noncreate и ios:: noreplace, которые были в традиционной.

Произвольный доступ в системе ввода-вывода реализуется с помощью функций **seekg()** и **seekp()**, используемых для позиционирования, соответственно, входного и выходного потока. Каждая из них имеет по два прототипа:

istream& seekg(pos type pos);

istream\_type& seekg (of f\_type& offset,

**ios\_base::seekdir dxr);**

ostream& seekp(pos\_type pos);

ostream\_type& seekp (off\_type offset,

**ios\_base::seekdir dir);**

Здесь параметр **pos** задает абсолютную позицию в файле относительно начала файла. Параметр **offset** задаст смещение в файле, а параметр **dir** — направление смещения которое может принимать значения:

ios:: beg — смещение от начала файла;

ios:: cur — смещение относительно текущей позиции;

ios:: end — смещение от конца файла

## Файлы в C++. Классы ios\_base и ios. Потоки вывода. Функции open, close и is\_open. Функция flush. Функция put. Функция seekp и tellp. Функция write. Функция str. Классы потокового вывода.

В классах ios\_base и ios определены методы, которые являются общими для входных и выходных потоков. Класс ios\_base содержит методы, которые не зависят от параметров шаблона. Наоборот, класс ios содержит шаблонно-зависимые методы. Объект класса ios\_base не может быть создан непосредственно, а только в наследуемых классах.

Объекты типа ios\_base поддерживают следующую информацию о состоянии потока.

Информацию о форматировании ввода-вывода:

* флаги форматирования,
* длину полей ввода-вывода,
* разрешимость дисплея.

Информацию о состоянии потока:

* состояние ошибки,
* маску исключений,

Другая информация:

* стек вызовов callback функций при наступлении некоторых событий,
* внутренний массив с элементами типа long,
* внутренний массив с элементами типа void\*.

Перечислим некоторые методы класса ios\_base:

flags чтение-установка флагов форматирования,

precision чтение-установка точности double,

setf установка некоторых флагов форматирования,

unsetf сброс флагов форматирования.

Перечислим некоторые методы класса ios:

operator возвращает true, если установлен любой из флагов failbit или badbit, иначе возвращает false,

bad возвращает true в случае неустранимой ошибки ввода-вывода,

clear сбрасывает управляющие состояния,

Функция ореn() является частью UNIX-подобной системы ввода/вывода и не определена в стан­дарте ANSI С.

В отличие от буферизированной системы ввода/вывода в UNIX-подобной системе предпочте­ние отдается использованию дескрипторов файлов типа int, а не указателей на FILE. Функция ореn() открывает файл с именем filename и устанавливает режим доступа к нему в соответствии со значением аргумента access. Аргумент access представляет собой комбинацию основного режима доступа и модификаторов.

| **Base** | **Значение** |
| --- | --- |
| O\_RDONLY | Открыть только для чтения |
| O\_WRONLY | Открыть только для записи |
| O\_RDWR | Открыть для чтения/записи |

Выбрав одно из этих значений, можно объединить его, используя ИЛИ, с одним или большим количеством следующих модификаторов доступа:

| **Модификаторы доступа** | **Значение** |
| --- | --- |
| O\_NDELAY | Не используется; включен для совместимости с UNIX |
| O\_APPEND | Заставляет указатель файла переместиться в конец файла перед каждой операцией записи |
| O\_CREAT | Если файл не существует, создает его с атрибутами, установленными в соответствии со значением mode |
| O\_TRUNC | Если файл существует, урезает его до длины 0, но сохраняет атрибуты файла |
| O\_EXCL | При использовании с O\_CREAT не будет создавать выходной файл, если файл с таким именем уже существует |
| O\_BINARY | Открывает двоичный файл |
| O\_ТЕХТ | Открывает текстовый файл |

 Аргумент mode требуется только при использовании модификатора O\_CREAT. В этом случае mode может принимать одно из трех значений:

| **Mode** | **Значение** |
| --- | --- |
| S\_IWRITE | Доступ по записи |
| S\_IREAD | Доступ по чтению |
| S\_IWRITE | S\_IREAD | Доступ по чтению/записи |

 В случае успеха ореn() возвращает положительное целое число, являющееся дескриптором откры­того файла. Возврат —1 означает, что файл не может быть открыт. В этом случае errno устанавли­вается в одно из следующих значений:

|  |  |
| --- | --- |
| ENOENT | Файл не существует |
| EMFILE | Слишком много открытых файлов |
| EACCESS | Доступ запрещен |
| EINVACC | Недействительный код доступа |

