C++: Basics

# Stepik: Программирование на языке C++

# [Программа курса – Программирование на языке C++ – Stepik](https://stepik.org/course/7/syllabus)

* [Online C++ Compiler - online editor (onlinegdb.com)](https://www.onlinegdb.com/online_c++_compiler)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# interface info

ctrl + shift + v - вставка без форматирования

## Комментирование

комментирование “//”

выделить фрагмент, нажать ctrl + X, удерживая ctrl нажать новую Y:

ctrl k - c - комментирование

ctrl k - u - раскомментирование

* можно просто выделить фрагмент + / и повторно для раскомментирования

ctrl + shift + / - комментирование “/\* - \*/”

* можно просто выделить фрагмент + \* и повторно для раскомментирования

# first screen

// #include <stdio.h> - не идентифицирует cin cout

**#include<iostream>**

using namespace std;

int main()

{

setlocale(LC\_ALL, "ru");

printf ("Hello world\n");

printf ("Hello world\n");

return 0;

}

После return функция не выполняется.

#include <stdio.h> подключение стандартных функций вроде printf.

\_\_\_\_\_\_\_\_\_\_

# Организация h файлов проекта

## Стандартный подход

#include< > - стандартная библиотека

#include” ” - локальная библиотека (из папки проекта)

1. основной cpp main файл ( #include “x.h”)
2. создание h файла “x.h” с объявлениями функций
3. cоздание cpp файла “func.cpp” с определениями функций ( #include “x.h”)

В файле main.h вызывается функция sum (func.cpp) для вычисления суммы чисел. Но перед использованием функции sum она должна быть определена или по крайней мере должен быть известен ее заголовок. Однако если функции определены в отдельных файлах, то более оптимально помещать объявления функций в специальные заголовочные файлы и потом подключать эти файлы. Именно поэтому в начале с помощью директивы include подключается файл sum.h, который содержит объявление или заголовок функции.

Можно было бы и не подключать файл x.h и вообще не создавать его, а объявление функции поместить непосредственно в файл main.cpp. Но при изменении функции может потребоваться изменить и ее объявление. И если функция sum используется в нескольких файлах с исходным кодом, то в каждом из этих файлов придется менять ее объявление. В данном же случае достаточно изменить объявление функции в одном файле - x.h.

### Вопросы

#### как это работает?

1) На этапе компиляции(все файлы компилируется независимо) нужно, чтобы в файле перед использованием функций была декларация, чтобы компилятор знал сигнатуру функции и понимал как обращаться с функцией. → Подключают h файлы.

\* в файле engine.cpp перед определением функции подключают header.h для удобства и предотвращению ошибок.

2) На следующем этапе - линковке - компилятор совмещает все файлы программы и соединяет вызовы функций и их определения. → поэтому определения функции никуда подключать необязательно

#### include h файл к engine.cpp

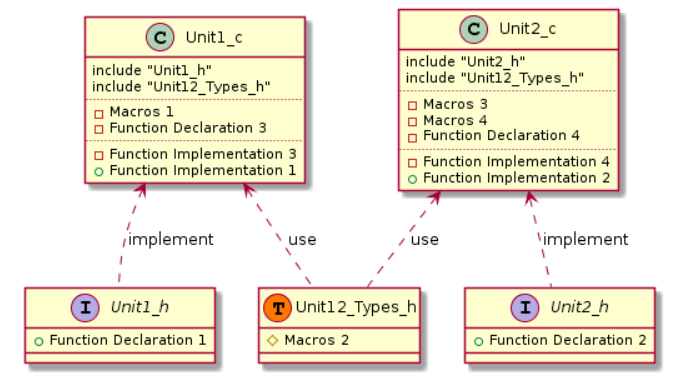
Подключать header.h к engine.cpp необязательно, всё будет работать, НО

**подключать заголовочный файл к файлу определений функций** (например, engine.cpp) — это **рекомендуемая практика**, и вот почему:

