**1. Трансляция программы.**

Транслятор ***translator*** –– программа или устройство, переводящее текст с одного языка (исходный язык ***source language***) на другой язык (целевой язык ***target language***). Процесс работы транслятора называют трансляцией ***translation***.

Исходный код ***source code*** –– фрагмент текста, написанный на исходном языке для некоторого транслятора, “программа” как текст.

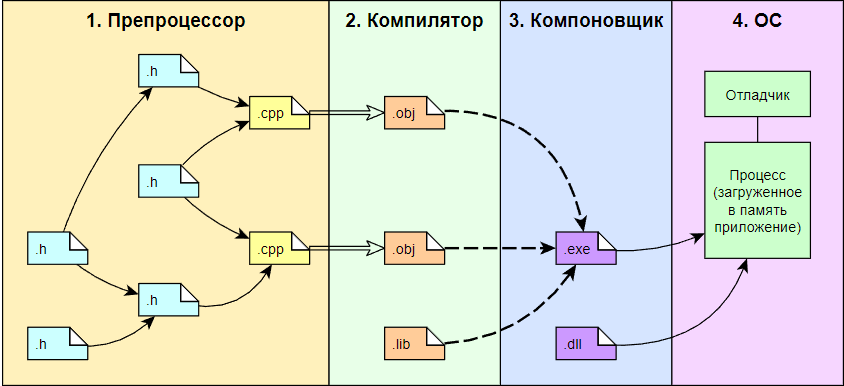
Процесс:  
**0.** Каждая единица трансляции (.cpp и .c файлы) обрабатывается независимо.

**1.** Препроцессор просматривает файл и ищет в нём директивы препроцессора (строки, начинающиеся на знак #) и макросы (идентификаторы, для которых определены замены). Данный процесс может производиться рекурсивно. Например, пусть есть три файла a.h, b.h и hw.cpp. Последний является единицей трансляции, поэтому все операции будут выполняться с ним.

**2.** Компилятор транслирует каждую единицу трансляции в файл объектного кода (традиционно обозначаемые расширением .obj в Windows, .o в Unix-подобных системах). Это самая сложная операция во всём процессе, производящаяся в несколько стадий (фаз трансляции), но на данном этапе мы не будем погружаться в подробности.

**3.** Компоновщик формирует результирующий модуль, соединяя файлы объектного кода и файлы статически-компонуемых библиотек (традиционно обозначаемые расширением .lib в Windows, .a в Unix-подобных системах).

**4.** При запуске исполняемого файла создаётся объект операционной системы — процесс, который представляет собой исполняющееся приложение и распоряжается всеми ресурсами, выделенными приложению операционной системой. В частности, процесс обладает собственной памятью, в которую загружается исходный код исполняемого файла и дополнительных модулей библиотек (динамически-компонуемых библиотек, обозначаемых расширением .dll в Windows, .so в Unix-подобных системах). В случае, если исполняемый файл запускается в режиме отладки, то запущенный процесс контролируется отладчиком (который исполняется в отдельном процессе).



**3. Подключение внешних файлов к проекту (инструкция #include).**

Указывает препроцессору включить содержимое указанного файла в точку, где отображается директива.

## Синтаксис

**#include ""**  
**#include <>**

Форма синтаксиса.

***Кавычки:***

Препроцессор ищет включаемые файлы в слудющем порядке:

1) в том же каталоге, что и файл, содержащий #include инструкцию.  
  
2) в каталогах открытых в данный момент файлов включения в порядке, в котором они были открыты. Поиск начинается в каталоге родительского включаемого файла, а затем выполняется в каталогах всех включаемых файлов-прародителей.  
  
3) вдоль пути, указанного в каждом /I параметре компилятора.  
  
4) вдоль путей, указанных в INCLUDE переменной среды.

***Угловые скобки:***

Препроцессор ищет включаемые файлы в следующем порядке:  
  
1) вдоль пути, указанного в каждом /I параметре компилятора.  
  
2) при компиляции происходит в командной строке вместе с путями, заданными INCLUDE переменной среды.

Если имя файла, заключенное в двойные кавычки, является неполным указанием пути, препроцессор сначала выполняет поиск в каталоге родительского файла. Родительский файл — это файл, содержащий #include директиву. Например, если включить файл с именем file2 в файл с именем file1, то файл file1 будет родительским.

**4-5. Ввод / вывод данных в Си.**

В языке Си нет операторов ввода-вывода.  
Ввод и вывод информации осуществляется через функции стандартной библиотеки. Прототипы рассматриваемых функций находятся в файле stdio.h. Эта библиотека содержит функции

* printf() — для вывода информации. Предназначена для формат.вывода.
* scanf() — для ввода информации. Предназначена для форма.ввода.

Примеры:

Printf:

*int main()  
{  
  int a = 5;  
  float x = 2.78;  
  printf("a=%d\n", a);  
  printf("x=%f\n", x);  
  getchar();  
  return 0;  
}*

Scanf:

*int main()  
{  
  float y;  
  system("chcp 1251"); // переходим в консоли на русский язык  
  system("cls");       // очищаем окно консоли  
  printf("Введите y: "); // выводим сообщение  
  scanf("%f", &y);     // вводим значения переменной y  
  printf("Значение переменной y=%f", y); // выводим значение переменной y  
  getchar(); getchar();  
  return 0;  
}*

**6-7. Ввод / вывод с++**

Поточный ввод-вывод в C++ выполняется с помощью функций сторонних библиотек. В С++, как и в С, нет встроенных в язык средств ввода-вывода.  
  
В С для этих целей используется библиотека stdio.h.  
В С++ разработана новая библиотека ввода-вывода iostream, использующая концепцию объектно-ориентированного программирования:

#include <iostream>

Библиотека iostream определяет три стандартных потока:

* cin стандартный входной поток (stdin в С)
* cout стандартный выходной поток (stdout в С)
* cerr стандартный поток вывода сообщений об ошибках (stderr в С)

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:

* >> получить из входного потока
* << поместить в выходной поток

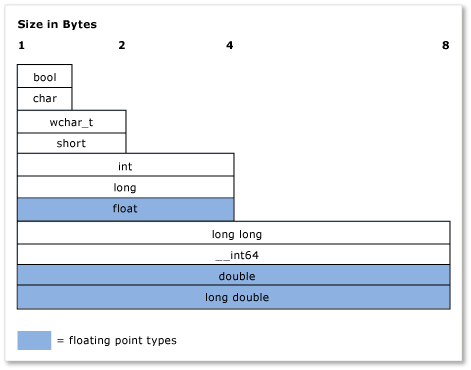
**8. Типы данных.**

Переменная: символическое имя количества данных, чтобы имя можно было использовать для доступа к данным, на которые он ссылается в области кода, где он определен. В C++ переменная обычно используется для ссылки на экземпляры скалярных типов данных, тогда как экземпляры других типов обычно называются объектами.

При объявлении переменной в коде необходимо либо явно указать ее тип, либо использовать auto ключевое слово, чтобы указать компилятору вывести тип из инициализатора.

При объявлении функции в коде необходимо указать тип каждого аргумента и его возвращаемое значение или void значение, если функция не возвращает никакого значения. Исключением является использование шаблонов функции, которые допускают аргументы произвольных типов.