Функция \_rtl\_open() имеет большее количество модификаторов режима доступа при работе под управлением Windows. Возможны следующие добавочные значения:

| **Access modifiers** | **Значение** |
| --- | --- |
| O\_NOINHERIT | Файл не передается программе-потомку |
| SH\_COMPAT | Допускаются другие операции, использующие SH\_COMPAT |
| SH\_DENYRW | Только текущий дескриптор файла имеет доступ к файлу |
| SH\_DENYWR | Разрешен доступ к файлу только по чтению |
| SH\_DENYRD | Разрешен доступ к файлу только по записи |
| SH\_DENYNO | Разрешены и другие опции совместного использования, за исключением SH\_COMPAT |

Функция close() относится к UNIX-подобной системе и не определяется стандартом ANSI С. При вызове функции close() с действительным дескриптором файла она закрывает связанный с ним файл, осуществив предварительно очистку буфера записи, если это необходимо. (Дескрипто­ры файлов создаются при успешном обращении к open() или creat() и не относятся к потокам или указателям на файлы.)

В случае успеха close() возвращает 0, в противном случае —1. Причины того, что нельзя за­крыть файл, могут быть разными, однако самая распространенная заключается в том, что был преждевременно извлечен носитель информации. Например, ошибка возникнет, если до закры­тия файла извлечь диск.

Функция \_rtl\_close() работает точно так же, как close().

Метод **is\_open()** вернет 1, если файл найден и успешно открыт. Иначе вернет 0 и сработает код прописанный в блоке **else**.

**Функция flush очищает буфер.** Эта функция имеет следующий прототип:

ostream& flush ( );

**Функция put выводит символ в файл.** Эта функция имеет следующий прототип:

ostream& put ( char **ch );**

**Функция seekp устанавливает, а функция tellp читает позицию в файле для функции put.** Эти функции имеют следующие прототипы:

streampos tellp ( );

ostream& seekp ( streampos pos );

где pos - новая позиция в потоке;

ostream& seekp ( streamoff off, ios\_base::seekdir dir );

где параметры имеют следующее назначение:

off смещение в потоке относительно позиции, указанной в параметре dir.

dir направление поиска позиции, может принимать одно из следующих значений:

* ios\_base::beg – поиск от начала файла;
* ios\_base::cur - поиск от текущей позиции;
* ios\_base::end – поиск от конца файла.

**Функция write пишет последовательность символов (байтов) в файл.** Эта функция имеет следующий прототип:

ostream& write ( constchar\* s , size n );

где s указывает на область памяти, из которой данные записываются в файл, а n – количество записываемых символов (байтов). Отметим, что эта функция используется для записи данных в бинарные файлы, то есть содержимое области памяти записывается в файл без изменения. Ниже приведен пример создания бинарного файла, используя функцию write.

**Функция str выполняет запись и чтение строки в поток.** Эта функция имеет следующий прототип:

string str ( ) constчтение строки из потока

void **str ( string & s );** запись строки s в поток

Все классы **ofstream**, производные от **fstreambase** и **ostream**, позволяют выполнять форматированный и неформатированный вывод в поток **streambuf**.

## Файлы в C++. Потоки ввода. Функция gcount. Функция get. Функция getline. Функция ignore. Функция peek. Функция putback. Функция read. Функция seekg. Функция sync. Функция unget. Функция str. Классы потокового ввода.

Потоки ввода являются объектами класса istream, который обеспечивает методы для чтения информации из буфера потока. Перечислим основные методы класса istream:

operator >> форматированный ввод,

gcount возвращает количество символов, прочитанных последней операцией ввода,

get ввод символа,

getline ввод строки,

ignore удаление символов из буфера,

peek читает символ, но не удаляет его из буфера,

putback возвращает символ в буфер,

read читает блок данных,

seekg устанавливает указатель позиции файла для метода get,

sync синхронизирует буфер потока с внешним устройством,

tellg получает указатель позиции файла для метода get,

unget возвращает символ в поток,

**Функция gcount возвращает количество символов, прочитанных последней не форматирующей операцией ввода.** Эта функция имеет следующий прототип:

streamsize gcount ( ) const;

К не форматирующим операциям ввода относятся следующие операции: get, getline, ignore, read и readsome.

**Функция get вводит символ из входного потока.** Эта функция имеет следующий прототип:

int **get();**

возвращает символ, извлеченный из потока;

istream& get (char& c );

извлекает символ из потока и сохраняет его в переменной с;

istream& get (char\* s, streamsize n, char **delim = ‘\n’ );**

вводит из потока символы и записывает их в массив s, ввод символов прекращается в следующих случаях:

* если введен (n-1) символ;
* если в потоке встретился символ разделитель delim, сам символ разделитель из потока не извлекается;
* файл закончился раньше, чем прочитан (n-1) символ.

После записи прочитанных символов в массив s функция get записывает в этот массив пустой символ NULL, отмечающий конец строки.