### **Причины подключать заголовочный файл в файле с определениями:**

1. **Проверка соответствия объявлений и определений**:
   * Когда вы подключаете заголовочный файл в файл с определениями, компилятор сверяет декларации (объявления) функций с их определениями. Это помогает убедиться, что сигнатуры функций (аргументы, тип возвращаемого значения) совпадают, и избегает ошибок при изменении интерфейса функции.
2. **Упрощение изменений**:
   * Если вы измените объявление функции в заголовочном файле (например, измените тип аргумента или добавите новый параметр), но забудете изменить это в файле реализации, компилятор выдаст ошибку. Это облегчает поддержку кода и уменьшает вероятность ошибок.
3. **Переиспользование кода**:
   * Заголовочные файлы могут содержать не только объявления функций, но и другие элементы, такие как константы, типы данных, макросы и т.д., которые могут понадобиться в файле реализации. Вместо того чтобы повторно объявлять их в файле реализации, проще подключить заголовочный файл.

## Какой то другой подход



## Классы

К счастью, C++ предоставляет способ отделить часть «объявления» класса от его «реализации». Это делается путем определения функций-членов класса вне определения класса. Для этого просто определите функции-члены класса, как если бы они были обычными функциями, но добавив к функциям префикс из имени класса с помощью оператора разрешения области видимости (::) (так же, как для пространства имен).

Вот наш класс Date с конструктором Date и функцией setDate(), определенными вне определения класса. Обратите внимание, что прототипы этих функций всё еще присутствуют внутри определения класса, но фактическая реализация вынесена за его пределы:

Можно и public и private функции писать так:

class Date

{

private:

int m\_year;

int m\_month;

int m\_day;

public:

Date(int year, int month, int day);

void setDate(int year, int month, int day);

int getYear() { return m\_year; }

int getMonth() { return m\_month; }

int getDay() { return m\_day; }

};

//header.h:

// Конструктор Date

Date::Date(int year, int month, int day)

{

SetDate(year, month, day);

}

// Функция-член Date

void Date::SetDate(int year, int month, int day)

{

m\_month = month;

m\_day = day;

m\_year = year;

}

## Мой подход с классами

1) Если класс используется только 1 раз и нам нужно его разрабатывать.

main.cpp ()

class.h (подключаем декларацию функций → include header.h)

engine.cpp (декларация перед определением → include header.h + include class.h)

header.h

2) полное погружение

main.cpp (include header.h)

engine.cpp (include header.h)

header.h

main.cpp - прописываем реализацию функций и классов в программе.

(ставим декларацию перед определением → include header.h).

engine.cpp - прописываем определения функций для классов как “int NameClass::func()” (ставим декларацию перед определением → include header.h).

header.h - прописываем определения функций и классов, других структур данных.

## повторное включение файла (#pragma once)

Исключение повторного включения (повторного копирования кода определения функции):

1 макросы препроцессора

#ifndef func\_bar

(пока флаг функции не определен, если флаг==0 программа спокойно идёт дальше)

#define func\_bar

(Поднятие флага функции, запрещает следующее копирование функции, тк флаг==1 уже поднят, что значит функция уже скопирована, дальше не выполняется операция)

Void bar() {return 0;}

#endif

(Выход из if)

2 просто использовать в заголовочном файле с функцией в начале файла

#pragma once (файл будет включаться один раз)

На этапе компиляции

# #include<>

1) Нужно прописывать include<> для ВСЕХ файлов, только если ты не прописал include”” и в нем include<>

2) Лучше прописывать include”” ДО include<>

#include<vector>

#include<string>

#include<iostream>

#include"Header.h"

#include "MyForm.h"

# Namespaces

## Global var в namespace

::a - явное указание на глобальную переменную.

пример

namespace povtorenie {

void some\_function() { read\_data(::data);

}

}

также возможно взятие адреса:

&::data - взятие адреса переменной ::data,

## Проблемы

если прописать using namespace x;

то все переменные и функции по умолчанию будут определяться как x::func/ x::var

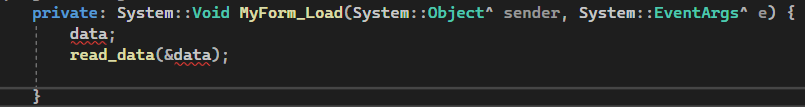
ЕСЛИ ФУНКЦИЯ/ПЕРЕМЕННАЯ/СТРУКТУРА объявлена за пределами namespace x, то она будет недостижима в namespace x, если не прописать differentNamespce::func/var/struct

Пример этой проблемы.

unordered\_map<string, UGOdata> data;

int read\_data(unordered\_map<string, UGOdata>\* data);

namespace povtorenie {



# prog info

## exe

exe лежит в x64 если открыть проект в проводнике, после компиляции

Нужно скомпилировать проект как release, можно выбрать в верхней панели visual studio - debug/release.