Размерность переменных:



В след.таблице представлены наиболее часто используемые фундаментальные типы данных:

| Тип | Размер | Комментировать |
| --- | --- | --- |
| **int** | 4 байта | Выбор по умолчанию для целочисленных значений. |
| **double** | 8 байт | Выбор по умолчанию для значений с плавающей запятой. |
| **bool** | 1 байт | Представляет значения, которые могут быть или true, или false. |
| **char** | 1 байт | Используйте для символов ASCII в старых строках в стиле C или в объектах std::string, которые никогда не будут преобразовываться в Юникод. |
| **wchar\_t** | 2 байта | Представляет "расширенные" символы, которые могут быть представлены в формате Юникод (UTF-16 в Windows, в других операционных системах возможно другое представление). Это символьный тип, используемый в строках типа std::wstring. |
| **unsigned char** | 1 байт | C++ не имеет встроенного типа Byte. Используется **unsigned char** для представления байтового значения. |
| **unsigned int** | 4 байта | Вариант по умолчанию для битовых флагов. |
| **long long** | 8 байт | Представляет очень большие целочисленные значения. |

**9. Константы. Объявление. Определение типа константы.**

Любой встроенный или пользовательский тип может квалифицироваться ключевым словом const. Кроме того, функции-члены могут быть const полными и даже const перегруженными. Значение const типа не может быть изменено после инициализации.

В С++ предусмотрены два типа констант — литеральные и символьные.

Литеральная константа — это значение, непосредственно задаваемое в программе;

Символьная константа – это целочисленное значение (типа int) представленное в виде символа, заключённого в одинарные кавычки, например 'a'.

Тип и значение такой константы определяется ее внешним видом. Различают следующие виды литеральных констант: числовые, символьные, строковые.

**10. Переменные. Объявление. Совместимость типов.**

Переменная — это «ячейка» оперативной памяти компьютера, в которой может храниться какая-либо информация.

В программировании переменная, как и в математике, может иметь название, состоящее из одной латинской буквы, но также может состоять из нескольких символов, целого слова или нескольких слов.

Для объявления переменной сначала указывается тип переменной, а далее название, после чего можно либо на следующей строке присвоить значение или в другой части кода, либо сразу после объявления при помощи знака равенства „=’’.

Нельзя называть переменную ключевыми словами ЯП. Имя переменной не может начинаться с цифры, не может быть пробелов, а также спец.сиволов вроде: #%/, не может совпадать с ранее заданными переменными, т.к будет являться переопределением переменной.

**11. Область видимости переменной.**

Область видимости переменных — это те части программы, в которой пользователь может изменять или использовать переменные в своих нуждах.

В C++ существуют отдельные блоки, которые начинаются с открывающей скобки ({) и заканчиваются соответственно закрывающей скобкой (}). Такими блоками являются циклы ([for](https://codelessons.ru/cplusplus/syntax/cikl-for-v-c-izuchaem-s-primerami.html), [while](https://codelessons.ru/cplusplus/syntax/cikl-while-v-c-sintaksis-i-igra.html), [do while](https://codelessons.ru/cplusplus/syntax/cikl-do-while-v-c-realizaciya-i-kak-on-rabotaet.html)) и функции.

Если переменная была создана в таком блоке, то ее областью видимости будет являться этот блок от его начала (от открывающей скобки — { ) и до его конца (до закрывающей скобки — } ) включая все дочерние блоки созданные в этом блоке.

Глобальными переменными называются те переменные, которые были созданы вне тела какого-то блока. Их можно всегда использовать во всей вашей программе, вплоть до ее окончания работы.

Локальные переменные — это переменные созданные в блоках. Областью видимости таких переменных является блоки ( и все их дочерние ), а также их область видимости не распространяется на другие блоки. Как ни как, но эти переменные созданы в отдельных блоках.

## Глобальная переменная уступает локальной

Если мы создадим глобальную переменную и с таким же именем локальную, то получится, что там где была создана локальная переменная будет использоваться именно локальная переменная, а не глобальная. Так как локальная переменная считается по приоритету выше глобальной.

Глобальный оператор разрешения — это два подряд поставленные двоеточия, с помощью которых мы говорим компилятору, что хотим использовать глобальную переменную, а не локальную.

***string str = "You lucky!";***

***void message() {***

***string str = "You very lucky man!";***

***cout << :: str;  // использовали глобальный оператор разрешения***

***}***

**12. Основные операторы языка Си.**

*Все операторы языка СИ разделены на категории:*

Условные операторы, к которым относятся оператор условия if и оператор выбора switch  
Операторы цикла (for,while,do while)  
Операторы перехода (break, continue, return, goto)  
Логические операторы (И, ИЛИ, НЕ)  
Другие операторы (оператор “выражение”, пустой оператор)  
**Составные операторы**

**13. Указатели. Объявление и инициализация указателя.**

Указатель – переменная, значением которой является адрес ячейки памяти. То есть указатель ссылается на блок данных из области памяти, причём на самое его начало. Указатель может ссылаться на переменную или функцию. Для этого нужно знать адрес переменной или функции. Так вот, чтобы узнать адрес конкретной переменной в С++ существует унарная операция взятия адреса &. Такая операция извлекает адрес объявленных переменных, для того, чтобы его присвоить указателю.

В основном указатели используются для организации динамического распределения памяти, например при объявлении массива, не надо будет его ограничивать в размере.

//объявление указателя

/\*тип данных\*/ \* /\*имя указателя\*/;

Чтобы получить значение, записанное в некоторой области, на которое ссылается указатель нужно воспользоваться операцией разыменования указателя \*. Необходимо поставить звёздочку перед именем и получим доступ к значению указателя.

**14. Выделение и освобождение памяти в куче.**

В языке программирования C++ оператор new обеспечивает выделение динамической памяти в куче. За исключением формы, называемой «размещающей формой new», new пытается выделить достаточно памяти в куче для размещения новых данных и, в случае успеха, возвращает адрес свежевыделенной памяти. Однако, если new не может выделить память в куче, то он генерирует (throw) исключение типа std::bad\_alloc. Это устраняет необходимость явной проверки результата выделения.

Массивы, созданные (выделенные) при помощи new [], аналогичным образом должны быть уничтожены (оcвобождены) при помощи delete [].

Синтаксис new выглядит следующим образом: p\_var = new typename; где p\_var — ранее объявленный указатель типа typename. typename может подразумевать собой любой фундаментальный тип данных или объект, определенный пользователем (включая, enum, class и struct). Если typename — это тип класса или структуры, то он должен иметь доступный конструктор по умолчанию, который будет вызван для создания объекта.

Вызов delete[] для массива объектов приведет к вызову деструктора для каждого объекта перед освобождением памяти, выделенной под массив. Исключения - возникновение непредвиденных ошибочных ситуаций, например деление на ноль при операциях с плавающей точкой.

**15-16. Функции в языке Си. Описание, реализация, вызов. Формальные и фактические параметры функции.**

*Функция* — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.  
Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

*Сигнатура* функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.  
  
*Семантика* функции определяет способ реализации функции. Обычно представляет собой тело функции.

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

* тип возвращаемого значения;
* имя функции;
* информация о формальных аргументах;
* тело функции.