**Функция getline вводит строки из файла.** Эта функция имеет следующий прототип:

istream& getline (char\* s, streamsize n, char **delim =** '\n');

вводит из потока символы и записывает их в массив s, ввод символов прекращается в следующих случаях:

* если введен (n-1) символ;
* если в потоке встретился символ разделитель delim, причем сам символ разделитель извлекается из потока, но не записывается в массив s;
* файл закончился раньше, чем прочитан (n-1) символ.

После записи прочитанных символов в массив s функция getline записывает в этот массив пустой символ NULL, отмечающий конец строки. Ниже приведена программа, которая вводит строки из потока и распечатывает их на консоли.

**Функция ignore удаляет символы из буфера потока.** Эта функция имеет следующий прототип:

istream& ignore ( streamsize n = 1, int **delim = EOF );**

где n – количество символов, которые необходимо удалить из потока, а delim – это символ ограничитель. Функция удаляет символы из потока до тех пор, пока не удалит n символов или не встретит символ разделитель.

**Функция peek читает символ, но не удаляет его из буфера**. Эта функция имеет следующий прототип:

int **peek ( );**

возвращает прочитанный символ, а если файл закончился или в потоке произошла ошибка, то возвращает EOF.

**Функция putback возвращает символ в буфер.** Эта функция имеет следующий прототип:

istream& putback (char **ch);**

если введенный символ совпадает с символом ch, то функция возвращает его в буфер ввода, иначе поведение функции непредсказуемо.

**Функция read читает блок данных из файла и имеет следующий прототип:**

istream& read (char\* s, streamsize n);

где s указывает на область памяти, в которую данные читаются из файла, а n – количество читаемых символов (байтов). Отметим, что эта функция используется для чтения данных из бинарных файлов, то есть содержимое файла без изменения читается в область памяти.

**Функция seekg устанавливает указатель позиции файла для метода get, а функция tellg получает указатель позиции файла для метода get.** Эти функции имеют следующие прототипы:

istream& seekg (streampos pos);

устанавливает позицию файла на значение pos, тип streampos эквивалентен типу long.

istream& seekg (streamoff off, ios\_base::seekdir dir);

сдвигает позицию файла на значение off относительно значения параметра dir, который может принимать следующие значения:

ios\_base::beg смещение относительно начала файла;

ios\_base::cur смещение относительно текущей позиции;

ios\_base::end смещение относительно конца файла.

streampos tellg ( );

возвращает указатель позиции файла для функции get, тип streampos эквивалентен типу long.

**Функция sync синхронизирует буфер потока с внешним устройством.** Эта функция имеет следующий прототип:

int **sync ( );**

**Функция unget возвращает в поток символ, предварительно извлеченный из потока.** Эта функция имеет следующий прототип:

int **unget();**

возвращает символ, извлеченный из потока.

**Функция str выполняет запись и чтение строки в поток.** Эта функция имеет следующий прототип:

string str ( ) constчтение строки из потока

void **str ( string & s );** запись строки s в поток

Используемый в следующем примере класс ifstream порожден от классов fstreambase и istream. Он выполняет операции ввода из потока filebuf.

## Файлы в C++. Потоки ввода-вывода. Классы буферизированных потоков. Класс строковых потоков. Форматирование потока. Резидентные в памяти потоки. Буферы и синхронизация.

Потоки ввода-вывода создаются на базе класса iostream, который является производным классом от классов istream и ostream и поэтому наследует все их методы. Для работы с файлами и со строками из класса iostream наследуются соответственно классы fstream и stringstream, которые позволяют, как писать данные в потоки, так и читать данные из потоков.

заголовочный файл <iostream> содержит потоки ввода-вывода: cin, cout, cerr и clog

Класс streambuf является основой для потокового ввода/вывода в С++. В этом базовом классе описаны все основные операции, выполняемые с символьными буферами.

Все производные классы, базирующиеся на классе ios, наследуют указатель на streambuf. Как показано на рисунке, класс filebuf происходит от streambuf и адаптирует этот родительский класс к работе с файлами.

03_02

Класс streambuf можно использовать для расширения возможностей класса iostream. На рисунке ранее показаны соотношения между классом ios и производными классами. Именно класс ios предоставляет порожденным классам интерфейс программирования и средства форматирования. Однако, все необходимые действия выполняют глобальные методы и виртуальные функции класса streambuf. Все производные от ios классы вызывают эти процедуры.

Все буферизированные объекты класса streambuf используют фиксированный буфер памяти, называемый областью резервирования. Эту область можно разделить на get-область для ввода и put-область для вывода. При необходимости области ввода и вывода могут накладываться друг на друга. При помощи защищенных методов класса программа получает к ним доступ и может манипулировать двумя отдельными указателями для ввода и для вывода, использующимися для символьного ввода/вывода. Учитывая реализацию производного класса в конкретной программе, каждое приложение определяет порядок использования буферов и указателей.