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# \*Основы компиляции и архитектуры

1. для запуска на архитектуре X не обязательно нужно иметь саму архитектуру, а достаточно лишь установить компилятор для этой архитектуры.

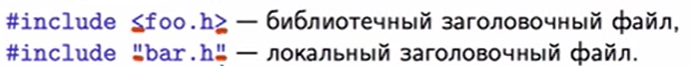
2. Программу необязательно каждый раз компилировать на архитектуре, если она предварительно уже была скомпилирована под эту архитектуру

3. Для интерпретируемых языков компиляция не нужна, нужен лишь интерпретатор этого языка, верно для любой архитектуры

4. Для языка под виртуальную машину также не нужна компиляция, а лишь компиляция один раз под виртуальную машину. Для запуска нужна лишь установленная виртуальная машина

\_\_\_\_\_\_\_\_\_\_

# Компиляция проекта



1. Выполнение директив # препроцессора на отдельном языке

2. Непосредственно компиляция С++ с учетом директив 1. (компилируются только cpp, h компилируется только как вставка в сpp) + каждый файл отдельно.

На выходе компиляции получается новый .o или .obj бинарный файлы 001001100 для процессора для каждого файла в отдельности

3. Линковка - соединение всех бинарных файлов в один исполняемый. (важна точка входа - int main())

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

чтобы консоль не сразу закрывалась

system ("pause");

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Менеджмент файлов в проекте и solution’ e

0 solution - сборка проектов

1 создание нового файла - сохранение нового файла где-нибудь (не отражается в explorer)

* файл нужно сохранить чтобы работы с ним

2 создание проекта ( шаблонной структуры)

* в explorer нужно создать структуру чтобы работать менеджментом файлов

(просто файлы без структуры не отражаются)

Если в solution есть несколько проектов со своими main(), то есть возможность назначать стартовый проект (set as start) - ПКМ по проекту.