Определение функции имеет следующий синтаксис:

ТипВозвращаемогоЗначения ИмяФункции(СписокФормальныхАргументов)  
{  
   ТелоФункции;  
  ...  
  return(ВозвращаемоеЗначение);  
}

Различают *системные* (в составе систем программирования) и *собственные* функции.  
  
*Системные* функции хранятся в стандартных библиотеках, и пользователю не нужно вдаваться в подробности их реализации. Достаточно знать лишь их сигнатуру. Примером системных функций, используемых ранее, являются функции printf() и scanf().  
  
*Собственные* функции — это функции, написанные пользователем для решения конкретной подзадачи.

Общий вид вызова функции

*Фактический аргумент* — это величина, которая присваивается формальному аргументу при вызове функции. Таким образом, *формальный аргумент* — это переменная в вызываемой функции, а *фактический аргумент* — это конкретное значение, присвоенное этой переменной вызывающей функцией. Фактический аргумент может быть константой, переменной или выражением. Если фактический аргумент представлен в виде выражения, то его значение сначала вычисляется, а затем передается в вызываемую функцию.

Переменная = ИмяФункции(СписокФактическихАргументов);

**17. Возврат значений из функции.**

Когда функция main() завершает свое выполнение, она возвращает целочисленное значение обратно в операционную систему, используя оператор return.

Функции, которые мы пишем, также могут возвращать значения. Для этого нужно указать тип возвращаемого значения (или «тип возврата»). Он указывается при объявлении функции, перед её именем. Обратите внимание, тип возврата не указывает, какое именно значение будет возвращаться. Он указывает только тип этого значения.

Затем, внутри вызываемой функции, мы используем оператор return, чтобы указать возвращаемое значение — какое именно значение будет возвращаться обратно в caller.

**18-19. Ссылки. Объявление и инициализация ссылок. Передача данных в функцию по имени, ссылке, указателю.**

Ссылки – особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разименовывается. Ссылка может быть объявлена как другим именем, так и как псевдоним переменной, на которую ссылается.

|  |  |
| --- | --- |
|  | // структура объявления ссылок  /\*тип\*/ &/\*имя ссылки\*/ = /\*имя переменной\*/; |

При объявлении ссылки перед её именем ставится символ амперсанда &, сама же ссылка должна быть проинициализирована именем переменной, на которую она ссылается. Тип данных, на который указывает ссылка, может быть любым, но должен совпадать с объектом, на который ссылается, то есть с типом данных ссылочной переменной. Для удобства, будем называть переменную, на которую ссылается ссылка «ссылочной переменной». Любое изменение значения содержащегося в ссылке повлечёт за собой изменение этого значения в переменной, на которую ссылается ссылка.

Ссылки, как правило, в большинстве случаев используют в функциях как ссылки-параметры или ссылки-аргументы. Напомню, что в языке программирования С++ в функции передаются данные по значению и по ссылке.

В чём же разница между указателями и ссылками? Основное назначение указателя – это организация динамических объектов, то есть размер, которых может меняться (увеличиваться или уменьшаться). Тогда как ссылки предназначены для организации прямого доступа к тому, или иному объекту. Главное отличие состоит во внутреннем механизме работы. Указатели ссылаются на участок в памяти, используя его адрес. А ссылки ссылаются на объект, по его имени (тоже своего рода адрес).

Если нет необходимости изменить передаваемое значение в ссылочной переменной, но нужно выиграть в скорости, используйте спецификатор const в объявлении параметров функций. Только так и можно защитить данные от случайного их изменения или полной потере.

|  |  |
| --- | --- |
| 1  2 | **int** sum\_by\_reference(**const** **int** &reference) // функция принимающая аргумент по ссылке  // квалификатор const не даёт изменить передаваемый аргумент внутри функции |

Передача данных по имени, ссылке и указателю:

// reference.cpp: определяет точку входа для консольного приложения.

#include "stdafx.h"

#include <iostream>

**using** **namespace** std;

**int** sum\_by\_value(**int** );// суммирование по значению

**int** sum\_by\_reference(**int** &);// суммирование по ссылке

**int** sum\_by\_pointer(**int** \*); // суммирование по указателю

**int** \_tmain(**int** argc, \_TCHAR\* argv[])

{

**int** value = 10;

cout << "sum\_by\_value     = " << sum\_by\_value(value)     << endl;

    cout << "value = " << value   << endl; // значение переменной осталось неизменным

    cout << "sum\_by\_reference = " << sum\_by\_reference(value) << endl;

    cout << "value = " << value   << endl; // значение переменной изменилось

    cout << "sum\_by\_pointer     = " << sum\_by\_pointer(&value)  << endl;

    cout << "value = " << value   << endl; // значение переменной изменилось ещё раз

**system**("pause");

**return** 0;

}

**int** sum\_by\_value(**int** value)// функция принимающая аргумент по значению

{

    value += value;

**return** value;

}

**int** sum\_by\_reference(**int** &reference) // функция принимающая аргумент по ссылке

{

    reference += reference;

**return** reference;

}

**int** sum\_by\_pointer(**int** \*ptrvalue)// функция принимающая аргумент через указатель

{

    \*ptrvalue += \*ptrvalue;// арифметика с указателем

**return** \*ptrvalue;}



Выше представлен результат работ.

**20. Функции с переменным числом параметров.**

Чтобы передать N’ое количество параметров в функцию, нужно в зону ввода параметров ввести **int fucn( int a, …) { … }.**

Далее придется обращаться к последующим элементам через указатели, но с условием, что они вообще существуют. Проверить это можно через цикл *while* с условием: while( \*P ) { .. }.

Полный код:

**void MyFunc(int a, …) {**

**int\* P = &a;** *// Взятие адреса у первого параметра.*

**while (\*P) {**

**cout << \*P << “ “;**  /*/ Вытаскиваем значение с адреса по месту параметра*

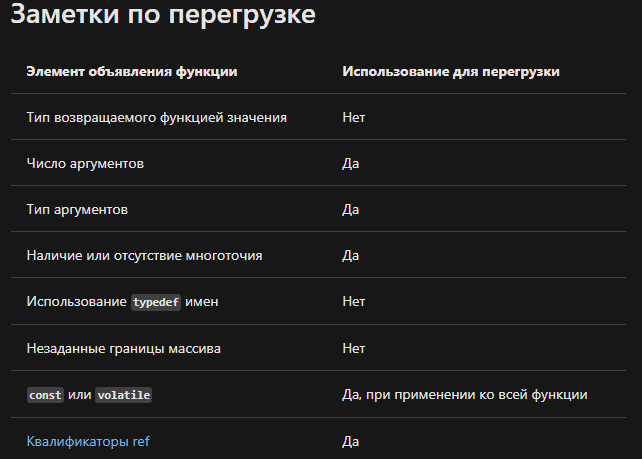
**P++;**  *// Адресная арифметика. Смена текущего адреса на следующий.*

**}**

**}**

Использование данного метода рационально в том случае, если все переданные параметры имеют один и тот же тип, в противном случае будут ошибки, но компилятор не будет выдавать их. Но ошибки не выведется только потому что если компилятор видит эти три точки на месте параметров, то он отключает проверку типов. На самом деле легко увидеть эффект небезопасности работы с функцией, в которой указано, что будет более одного параметра, если первый параметр объявить например как int, (соответственно указатель на первый параметр как int), а во время передачи в функцию написать число с точкой (например 1.0 запустить и потом 11.0 и запустить).