Для объектов streambuf имеются два конструктора, имеющие следующий синтаксис:

streambuf::streambuf();

streambuf::streambuf (char **\* рг,** int **nLength);**

Первый конструктор используется косвенно всеми порожденными от streambuf классами. Он устанавливает в null все внутренние указатели объекта типа streambuf. Второй конструктор создает объект типа streambuf, связывающийся с существующим символьным массивом.

Необходимо помнить о том, что для объектов, созданных на основе класса streambuf, имеются два различных указателя — для ввода и для вывода. Каждый указатель работает независимо от другого.

Для управления форматированием библиотека ввода-вывода предусматривает три вида средств:форматирующие функции, флаги и манипуляторы. Все эти средства являются членами класса basic\_ios и потому доступны для всех потоков.

Рассмотрим вначале форматирующие функции-члены. Их всего три: width (). precision () и fill ().

По умолчанию при выводе любого значения оно занимает столько позиций, сколько символов выводится. Функция width() позволяет задать минимальную ширину поля для вывода значения. При вводе она задает максимальное число читаемых символов. Если выводимое значение имеет меньше символов, чем заданная ширина поля, то оно дополняется символами-заполнителями до заданной ширины (по умолчанию - пробелами). Однако если выводимое значение имеет больше символов, чем ширина отведенного ему поля, то поле будет расширено до нужного размера. Эта функция имеет следующие прототипы:

streamsize width(streamsize wide);

streamsize width() const;

Тип streamsize определен в заголовочном файле <iostream> как целочисленный. Функция с первым прототипом задает ширину поля wide, а возвращает предыдущее значение ширины поля. Функция со вторым прототипом возвращает текущее значение ширины поля. По умолчанию она равна нулю, то есть вывод не дополняется и не обрезается. В ряде компиляторов после выполнения каждой операции вывода значение ширины поля возвращается к значению, заданному по умолчанию.

Функция precision () позволяет узнать или задать точность (число выводимых цифр после запятой), с которой выводятся числа с плавающей точкой. По умолчанию числа с плавающей точкой выводятся с точностью, равной шести цифрам. Функция имеет precision () следующие прототипы:

streamsize precision(streamsize prec);

streamsize precision() const;

Функция с первым прототипом устанавливает точность равной рrес и возвращает предыдущую точность. Функция со вторым прототипом возвращает текущую точность.

**Замечание**. Если не установлен флаг scientific или fixed, задает precision() задает общее число цифр.

Функция fill() позволяет прочесть или установить символ-заполнитель. Она имеет следующие прототипы:

char\_type fill(char\_type ch);

char\_type fill() const;

Функция с первым прототипом устанавливает ch в качестве текущего символа-заполнителя и возвращает предыдущий символ-заполнитель. Функция со вторым прототипом возвращает текущий символ-заполнитель. По умолчанию в качестве символа-заполнителя используется пробел. Тип данных char\_type является параметром класса basic\_ios и может обозначать набор "узких" или "широких" символов.

Язык C++ поддерживает два вида ввода-вывода: файловый ввод-вывод и резидентные в памяти потоки. Файловый ввод-вывод предполагает передачу данных из внешнего устройства или в него. Напротив, резидентные потоки не предполагают работу с внешними устройствами. В этом случае не требуется преобразования кодов и выполнения операций с внешним устройством: осуществляется только форматирование информации. Результат такого форматирования сохраняется в памяти и может быть восстановлен в виде символьной строки. Класс istrstream предназначен для чтения символов из массива в памяти. Он использует private-объект strstreambuf для управления связанным с ним массивом. Поскольку он является производным от basic\_istream, он может использовать функции для форматированного и неформатированного ввода. Класс ostrstream предназначен для записи массива в память. Он также использует private-объект strstreambuf для управления связанным с ним массивом. Поскольку он является производным от basic\_ostream, он также может использовать функции для форматированного и неформатированного вывода. Класс strstream позволяет читать и записывать в массив символов в памяти и является производным от класса basic\_iostrean. Все эти классы определены в заголовочном файле <strstream>. В контексте операций ввода-вывода массивы символов в памяти часто называют резидентными в памяти потоками (строк).

Для работы с резидентным в памяти потоком нужно сначала связать этот поток с некоторой областью памяти (массивом символов). Затем весь ввод-вывод в этот массив можно выполнять с помощью операторов излечения и помещения в поток. Чтобы открыть массив для ввода из него нужно создать объект класса istrstream и, тем самым, связать нужный поток с заданным массивом.

Потоки выполняют обмен данными между прикладной программой и файлом через буферы, которые являются объектами классов filebuf и stringbuf соответственно. Эти классы являются производными от абстрактного класса streambuf. Каждый буфер содержит в зависимости от режима доступа один или два символьных массива: один для ввода, а второй для вывода данных. Каждый поток поддерживает внутренний указатель на буфер. Ввод-вывод данных из буфера в файл или в строку называется синхронизацией буфера с внешним устройством ввода-вывода. Синхронизация буфера с внешним устройством может выполняться неявно или явно. Неявно синхронизация выполняется в случае закрытия файла или заполнения буфера. Для явной синхронизации используют манипуляторы flush и endl или функцию sync.