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Переменные

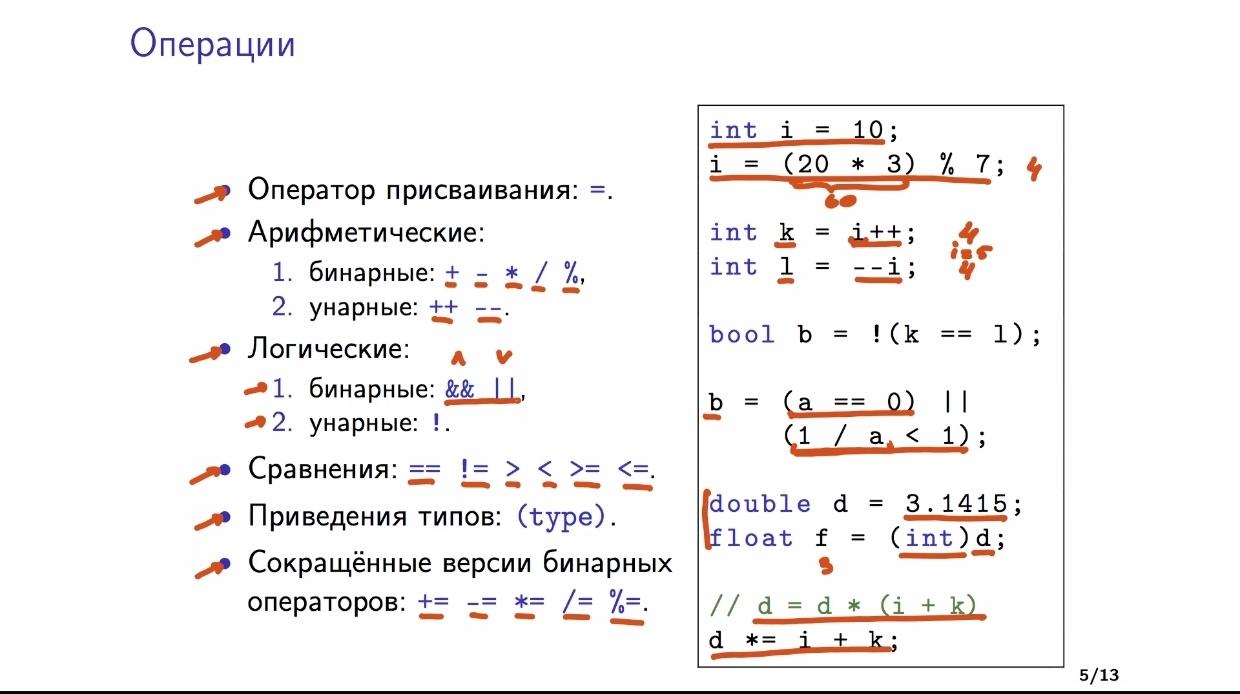
## стандрт. операторы переменных

Short i = 20;

По умолчанию, указывая 20 в с++ имеет тип int, но он автоматически преобразуется в short.

Если указать число не входящее в рамки short - short i= 7000000 - то оно тоже преобразуется в short, но некорректно.

Лучше устанавливать значение переменной сразу, тк это поможет в редактируемом и кода и читабельности



b = true , если a==0 или (1/a) <1

### % оператор

остаток от целочисленного деления

Оператор остатка от целочисленного деления (%) в языке программирования C++ возвращает остаток от деления одного числа на другое. Например, в выражении "a % b", оператор % вернет остаток от деления числа a на число b.

Если результат деления без остатка, то оператор % вернет остаток 0. Например, 10 % 5 даст результат 0, потому что 10 делится нацело на 5.

Однако, если результат деления имеет остаток, то оператор % вернет именно этот остаток. Например, 10 % 3 даст результат 1, потому что 10 делится на 3 нацело 3 раза с остатком 1.

a%b

xb = X // деление нацело

a - X = остаток единиц

## Инициализация переменной:

Инициализация переменных в C++ может быть проведена двумя способами, как вы указали:

double i = 0;

* в этом случае значение переменной i устанавливается явно, используя оператор присваивания = и указывая значение 0.

double i(0); // или int {12};

* в этом случае инициализация переменной i происходит с использованием синтаксиса конструктора, где значение 0 передается в круглых скобках.

## Объявление и определение:

Мне кажется, что стоит добавить в первый список пункт про extern и прописать в нужных объявлениях, что это еще и определения.

*Определение - содержит всю информацию, необходимую компилятору, чтобы выделить память для хранения объекта.*

* **extern int a; — объявление переменной типа int. |** глобальная переменная, определенная позже в программе (int a - локальная переменная, определенная ниже в программе)
* int a; — объявление **и определение** переменной типа int,
* void foo(); — объявление функции с именем foo,
* void bar() { foo(); } — объявление **и определение** функции с именем bar.

/\* extern int a - это объявление переменной a, которая уже была определена в другом месте программы. Она определяет ссылку на переменную a, которая уже существует в другом месте программы. Таким образом, переменная a может использоваться в текущем блоке так, как будто она была объявлена в нем, но ее определение инициализации происходит в другом блоке.

const int a - это объявление целочисленной константы a, которая не может быть изменена внутри блока, в котором она объявлена. Значение константы a должно быть установлено при ее объявлении и не может быть изменено в последующем выполнении программы. Попытка изменения значения константы приведет к ошибке компиляции. \*/

\*Таким образом, разница между "int a" и "extern int a" заключается в том, что первое выражение создает новую переменную "a" и резервирует для нее память, а второе выражение только объявляет переменную "a" без резервирования памяти для нее.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

const char\* str = "Hello";

char\* strTemp = const\_cast<char\*>(str); // Приведение типа для удаления const

В этом примере мы используем const\_cast для удаления квалификатора const и приводим указатель str к типу char\*, чтобы можно было изменять его значение.