**21. Перегрузки функций.**



C++ позволяет определять несколько функций с одинаковым именем в одной области. Эти функции называются перегруженными функциями.

Перегруженные функции выбираются для оптимального соответствия объявлений функций в текущей области аргументам, предоставленным в вызове функции. Если подходящая функция найдена, эта функция вызывается. Подходящее значение в этом контексте означает:

* Точное соответствие найдено.
* Тривиальное преобразование выполнено.
* Восходящее приведение целого типа выполнено.
* Стандартное преобразование в требуемый тип аргумента существует.
* Пользовательское преобразование (оператор преобразования или конструктор) в требуемый тип аргумента существует.
* Аргументы, представленные многоточием, найдены.

**22. Шаблоны функций.**

Шаблон — это конструкция, которая создает обычный тип или функцию во время компиляции на основе аргументов, предоставленных пользователем для параметров шаблона. Например, можно определить шаблон функции следующим образом:

***template <typename T>***

***T minimum(const T& lhs, const T& rhs) {***

***return lhs < rhs ? lhs : rhs;***

***}***

T является параметром шаблона; ключевое слово говорит о том, что этот параметр является заполнителем для типа. При вызове функции компилятор заменит каждый экземпляр T с конкретным аргументом типа, который либо задается пользователем, либо выведенным компилятором. Процесс, в котором компилятор создает класс или функцию из шаблона, называется созданием экземпляра шаблона. — это экземпляр шаблона minimum<T> .

**23. Указатели на функции.**

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:

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).

### *Массивы указателей на функции*

*Кроме одиночных указателей на функции мы можем определять их массивы. Для этого используется следующий формальный синтаксис:*

|  |  |
| --- | --- |
|  | тип (\*имя\_массива[размер]) (параметры) |

Например:

|  |  |
| --- | --- |
|  | **double** (\*actions[]) (**int**, **int**) |

Здесь actions представляет массив указателей на функции, каждая из которых обязательно должна принимать два параметра типа int и возвращать значение типа double.

**24. Массивы. Статические и динамические.**

Динамический массив — это массив, у которого количество ячеек можно задавать и переменной, и числовой константой. Это большой плюс перед использованием статического массива.

ля работы динамических массивов нам понадобится при инициализации [указатель](https://codelessons.ru/cplusplus/ukazateli-v-c-podrobnoe-rukovodstvo.html) (всего лишь при инициализации!) и уже знакомый конструктор new.

Чтобы создать динамический массив мы будем использовать конструкцию ниже:

|  |  |
| --- | --- |
| 1 | <тип данных> \*<имя массива> = new <тип переменных> [<количество ячеек>]; |

<тип данных> — без разницы какой тип данных тут будет находиться, но лучше тот, который будет совпадать с типом переменных.

<тип переменных> — указанный сюда тип и будут иметь ячейки массива.

<количество ячеек> — здесь мы задаем размер массива (например [n] или [25]).

Динамический массив полностью идентичен обычному массиву, кроме:

* Своей инициализации
* Возможностью своевременно освободить память.

Для удаления динамического массива нам понадобится уже знакомый оператор — delete.

|  |  |
| --- | --- |
| 1 | delete [] <имя массива>; |

Важно запомнить, что квадратные скобки нужно ставить перед <именем массива>.

Создание двумерного массива:

**int \*\*dinamic\_array2 = new int\* [5];   // создаем**

**for (int i = 0; i < 5; i++) {          // двумерный**

**dinamic\_array2[i] = new int [i + 1]; // массив**

**}**

Если вы очищаете двумерный массив, то осуществляется это таким образом:

**for (int i = 0; i < 5; i++) {**

**delete [] dinamic\_array2[i];  // удаляем массив**

**}**

**25. Связь массивов и указателей.**

Массивы и указатели в языке C тесно связаны. Значение переменной типа «массив» является адресом 1-го элемента массива (элемента с нулевым индексом). То есть имя массив

int \*p; /\* указатель на переменную типа int \*/

int a[10]; /\* массив из 10 элементов \*/

после выполнения присваивания:

p = a; /\* указателю p присваивается адрес 1-го элемента массива a \*/

в обеих переменных (p и a) хранится адрес 1-го элемента массива a[0].

Адрес 1-го элемента массива a[0] также можно получить, используя оператор &. Результат следующего присваивания будет тем же:

p = &a[0]; /\* то же самое, что p=a; \*/

**26. Указатели на указатели.**

Обычный указатель типа int объявляется с использованием одной звездочки:

int \*ptr;

Указатель на указатель типа int объявляется с использованием двух звездочек:

int \*\*ptrptr;

Указатель на указатель работает побочно обычному указателю: вы можете его разыменовать для получения значения, на которое он указывает. И, поскольку этим значением является другой указатель, для получения исходного значения надо еще раз разыменовать. Это следует делать последовательно:

**#include <iostream>**

int main()

{

int value = 7;

int \*ptr = &value;

std::cout << \*ptr << std::endl; *// разыменовываем указатель, чтобы получить значение типа int*

int \*\*ptrptr = &ptr;

std::cout << \*\*ptrptr << std::endl;

return 0;

}

Результат:

7

7

Обратите внимание, вы не можете инициализировать указатель на указатель напрямую значением:

int value = 7;

int \*\*ptrptr = &&value; // НЕЛЬЗЯ.

Это связано с тем, что оператор адреса (&) требует [l-value](https://ravesli.com/urok-10-peremennye-initsializatsiya-i-prisvaivanie-v-s/#toc-1), но &value — это r-value. Однако указателю на указатель можно задать [значение null](https://ravesli.com/urok-81-nulevye-ukazateli/):

|  |
| --- |
| int \*\*ptrptr = nullptr; *// используйте 0, если не поддерживается C++11* |

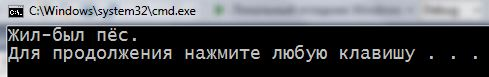
**27. Строка как массив символов.**

Итак о первом: в C++ для хранения строк используют символьные массивы. Это такие же массивы, как мы с вами уже рассматривали в статье о [массивах в С++](https://purecodecpp.com/archives/1), но хранят они не числовые данные, а символьные. Можно представить символы такого массива расположенными последовательно в соседних ячейках памяти – в каждой ячейке хранится один символ и занимает один байт. Один байт потому, что каждый элемент символьного массива имеет [тип char](https://purecodecpp.com/archives/165). Последним символом каждой такой строки является символ \0 (нулевой символ). Например:



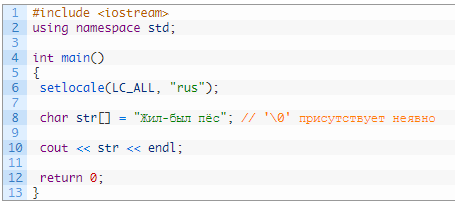
Сам текст, включая пробел, состоит из 11-ти символов. Если бы в последней ячейке находилась например . (точка), а не нулевой символ \0 – для компилятора это уже не строка. И работать с таким набором символов надо было бы, как с обычным массивом – записывать данные в каждую ячейку отдельно и выводить на экран посимвольно (при помощи цикла):

|  |  |
| --- | --- |
|  | #include <iostream>  using namespace std;    int main()  {  setlocale(LC\_ALL, "rus");    char str[12] = {'Ж','и','л','-','б','ы','л',' ','п','ё','с','.'};    for (int i = 0; i < 12; i++)  {  cout << str[i];  }  cout << endl;    return 0;  } |