## Файлы в C++. Условные признаки файлов в C++. Опрос и установка состояния потока. Ошибки потоков. Часто применяемые функции. Двоичные файлы. Объединение программ на С и С++. Использование спецификатора extern "С".

С каждым потоком связано слово состояния ошибок. При возникновении некоторой ошибки в слове состояния устанавливаются разряды, соответствующие ее типу. В соответствие с принятым соглашением при выводе объектов в поток ostream, имеющий установленные разряды ошибок, операции записи игнорируются и не изменяют состояние потока. Объекты библиотеки ostream имеют набор предопределенных условных признаков, отображающих текущее состояние потока.

|  |  |
| --- | --- |
| **Методы функций** | **Действие** |
| eof() | Возвращает ненулевое значение при достижении конца файла |
| fail() | Возвращает ненулевое значение, если операция завершена с ошибкой |
| bad() | Возвращает ненулевое значение, если возникла ошибка |
| good() | Возвращает ненулевое значение, если отсутствуют установленные разряды |
| rdstate() | Возвращает текущее состояние потока |
| сleаг() | Устанавливает состояние потока (int=0) |

Эти методы классов можно использовать при реализации различных алгоритмов для определения конкретных состояний ввода/вывода и для большей читаемости программы

Класс ios поддерживает информацию о состоянии потока после каждой операции ввода-вывода. Текущее состояние потока хранится в объекте типа iostate, который объявлен следующим образом:

typedefint **iostate;**

Состояния потока являются элементами перечислимого типа io\_state, который может иметь значения, представленные в таблице.

**Состояния потока и их значения**

|  |  |
| --- | --- |
| **Состояние** | **Значение** |
| Goodbit | Ошибок нет |
| Eofbit | Достигнут конец файла |
| Failbit | Имеет место ошибка форматирования или преобразования |
| Badbit | Имеет место серьёзная ошибка |

Для опроса и установки состояния потока можно использовать функции-члены класса ios. Имеется два способа получения информации о состоянии операции ввода-вывода. Во-первых, можно вызвать функцию rdstate(), имеющую следующий прототип:

iostate rdstate() const;

Функция возвращает состояние операции ввода-вывода. Во-вторых, можно воспользоваться одной из следующих функций-членов:

bool **good()** const;

bool **eof()** const;

bool **fail()** const;

bool **bad()** const;

Каждая из этих функций возвращает true, если установлен соответствующий бит состояния (точнее, функция fail() возвращает true, если установлен бит failbit или badbit).

Если прочитано состояние, которое сигнализирует об ошибке, его можно сбросить с помощью функции clear ():

void **clear(iostate state = ios::goodbit);**

Установить нужное состояние можно с помощью функции setstate():

void **setstate(iostate state);**

Кроме перечисленных функций, класс ios содержит функцию приведения типа

operatorvoid\*() const;

(она возвращает NULL, если установлен бит badbit) и перегруженный оператор логического отрицания

По умолчанию поток ввода-вывода не выбрасывает никаких исключений. Вы должны сами активизировать исключение. Для этой цели поток содержит маску исключений. Каждый из флагов этой маски соответствует одному из флагов ошибки (состояния потока). Ключевую роль в выбрасывании исключений играют две функции-члена класса ios:

iostate exceptions() const;

Эта функция возвращает маску, которая определяет, какие флаги состояния, установленные в маске флагов состояния, будут вызывать выбрасывание исключения. Она неявно использует вызов функции rdstate (). Другая функция

void **exceptions(iostate except);**

устанавливает маску исключений потока в значение, определяемое переменной except, а затем вызывает clear (rdstate ()) для сброса, возможно, установленных битов маски состояния потока.

Функция get() может также использоваться для чтения строки символов.

Для того, чтобы пропустить при вводе несколько символов, используется функция ignore():

Функция peek (), позволяет "заглянуть" во входной поток и узнать следующий вводимый символ. При этом сам символ из потока не извлекается.

. Функция flush() позволяет вызвать принудительную запись в файл до заполнения буфера.

Функция rdbuf() позволяет получить указатель на связанный с потоком буфер

функция setbuf () позволяет связать с потоком другой буфер

Когда вы открываете файл в текстовом режиме, происходит следующее:

* при вводе каждая пара символов '\r'+'\n' (возврат каретки + перевод строки) преобразуется в символ перевода строки ('\n');
* при выводе каждый символ перевода строки ('\n') преобразуется в пару ' \r' + ' \n' (возврат каретки + перевод строки).