По логике курса:

1 ОБЪЯВЛЕНИЕ функции должны быть в хэдере.

2. Определение функции в отдельном файле.

3. Реализация функции в рабочем файле.

Если мы изменим определение функции, то нам нужно будет менять ВСЕ объявления в разных файлах => вместо этого подключаем include хэдера с объявлением. Тогда при изменении определения мы должны изменить лишь в одном месте объявление.

У нас есть два независимых файла main.cpp и func.cpp. И в мейне мы используем функцию из func.

Для того, чтобы скомпилировать main нужно сделать объявление func( указать, что позже функция будет определена) через инклюд хэдера с объявлениями.

Тогда мы скомпилировали 2 файла.

На этапе и линковки они превращаются в один и функции одной сигнатуры соединяет свои объявл. и опред.

Компиляция мейна: читает объявление и реализацию

Линк: соединяет объявление и определение

тогда

1.определения.cpp

2. объявления хэдер.h

3. реализация рабочий файл.cpp +include 2.

\*Отличия .h от .cpp файлов: сурс не определен.

(компилируются только cpp, h компилируется только как вставка в сpp)

## Разные СИ инициализация

Восьмеричная система (octal numbers)

int x {012} ;

Шестнадцатеричная система (Hexadecimal):

int x{ 0xF };

Двоичная система (**Binary literals)**

**int bin = 0b1;**

вывод для разных СИ:

**int x { 12 };**

**std::cout << x << '\n'; // decimal (by default)**

**std::cout << std::hex << x << '\n'; // hexadecimal**

**std::cout << x << '\n'; // now hexadecimal**

**std::cout << std::oct << x << '\n'; // octal**

**std::cout << std::dec << x << '\n'; // return to decimal**

**std::cout << x << '\n'; // decimal**

**Вывод для лвоичной СИ**

**#include <bitset> // for std::bitset**

**#include <iostream>**

**int main()**

**{**

**// std::bitset<8> means we want to store 8 bits**

**std::bitset<8> bin1{ 0b1100'0101 }; // binary literal for binary 1100 0101**

**std::bitset<8> bin2{ 0xC5 }; // hexadecimal literal for binary 1100 0101**

**std::cout << bin1 << '\n' << bin2 << '\n';**

**std::cout << std::bitset<4>{ 0b1010 } << '\n'; // create a temporary std::bitset and print it**

**return 0;**

**}**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Стандартные процедуры (for,while,if,switch)

Int интерпретируется в булевые значения как: 0 - false

1,2,3,4 …. - true

Цикл for

стандартное условие (int i=0; i<a; ++i) // ++i и i++ без разницы

означает, что цикл выполнится ровно а раз (учитывая 0 цикл)

при этом i будет означать номер итерации.  
  
В цикле for сначала выполняется инициализация, затем проверка условия, а затем выполняется тело цикла. В вашем примере инициализация i устанавливает его значение в 0 (int i = 0), а затем в условии i < SIZE проверяется, является ли i меньшим, чем значение константы SIZE. Если условие истинно, то выполняется тело цикла, а затем инкрементируется i с помощью выражения i++.

0. ИНКРЕМЕНТЫ В ФУНКЦИЯХ ++i : иногда инкременты не работают сразу в функциях, но всегда работает ( i + 1)

1.СЛОЖНЫЕ УСЛОВИЯ делать в скобках

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

цикл while

int a, b, r;

while (fscanf\_s(ptr, "%d %d", &a, &b) != EOF)