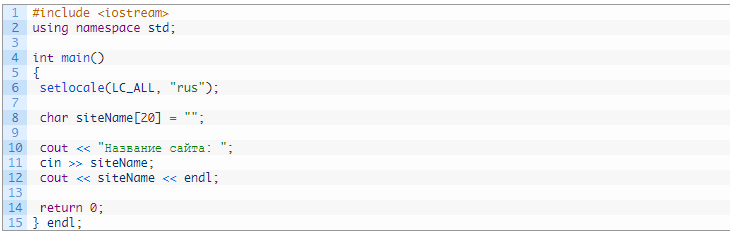
К счастью в C++ есть куда более удобный способ инициализации и обращения к символьным массивам – строкам. Для этого последним символом такого массива обязательно должен быть нулевой символ \0. Именно он делает набор символов строкой, работать с которой, гораздо легче, чем с массивом символов.

Объявляется строка таким образом – создаем массив типа char, размер в квадратных скобках указывать не обязательно (его подсчитает компилятор), оператор = и в двойных кавычках пишем необходимый текст. То есть инициализируем массив строковой константой:



Если произвольно выставить \0 в полученной строке от пользователя строки, то выведется все до первого терминирующего нуля.

Также, чтобы сделать массив символов для ввода с клавиатуры, надо выделить количество ячеек с учетом терминирующего нуля свыше необходимого или вровень.

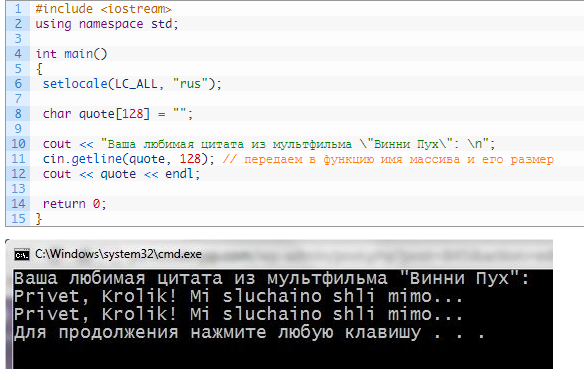
****

Существует проблема, что если при считывании строки через cin есть пробел, то компилятор воспринимает это как конец строки.

Поэтому на выходе получим содержимое до первого пробела.

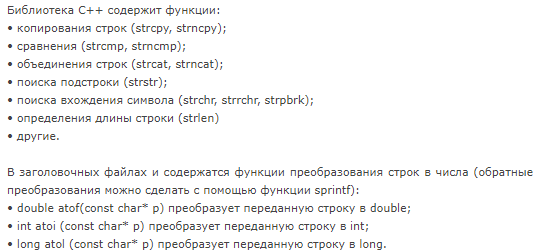
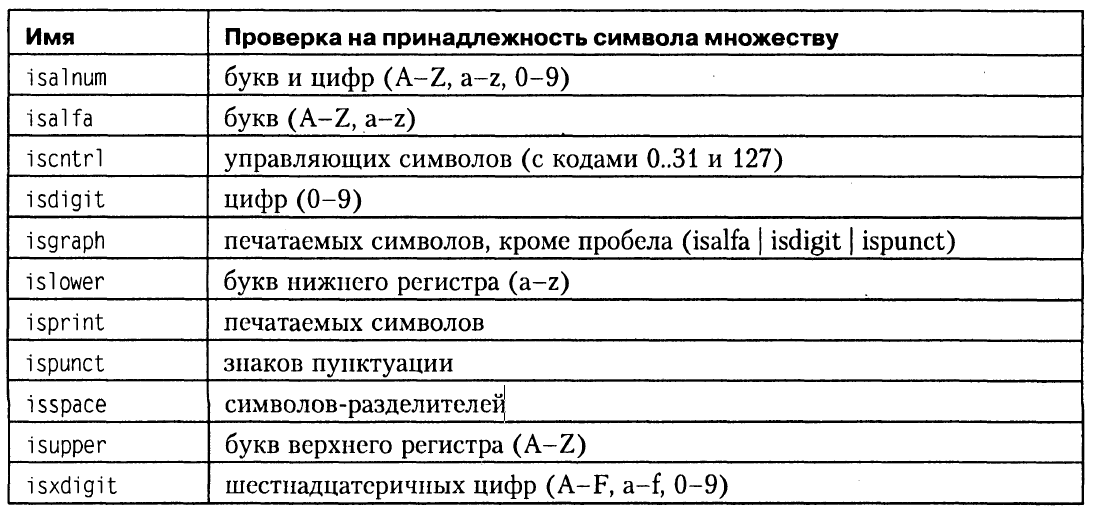
Чтобы избежать это, нужно использовать функцию get() или getline().

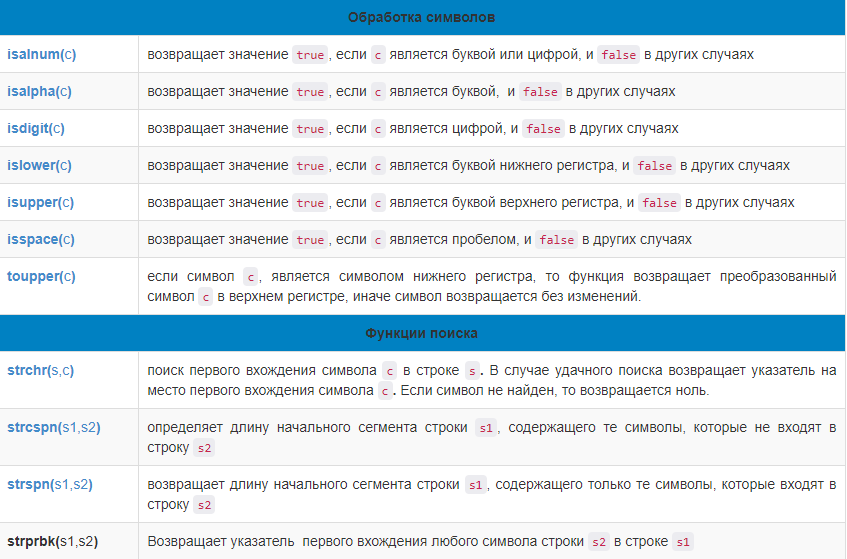
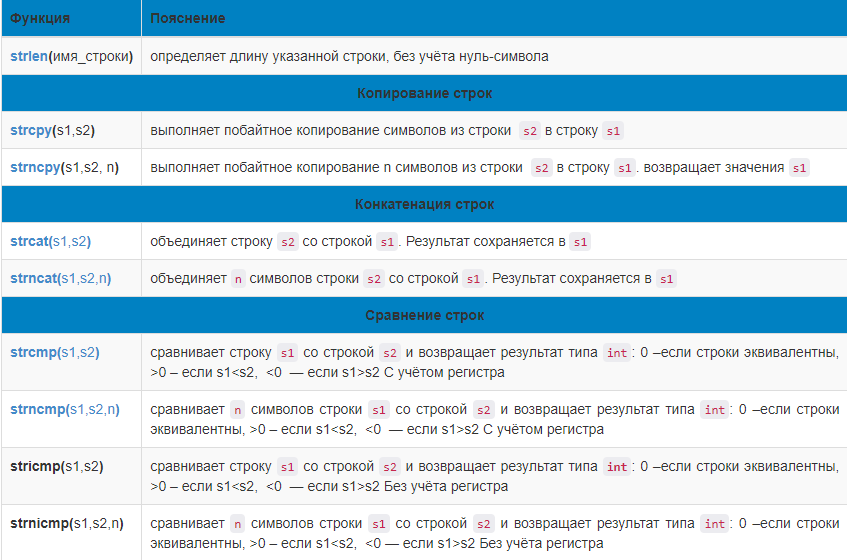
На примере getline():