С этой целью система ввода-вывода предоставляет вам возможность осуществления неформатируемого ввода-вывода, то есть записи и чтения двоичной информации (иногда говорят - сырых данных). Для осуществления ввода-вывода в двоичном режиме, включите флаг ios :: binary в параметр open\_mode, передаваемый конструктору потока или функции open(). Чтение двоичной информации из файла осуществляется функцией read(), которая имеет следующий прототип:

Двоичные файлы или потоки содержат последовательность байтов, полностью соответствующих их представлению на внешнем устройстве (диске, ленте или терминале). В двоичном файле никаких преобразований символов не выполняется, и поэтому число прочитанных или записанных байтов совпадает с числом байтов, располагающихся в файле на внешнем устройстве.

Однако, в С/С++ можно использовать extern вместе со строкой. Эта строка указывает на то, что для описываемых идентификаторов используются соглашения о компоновке другого языка. По умолчанию для программ С++ задается строка "С++".

Это необходимо для того, чтобы из программы, написанной на С++, были доступны функции и данные С. Конечно же, это нужно делать только для функций с одинаковыми именами. Если не отменить эту опцию, то компоновщик обнаружит несколько глобальных функций с одним и тем же именем.

extern"С" **тип\_ функции имя\_ функции(тип\_параметра (тип\_параметров) параметр (параметры));**

## Файлы в C++. Манипуляторы. Манипуляторы без параметров. Манипуляторы с одним параметром. Манипуляторы с несколькими параметрами.

**Манипулятором** называется глобальная функция, которая изменяет поток. Манипуляторы предназначены для включения в последовательность операторов ввода или вывода и могут изменять состояние потока, управляя его флагами. Манипуляторы подразделяются на две группы: простые манипуляторы и параметризованные манипуляторы.

Простые манипуляторы не требуют аргументов. Ниже перечислены простые манипуляторы:

endl помещает в выходной поток символ ‘\n’ и вызывает манипулятор flush,

ends помещает в поток символ ‘\0’,

flush освобождает поток,

dec устанавливает 10 c/c,

hex устанавливает 16 c/c,

oct устанавливает 8 c/c,

ws игнорирует при вводе ведущие пробелы.

Манипуляторы с аргументами называются параметризованными. Ниже перечислены параметризованные манипуляторы, которые объявлены в заголовочном файле <iomanip>:

resetiosflags сбрасывает флаги форматирование,

setiosflags устанавливает флаги форматирования,

setbase устанавливает систему счисления,

setfill устанавливает символ заполнитель,

setprecision устанавливает точность для плавающих чисел,

setw устанавливает ширину поля для символа заполнителя.

Манипуляторы используются вместе с операциями вставки << и выделения >> аналогично выходным данным или входным переменным. Однако, в соответствии со своим названием манипуляторы могут выполнять различные действия над входным и выходным потоками.

Все функции-манипуляторы вывода без параметров имеют следующий формат записи:  
  
ostream &имя\_манипулятора (ostream &поток)// кодreturn поток;}

В библиотеке классов Microsoft iostream, имеющей прототипы в файле iomanip.h, описан набор специальных макросов, предназначенных для создания параметризованных макроопределений. Простейший параметризованный макрос может принимать один аргумент типа int или long.

## Функции для работы с файлами. Полезные функции для работы с файлами. clearerr(). fclose(). fcloseall(). fdopen(). feof(). ferror(). fflush(). fgetc(). fgetchar(). fgetpos().

Функция clearerr() используется для сброса в 0 флага файловой ошибки, на который указывает stream. Указатель конца файла также сбрасывается.

Флаги ошибки для каждого потока при успешном вызове функции fopen() изначально устанав­ливаются в 0. После возникновения ошибки флаги остаются установленными до явного вызова функции clearerr() или rewind().

Сигнал о файловой ошибке может выдаваться по самым разным причинам, многие из которых зависят от системы. Точная природа ошибки может быть определена вызовом функции perror(), которая отображает информацию о происшедшей ошибке.

Функция fclose() используется для закрытия потока, ранее открытого с помощью fopen(). Она сохраняет в файл данные, находящиеся в дисковом буфере, и выполняет операцию системного уровня по закрытию файла. Вызов fclose() освобождает блок управления файлом, связанный с потоком, и делает его доступным для повторного использования.

Функция fclose() имеет прототип:  
  
int fclose(FILE \*fp);  
  
где fp - это указатель на файл, возвращенный fopen(). Если возвращен 0, то это означает, что операция закрытия выполнена успешно, а если EOF, то, значит, была ошибка.

Функция fcloseall() закрывает все открытые потоки, кроме stdin, stdout, stdaux, stdprn и stderr. Она не определена стандартом ANSI С.