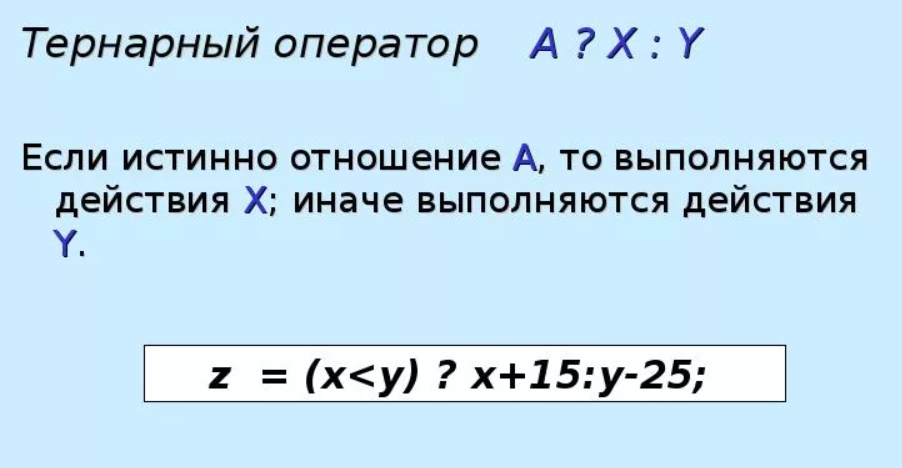
{

r = a + b;

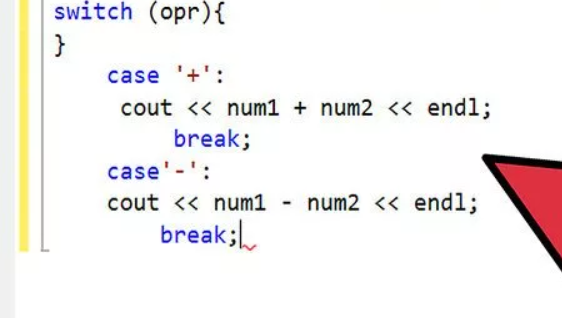
push\_back(pArr, size, r);

}

fСКАНФ В УСЛОВИИ ВЫПОЛНЯЕТСЯ ТАК КАК БУДТО ОН В ТЕЛЕ ЦИКЛА



switch(x):



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Строки

#include<string>

ВМЕСТО FILE \* stream МОЖНО ИСПОЛЬЗОВАТЬ stdin/stdout И КОНКРЕТНЫЙ БУФЕР char \*str

## Инициализация строк:

1.

char str1[] = "first line";

const char\* str = "first line";

2. ВАЖНО при начале работы с неизвестной строкой - сделать нулевую строку:

два варианта:

1Если вы хотите, чтобы line была пустой строкой, вы можете инициализировать ее как const char line[] = ""; . - массив будет const - неудобно

2 char\* line = new char[1];

line[0] = '\0'; //Пустая строка ⇒ массив можно редактировать

### cin/cout

0. cin >> int x; // cin читает первое слово ДО первого пробела

cout << x+1 << ‘\n’;

cin - класс

cin.get()

cin.ignore()

Возвращаемое значение оператора >> можно использовать в условных выражения для проверки состояния потока. Если операция ввода прошла успешно, то объект потока будет приведен к типу true. В противном случае, объект потока будет приведен к типу false. ⇒ while (cin >> noskipws >> s)

### Посимвольно (getchar/fputc):

1.char s - хранит 1 символ.

//ввод и вывод одного символа:

//char s;

//s = getchar();

//putchar(s); // (после путчар он не заканчивает строчку) ВАЖНО очищать входной буфер, чтобы другие входы не портились

gets и puts (удалены начиная с С23 - используются fgets fputs)

объявление gets:

char\* gets(char \*buf);

1. функция читает stdin строку до “/0” и сохраняет её в буфере по адресу \*buff

(\*buff адресс первый буквы строки, сохраненной в буфере, которую мы ввели)

1. функция возвращает указатель на место первого символа строки buff в буфере.

fgetc: Читает один символ из файла, возвращает его значение в формате int или EOF в случае ошибки или конца файла.

Объявление: int fgetc(FILE\* stream);

char\* str = "Hello";

char firstChar = fgetc(str);

fputc: Записывает один символ в файл, возвращает его значение в формате int или EOF в случае ошибки.

int fputc(int character, FILE\* stream);

### Читают до \n (fgets(), getline(str/cin))

Вы можете использовать функцию fgets() для чтения введенной пользователем строки с клавиатуры, используя stdin в качестве аргумента stream.

char \*fgets(char \*str, int n, FILE \*stream);

char \*fgets(адрес буфера, кол-во символов буфера, stdin)

fputs(s, stdout); - для вывода в консоль

FILE\* fp;

fputs(s, fp); - для вывода в файл.