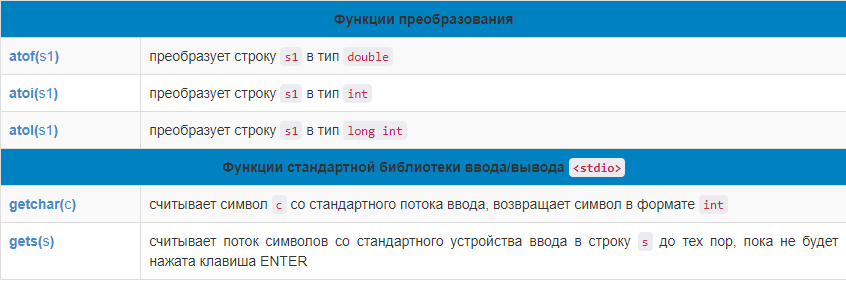


Вы видите, что в скобках мы указали для функции два аргумента – в какой массив считать символы (имя массива) и размер этого массива – строка 11. Результат нам подходит – всё отобразилось правильно. cin.getline() считывает в массив всю строку включая пробелы и табуляции, пока не произойдет нажатие Enter или пока не будет превышен размер массива. Символ новой строки в массиве не сохранится, а заменится на нулевой символ.

**28. Функции для работы со строками.**







Литература: <http://cppstudio.com/post/437/>

**29. Разделение строки на подстроки. (strtok)**

Функция strtok выполняет поиск лексем в строке string. Последовательность вызовов этой функции разбивают строку string на лексемы, которые представляют собой последовательности символов, разделенных символами разделителями.

На первый вызов, функция принимает строку string в качестве аргумента, чей первый символ используется в качестве начальной точки для поиска лексем. В последующие вызовы, функция ожидает нулевого указателя и использует позицию сразу после окончания последней лексемы как новое местонахождение для сканирования.

Для определения начала лексемы функция сначала определяет символы, не содержащиеся в строке delim, то есть они являются символами разделителями. А затем посимвольно проверяет остальную часть строки до первого символа-разделителя, который сигнализирует конец лексемы.

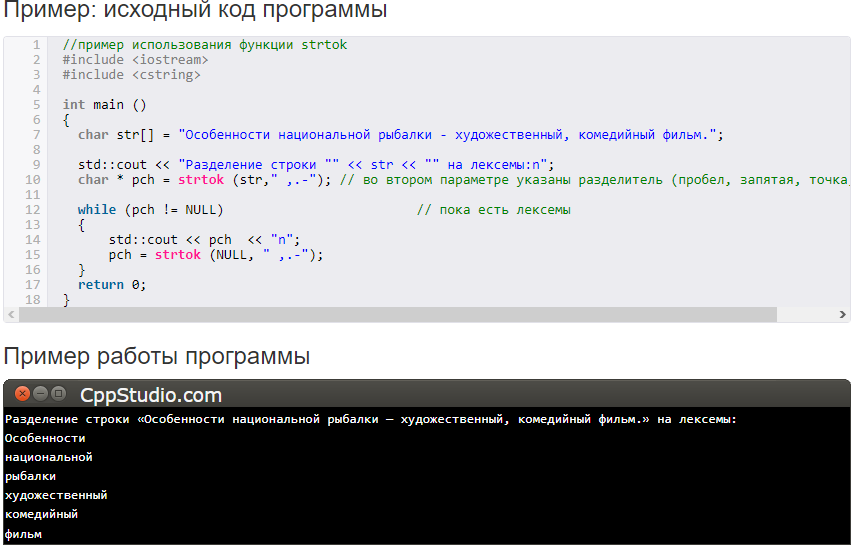
Этот конечный маркер автоматически заменяется нулевым символом, и лексема возвращается функцией. После этого, следующие вызовы функции strtok начинаются с этого нулевого символа.

### Параметры:

* string  
  Строка для поиска в ней лексем. Содержание этой строки будет изменено, она разбивается на более мелкие строки (лексемы). Данный параметр может содержать нулевой указатель, в этом случае функция продолжает сканирование с того места, где был остановлен предыдущий успешный вызов функции.
* delim  
  Строка, содержащая разделители. Они могут варьироваться от одного вызова к другому вызову функции.

### Возвращаемое значение

Указатель на последнюю найденную лексему в строке.  
Возвращается пустой указатель, если нет найденных лексем.



**30-31. Работа с файлами. Различные режимы открытия файлов. (Стили Си и Си++). Чтение и запись.**

Для работы с файлами следует подключить заголовочный файл <fstream>. В нем определены несколько классов и подключены заголовочные файлы <ifstream> - файловый ввод и <ofstream> - файловый вывод.

Файловый ввод/вывод аналогичен стандартному вводу/выводу. Единственное отличие — все это происходит непосредственно с/в файле.

Если необходимо создать текстовый файл и записать в него строку. Для этого нужно:

1. Создать объект класса *ofstream*.

2. Связать объект класса с файлом, в который будет производиться запись;

3. Записать строку в файл;

4. Закрыть ф айл.

Почему необходимо создавать объект класса ofstream, а не класса ifstream? Потому, что нужно сделать запись в файл, а если бы нужно было считать данные из файла, то создавался бы объект класса ifstream.

ofstream *\*имя объекта\**;

Назовет объект fout. → ofstream fout;

Далее: fout.open(“cppstudio.txt”); // создается файл с таким названием.

Запись: fout << “Работает!”;

Закрытие: fout.close();

Для чтения алгоритм:

1. Создать объект класса ifstream и связать с файлом, из которого будет производиться считывание.

2. Прочитать файл.

3. Закрыть файл.