Функция fdopen() не определена стандартом ANSI С. Она возвращает поток, который совместно использует файл, связанный с handle, где handle — это действительный дескриптор файла, полученный после вызова одной из UNIX-подобных процедур ввода/вывода. В сущности, fdopen() — это мост между основанной на потоках файловой системой ANSI и UNIX-подобной файловой системой. Значение mode должно быть таким же, каким был режим при открытии файла в первый раз.

Макросом feof() контролируется указатель положения в файле — для того, чтобы установить, достигнут ли конец файла, связанный с потоком stream. Если указатель положения в файле нахо­дится в конце файла, то возвращается ненулевое значение; в противном случае возвращается нуль.

Если конец файла был уже достигнут, то последующие операции чтения будут возвращать EOF до одного из двух событий: вызова rewind() или смещения указателя положения в файле с помо­щью fseek().

Макрос особенно полезен при работе с двоичными файлами, поскольку маркер конца файла — также допустимое двоичное целое число. Для определения момента достижения конца файла необходимо в явной форме вызывать feof(), а не просто проверять переменную, возвращаемую, например, функцией getc().

Функция ferror() проверяет, имеются ли файловые ошибки в данном потоке stream. Возврат 0 означает отсутствие ошибок, а ненулевая величина указывает на наличие ошибки.

Флаги ошибок, связанные с потоком stream, остаются установленными до закрытия файла или до вызова rewind() или clearerr().

Для точного определения природы ошибки используется perror().

Если stream связан с файлом, открытым для записи, то вызов fflush() приводит к физической записи содержимого буфера в файл. Если же stream указывает на вводимый файл, то очищается входной буфер. В обоих случаях файл остается открытым.

Возврат 0 означает успех, а возврат ненулевой величины указывает на наличие ошибки по за­писи. Очистка всех буферов производится автоматически при нормальном завершении програм­мы или при заполнении буферов. Закрытие файла также приводит к очистке буферов.

Функция fgetc() возвращает следующий за текущей позицией символ из входного потока и дает приращение указателю положения в файле.

При достижении конца файла функция fgetc() возвращает EOF.

Макрос fgetchar() определен как fgetc(stdin).

Функция fgetpos() хранит текущее значение указателя положения в файле, связанном с указа­телем stream, в переменной, на которую указывает pos. Тип fpos\_t определен в stdio.h.

## Функции для работы с файлами. fgets(). filelength(). fileno(). flushall() fflush()). fopen(). fprintf(). fputc(). fputchar(). fputs(). fread().

Функция fgets() считывает до num-1 символов из файла stream и помещает их в массив символов, на который указывает str. Символы считываются до тех пор, пока не встретится символ «новая строка», EOF или до достижения указанного предела. По окончании считывания в массив str сразу после последнего считанного символа помещается нулевой символ. Символ «новая строка» при считывании будет сохранен и станет частью массива str.

В случае удачи fgets() возвращает str, при неудаче возвращается NULL.

Функция filelengtn() не определена стандартом ANSI С. Она возвращает выраженную в байтах длину файла, ассоциированного с дескриптором файла handle. Надо помнить, что возвращаемая переменная имеет тип long.

Функция fileno() не определена стандартом ANSI С. Она возвращает дескриптор файла указанного потока.

Не определена стандартом ANSI С. Вызов flushall() приводит к записи физического содержимого всех выходных буферов, связан­ных с потоками файлов, в соответствующие им файлы и к очистке всех входных буферов. Все потоки остаются открытыми.

Возвращается количество открытых потоков. Те же операции автоматически осуществляются над буферами при нормальном завершении программы или при их заполнении. При закрытии файла также производит описанную очистку с буферами.

Если stream связан с файлом, открытым для записи, то вызов fflush() приводит к физической записи содержимого буфера в файл. Если же stream указывает на вводимый файл, то очищается входной буфер. В обоих случаях файл остается открытым.

Функция fopen() открывает файл, имя которого указано аргументом fname, и возвращает свя­занный с ним указатель. Тип операций, разрешенных над файлом, определяется аргументом mode.

int fprintf(FILE \**stream*, const char \**format*, ...);

Функция fprintf() выводит в поток, адресуемый параметром *stream*, значения аргументов, составляющих список аргументов, в соответствии с заданной строкой формата *format*. Возвращаемое значение равно количеству реально выведенных символов. Если при выводе возникла ошибка, возвращается отрицательное число.

Функция fputc() записывает символ ch в указанный поток в позицию, соответствующую текуще­му значению указателя положения в файле, а затем дает приращение указателю положения в файле. Даже если ch был объявлен как int, он конвертируется функцией fputc() в unsigned char. Поскольку аргументы символьного типа при вызове приводятся к целым, обычно в качестве аргу­ментов используются символьные переменные. Если используется целое число, то старший байт попросту отбрасывается.

Функция fputc() возвращает значение записанного символа. В случае ошибки она возвращается EOF.