//важно что для вывода 1 символа через fputs нужно преобразовать символ в строку.

- - - - - - - - - - - - - – - - -

getline: // читает до \n

string str;

getline(cin, str);

std::cout << "Вы ввели: " << str << std::endl;

Пример использования:

//На каждой итерации цикла getline будет читать новую строку из файла

ifstream dataFile("D:\\Visual studio\\projects\\CKb LETI\\1 zadanie\\data\\data.txt");

string line;

while ( getline(dataFile, line)) {

istringstream ss(line);

string key, name, type;

ss >> key >> name >> type;

cin.get() - ввод одного символа

cout << "Enter a character: ";

ch = cin.get();

cout << "You entered: " << ch << endl;

Если входной поток пуст, то метод возвращает значение -1. Это позволяет определить конец файла.

### printf / scanf

3. printf / scanf // НЕ принимаются в новых С++ стандартах из-за небезопасности

syntax:

printf(“[флаг][ширина][точность][модификатор][тип]”, переменные);

**printf("%d\t%4.2lf\t%6.1f\n", i, d, f);**

**“12 23.00 0123.3\n”**

scanf(“[\*][ширина][модификатор][тип]”, переменные);

**scanf(" %\*d) %f %50s", &f, str);**

**“ 10) 1.32 яблоня\n”.**

Чтобы число прочиталось, но не присвоилось никому, нужно %\*d вместо %d

#### версии printf, scanf:

scanf\_s() - тоже самое, но можно в конце добавить ещё один аргумент(необязательно), обозначающий макс длину строки для чтения. scanf\_s("%99s", str, 100);

sscanf() - не из stdin, а из готовой строки или массива.S

char str[] = "42";

int a;

sscanf(str, "%d", &a);

sprintf()- запишет строку не в stdout, а в готовую строку или массив.

int sprintf(char \*str, const char \*string,...);

If successful, it returns the total number of characters written excluding null-character appended in the string.

In case of failure a negative number is returned .

#include <iostream>

snprintf() - ещё более безопасный метод, тоже самое что и sprintf(), но позволяет контролировать количество символов в буфере.

string MyString = “words”;

char buffer[100];

int snprintf(buffer, sizeof(buffer), "Моя строка: %s\n", myString.c\_str());

Возвращает:

возвращает количество символов, которое было бы записано в буфер, если бы буфер был достаточно большим.

Важно: Возвращаемое значение не включает завершающий нулевой символ (\0), который добавляется в конец строки C-стиля.

### istringstream

isstringstream - это класс, который при создании объекта класса принимает строку и имеет разбивать строку на слова, разделенные пробелами.

#include <sstream>

сигнатура: std::istringstream(const std::string& str = "");

объявление: isstring obj(input)

аргументы:

str (необязательный): Строка, которую нужно преобразовать в поток ввода. По умолчанию используется пустая строка.

\*если не передавать никакой аргумент в функцию, то по условию она содержит пустую строку.

пример:

int main() {

std::string input = "123 456.789";

std::istringstream iss(input); // Создаем istringstream из строки

int num1, num2;

double num3;

iss >> num1 >> num2 >> num3; // Извлекаем данные из потока

std::cout << "num1: " << num1 << std::endl;

std::cout << "num2: " << num2 << std::endl;

std::cout << "num3: " << num3 << std::endl;

### перевод в C строку

метод c\_str() в классе string переводит строку в стиль С.

string str= “string”;

string c\_str = str.c\_str();

## Инфо

В C-стилевых строках, конец строки обозначается нулевым символом, также известным как символ завершения строки или нулевым терминатором. В ASCII-кодировке этот символ имеет числовое значение 0 (ноль).

В C, строки представляются в виде массива символов типа `char`, где последний символ массива является нулевым символом. Например, строка "Hello" будет представлена как массив `{'H', 'e', 'l', 'l', 'o', '\0'}`.

* \_ - - - - - – - - - - - - - - - - - - - – - - – - - - - - - - - – – - - - -

### снятие/добавление const

const\_cast<char\*>(string) - это оператор const\_cast, который используется для удаления или добавления квалификатора const для указателя или ссылки на объект.