**char** buff[50]; // буфер промежуточного хранения считываемого из файла текста

ifstream fin("cppstudio.txt"); // открыли файл для чтения

    fin >> buff; // считали первое слово из файла

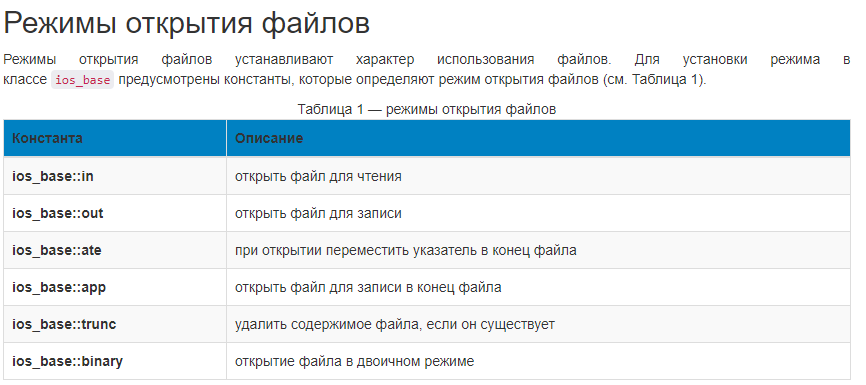
    cout << buff << endl; // напечатали это слово

    fin.getline(buff, 50); // считали строку из файла

    fin.close(); // закрываем файл

    cout << buff << endl; // напечатали эту строку





Режимы открытия файлов можно устанавливать непосредственно при создании объекта или при вызове функции open().

ofstream fout("cppstudio.txt", ios\_base::app); // открываем файл для добавления информации к концу файла

fout.open("cppstudio.txt", ios\_base::app); // открываем файл для добавления информации к концу файла

Режимы открытия файлов можно комбинировать с помощью поразрядной логической операции или |, например: ios\_base::out | ios\_base::trunc — открытие файла для записи, предварительно очистив его.

Объекты класса ofstream, при связке с файлами по умолчанию содержат режимы открытия файлов ios\_base::out | ios\_base::trunc. То есть файл будет создан, если не существует. Если же файл существует, то его содержимое будет удалено, а сам файл будет готов к записи. Объекты класса ifstream связываясь с файлом, имеют по умолчанию режим открытия файла ios\_base::in — файл открыт только для чтения. Режим открытия файла ещё называют — флаг, для удобочитаемости в дальнейшем будем использовать именно этот термин. В таблице 1 перечислены далеко не все флаги, но для начала этих должно хватить.

**32. Управление курсором.**

|  |  |
| --- | --- |
| value | offset is relative to... |
| ios\_base::beg | beginning of the stream |
| ios\_base::cur | current position in the stream |
| ios\_base::end | end of the stream |

tellg – возвращает текущую позицию в файле для чтения.

Потоки входного файла хранят внутренний указатель на позицию в файле, с которой будет продолжаться чтение. Этот указатель можно установить функцией seekg

Функции seekg() и seekp() принимают следующие два параметра:

первый параметр — это смещение на которое следует переместить файловый указатель (измеряется в байтах);

второй параметр — это флаг [ios](https://ravesli.com/urok-209-funktsional-klassov-ostream-i-ios-formatirovanie-vuvoda/), который обозначает место, от которого следует отталкиваться при выполнении смещения.

Флаги ios, которые принимают функции seekg() и seekp() в качестве второго параметра:

beg — cмещение относительно начала файла (по умолчанию);

cur — cмещение относительно текущего местоположения файлового указателя;

end — смещение относительно конца файла.

Положительное смещение означает перемещение файлового указателя в сторону конца файла, тогда как отрицательное смещение означает перемещение файлового указателя в сторону начала файла. Например:

inf.seekg(15, ios::cur); *// перемещаемся вперед на 15 байт относительно текущего местоположения файлового указателя*

inf.seekg(-17, ios::cur); *// перемещаемся назад на 17 байт относительно текущего местоположения файлового указателя*

inf.seekg(24, ios::beg); *// перемещаемся к 24-му байту относительно начала файла*

inf.seekg(25); *// перемещаемся к 25-му байту файла*

inf.seekg(-27, ios::end); *// перемещаемся к 27-му байту от конца файла*

Перемещение в начало или в конец файла:

|  |  |
| --- | --- |
| 1  2 | inf.seekg(0, ios::beg); *// перемещаемся в начало файла*  inf.seekg(0, ios::end); *// перемещаемся в конец файла* |

**34. Структурированные типы данных (структуры) С++.**

Структура — это агрегатный тип данных, так как может содержать в себе разнотипные элементы.

struct Name

{

 type atrib;

    // остальные элементы структуры

} structVar1, structVar2, ...;

где,

* struct — ключевое слово, которое начинает определение структуры
* Name — имя структуры
* type — тип данных элемента структуры
* atrib — элемент структуры
* structVar1-2 — структурные переменные

Объявление структуры всегда должно начинаться с ключевого слова struct. Необязательно, чтобы структура имела имя, но тогда такая структура обязательно должна иметь структурные переменные, объявленные между закрывающей фигурной скобкой и точкой с запятой, строка 5. Обязательно в объявлении структуры должны присутствовать фигурные скобочки, они обрамляют тело структуры, в котором объявляются её атрибуты (элементы), строка 3. Структурные переменные, при объявлении структуры, указывать необязательно, строка 5.

Чтобы объявить указатель на структуру, в С++ вы просто перед именем структурной переменной ставите символ указателя — \*.

В С++:

|  |  |
| --- | --- |
| 1 | structName \*structVar; // указатель на структуру structName |

#### Доступ к элементам структуры

Доступ к элементам структуры так же прост, как использование символа «точка». Предположим. что у нас есть структурная переменная с именем car и у нее есть элемент с именем speed, к которому, мы сейчас получим доступ:

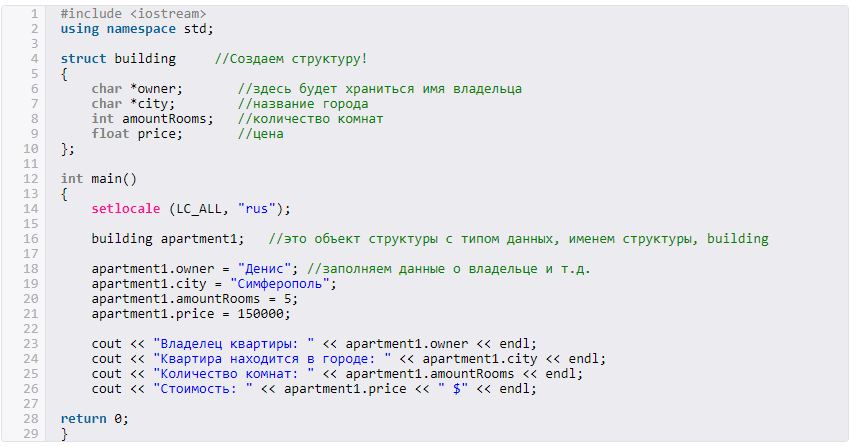
|  |  |
| --- | --- |
| 1 | car.speed; |

#### Доступ к элементам указателя на структуру

Чтобы получить доступ к элементам структуры, через указатель на структуру, вместо оператора «точка», используйте оператор стрелка ->:

|  |  |
| --- | --- |
| 1 | carPtr->speed; |

Пример структуры:



Что ещё важно знать:

* Объект структуры можно объявить до функции main(). Это выглядело бы так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **struct** building  {  **char** \*owner  **char** \*city;  **int** amountRooms;  **float** price;  }apartment1; //объявление объекта типа building |

Инициализировать структуру можно и таким способом:

|  |  |
| --- | --- |
| 1 | building apartment1 = {"Денис", "Симферополь", 5, 150000}; |

# 35. Обработка двоичных файлов в C++

Для того чтобы записать данные в двоичный файл, необходимо:

1. описать файловую переменную типа FAIL \* с помощью оператора FILE \*filename, здесь filename — имя переменной, где будет храниться указатель на файл.
2. открыть файл с помощью функции fopen
3. записать информацию в файл с помощью функции fwrite
4. закрыть файл с помощью функции fclose

Для того чтобы считать данные из двоичного файла, необходимо:

1. описать переменную типа FILE \*
2. открыть файл с помощью функции fopen
3. считать необходимую информацию из файла с помощью функции fread, при этом следить за тем достигнут ли конец файла.
4. закрыть файл с помощью функции fclose

Рассмотрим основные функции, необходимые для работы с двоичными файлами.