Функция fputchar() записывает символ ch в stdout. Даже если ch был объявлен как int, он кон­вертируется функцией fputchar() в unsigned char. Поскольку все аргументы символьного типа при вызове приводятся к переменным целого типа, то обычно в качестве аргументов используются символьные переменные. Если используется целое число, то старший байт попросту отбрасывает­ся. Вызов fputcnar() функционально эквивалентен вызову fputc(ch, stdout).

Функция fputchar() возвращает значение записанного символа. В случае ошибки возвращается EOF.

Функция fputs() записывает содержимое строки, на которую указывает sir, в заданный поток. Нуль в конце строки не записывается.

В случае успеха функция fputs() возвращает последний записанный символ, а в случае неудачи EOF.

Функция fread() считывает count объектов — каждый объект по size символов в длину — из потока, указанного stream, и помещает их в символьный массив, указанный в buf. Указатель пози­ции в файле продвигается вперед на количество считанных символов.

Функция fread() возвращает количество действительно считанных объектов. Если количество считанных объектов меньше, чем это указано при вызове, то либо произошла ошибка, либо был достигнут конец файла.

## Функции для работы с файлами. freopen(). fscanf(). fseek(). fsetpos(). fstat(). ftell(). fwrite(). getc(). getchar(). gets(). getw().

**FILE\* freopen(const char\*filename, const char\*mode, FILE\*stream)**

закрывает поток, на описание которого указывает stream, а затем открывает файл, на спецификацию которого указывает ASCIIZ-строка filename. Режим открытия файла задает ASCIIZ- строка mode. Символы, формирующие режим, приведены в табл.3.1. В случае успеха функция возвращает указатель на описание открытого потока, которое будет совпадать с stream. В противном случае функция возвращает NULL. Повторное открытие того же самого файла изменяет права доступа, очищает внутренние буферы и позиционирует указатель записи-чтения либо на начало файла (r, w, r+, w+), либо на его конец (а, а+). Если файлы разные, переоткрытие приводит к переадресации файлового ввода-вывода. Эта функция обычно применяется для переадресации предоткрытых потоков stdin, stdout, stderr, stdaux, stdprn в файлы, определяемые пользователем.

**int fscanf(FILE \* stream, char \* format[, adress,...]);**

сканирует посимвольно набор вводимых полей, считывая их из потока. Затем каждое поле из потока форматируется в соответствии со спецификацией формата, которая передается fscanf в виде указателя на строку format. Полученное в результате этого поле fscanf запоминает в аргументах, передаваемых функции fscanf после параметра format. Количество аргументов должно совпадать с количеством спецификаций формата.

**int fseek(FILE \*fp, long число\_байт, int начало);**

Можно выполнять операции произвольного чтения и записи, используя систему буферизированного ввода/вывода, с помощью fseek(), устанавливающей текущую файловую позицию.

**int fsetpos(FILE\* stream, const fpos\_t\* pos);**

Устанавливает индикатор позиции файла в позицию, на которую указывает pos. Индикатор конца файла сбрасывается. Код возврата: удача – 0, неудача – не 0.

**fstat()** - Возвращает информацию об открытом файле

**long int ftell(FILE \*stream);**

возвращает положение указателя текущей позиции файла, связанного с потоком stream. Значение выдается в виде смещения в байтах относительно начала файла.

Значение, возвращаемое ftell, можно в дальнейшем использовать при вызове функции fseek.

Функция ftell возвращает положение указателя текущей позиции при успешном завершении. При ошибке возвращает-1L, и присваивает переменной errno положительное значение.

**fwrite()** - Записывает неформатированные данные в поток

**getc()** - Этот макрос считывает символ из потока

**getchar()** - Этот макрос считывает символ из stdin

**gets()** - Возвращает строку из stdin

**getw()** - Считывает целое число из потока

## Функции для работы с файлами. perror(). printf(). putc(). putchar(). puts(). putw(). remove(). rename(). rewind(). scanf(). setbuf().

## Функции для работы с файлами. setvbuf(). sprintf(). sscanf(). tmpfile(). tmpnam(). ungetc(). vfprintf(). vfscanf(). vprintf(). vsprintf(). vsscanf().

## Функции для работы с файлами. Низкоуровневый ввод и вывод. close(). lseek(). open(). read(). unlink() (\_unlink()). write().

## Структуры и другие типы данных. Структуры С и C++. Структуры С и C++: синтаксис и правила. Структуры C++: расширения синтаксиса и правил. Доступ к элементам структуры. Инициализация структуры. Передача структур в функции.

## Структуры и другие типы данных. Массив структур. Описание массива структур. Определение элементов массива структур. Вложенные структуры.

## Структуры и другие типы данных. Использование указателей на структуры. Описание и инициализация указателя на структуру. Доступ к элементу структуры при помощи указателя.