Для открытия файла предназначена функция fopen.

FILE \*fopen(const \*filename, const char \*mode)

Здесь filename — строка, в которой хранится полное имя открываемого файла, mode — строка, определяющая режим работы с файлом; возможны следующие значения:

* «rb» — открываем двоичный файл в режиме чтения;
* «wb» — создаем двоичный файл для записи; если он существует, то его содержимое очищается;
* «ab» — создаем или открываем двоичный файл для дозаписи в конец файла;
* «rb+» — открываем существующий двоичный файл в режиме чтения и записи;
* «wb+» — открываем двоичный файл в режиме чтения и записи, существующий файл очищается;
* «ab+» — двоичный файл открывается или создается для исправления существующий информации и добавления новой в конец файла.

Функция возвращает в файловой переменной f значение NULL в случае неудачного открытия файла. После открытия файла доступен 0-й его байт, указатель файла равен 0, значение которого по мере чтения или записи смещается на считанное (записанное) количество байт. Текущие значение указателя файла — номер байта, начиная с которого будет происходить операция чтения или записи.

Для закрытия файла предназначена функция fclose:

int fclose(FILE \*filename);

Она возвращает 0 при успешном закрытие файла и EOF в противном случае.

Функция remove предназначена для удаления файлов:

int remove(const char \*filename);

Эта функция удаляет с диска файл с именем filenema. Удаляемый файл должен быть закрыт. Функция возвращает ненулевое значение, если файл не удалось удалить.

Для переименования файлов предназначена функция rename:

int rename(const char \*oldfilename, const char \*newfilename);

Первый параметр — старое имя файла, второй — новое. Возвращает 0 при удачном завершении программы.

Чтение из двоичного файла осуществляется с помощью функции fread:

fread(void \*ptr, size, n, FILE \*filename);

Функция fread считывает из файла filename в массив ptr n элементов размера size. Функция возвращает количество считанных элементов. После чтения из файла его указатель смещается на n\*size байт.

Запись в двоичный файл осуществляется с помощью функции fwrite:

fwrite(const void \*ptr, size, n, FILE \*filename);

Функция fwrite записывает в файл filename из массива ptr n элементов размера size. Функция возвращает количество записанных элементов. После записи информации в файл указатель смещается на n\*size байт.

Для контроля достижения конца файла есть функция feof:

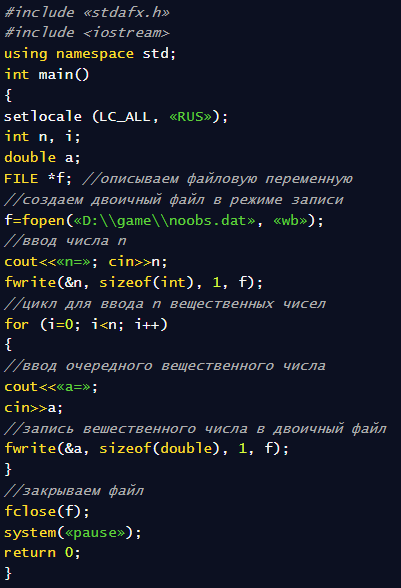
int feof(FILE \*filename);

Она возвращает ненулевое значение если достигнут конец файла.

### Задача 1

Создать двоичный файл D:\\game\\noobs.dat и записать в него целое число n и n вещественных чисел.

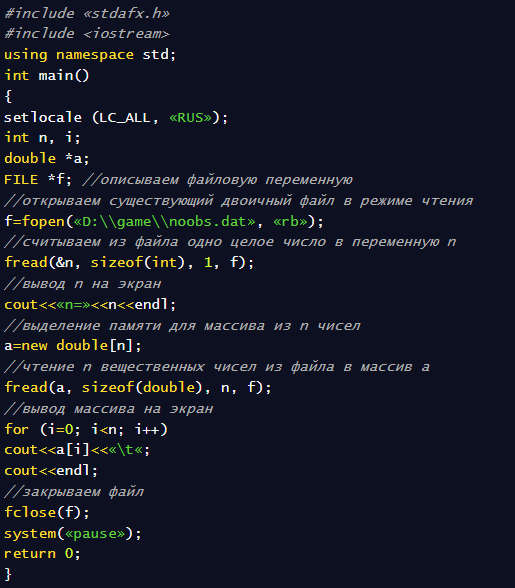
Решение:



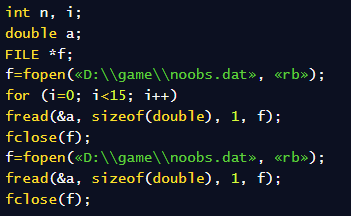
### Задача 2

Вывести на экран содержимого созданного в прошлой задаче двоичного файла D:[\\game\\noobs.dat](file:///\\game\noobs.dat)

Решение:



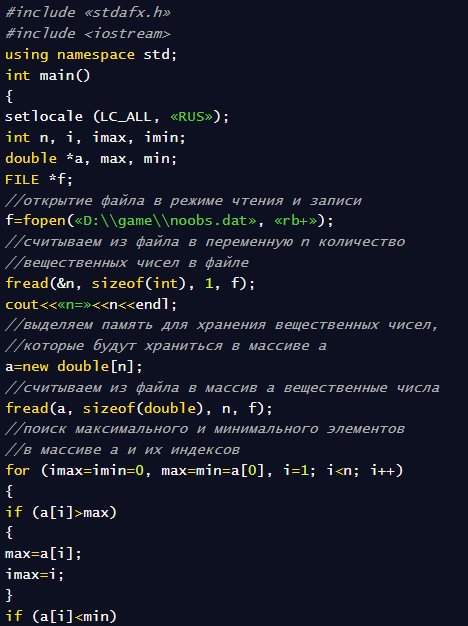
Двоичный файл — последовательная структура данных, после открытия файла доступен первый байт, хранящийся в нем. Можно последовательно записывать или считывать данные из файла. Допустим, необходимо считать пятнадцатое число, а затем первое. С помощью последовательного доступа это можно сделать следующим способом:



### Задача 3

В созданном раннее двоичном файле D:\\game\\noobs.dat, поменять местами наибольшее и наименьшее из вещественных чисел.

Алгоритм решения задачи состоит из следующих этапов:

1.  Чтение вещественных из файла в массив a.
2. Поиск в массиве а максимального (max) и минимального (min) значения и их номеров (imax, imin).
3. Перемещения указателя файла к максимальному значению и запись min.
4. Перемещения указателя файла к минимальному значению и запись max.

Ниже приведен текст программы решения задачи с комментариями.