В вашем коде const char\* string объявлен как указатель на константный (const) символ. Однако, для использования функции strcpy\_s, которая требует неконстантного (non-const) указателя, необходимо удалить квалификатор const временно.

## Очистка stdin/stdout буфера

2.ОЧЕНЬ ВАЖНО СЛЕДИТЬ ЗА ОЧИСТКОЙ БУФЕРА, когда работаешь со строчками

Очистка выходного буфера: fflush(FILE) / fflush(stdout)

Очистка Входного буфера:

void clear\_input\_buffer()

{

int c;

while ((c = getchar()) != '\n' && c != EOF) {}

} // имеет проблемы если stdin уже пустой

Версия с проверкой на пустоту буфера (не работает пока)

void clear\_input\_buffer() {

int c;

if (feof(stdin)) { // используем feof(stdin) для проверки остатков буфера

printf("В stdin нет остаточных данных\n");

} else {

printf("В stdin есть остаточные данные\n");

}

while ((c = getchar()) != '\n' && c != EOF) {}

}

## Чтение пробела cin’ом

* - - - - - - - - - - - - -- - - - - - - - - - - - -- - - - - - - - - - - - -- - - - - - - - - - - - -

Оператор чтения из потока ввода (>>), обычно, игнорирует пробельные символы (это поведение можно изменить с помощью std::noskipws)

Значит, что noskipws заставляет cin читать ‘\n’ и ‘ ‘ и ‘\0’

char ch;

cout << "Enter a string with spaces: ";

cin >> std::noskipws >> ch;

cout << "You entered: " << ch << endl;

В этом примере мы используем манипулятор std::noskipws перед оператором >>, чтобы отключить пропуск символов пробела.

* - - - - - - - - - - - - -- - - - - - - - - - - - -- - - - - - - - - - - - -- - - - - - - - - - - - -

## EXCEL Импорт/экспорт

создать файл в стиле C или с помощью ofstream, но с расширением “.csv”

по умолчанию с помощью ofstream file; file << “aaa” << “b” << …… ; заполняется только первый столбец

для работы со столбцами в стиле С++:

[https://www.sourcecodester.com/cc/6088/c-read-and-write-excel-files.html](https://www.youtube.com/redirect?event=comments&redir_token=QUFFLUhqbFFmVFZjRklfLUJyeHVXamZDTnlGMFpoUTZsQXxBQ3Jtc0ttSm9aV1pqcmtoeUlqaXlwTHA4dExzN1JoOThLeTJJV0FfQWpNVXk5U3RVUHBHaG9LMmpZYjl0UnRiOHZVSjkzMm9ERWx6Q3JzaERSUHQzY3VkSjAtajlHXzVDRVN2QzFkbG9yZWlselltdDV3Tm5HOA&q=https%3A%2F%2Fwww.sourcecodester.com%2Fcc%2F6088%2Fc-read-and-write-excel-files.html&stzid=Ugxhq1wkt_feC7ii-dV4AaABAg.8_ZN0Zu-bpQ8_hZOwP4I3w)<http://www.cplusplus.com/forum/beginner/56302/>

## 

## Преобразование числа в строку

char buffer[16];

int i =1;

sprintf(buffer, "%d", i);

char buffer[16];

double ii =2;

sprintf(buffer, "%d", ii);

# String

strcpy() - не используется  
errno\_t strcpy\_s(char\* dest, rsize\_t destSize, const char\* src);

char dest[20];

const char\* src = "Привет, мир!";

errno\_t result = strcpy\_s(dest, sizeof(dest), src);

strlengh(сhar \*ptr) - работает только с массивом char

string является классом

можно узнать количество символов: string a;

cout >> a.size() ;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Указатели

int a=5;

int \*px = &a; (& взятие адреса) // px = 0xF3910

тип указателя нужен только такой же как и переменная на которую он указывает

указатель на массив:

char c[10];

char\* pc = c;

char\* s и char \*s - одно и тоже

Когда вы создаете указатель на массив, вы не используете оператор &, потому что имя массива в языке C уже является указателем на первый элемент массива.

Таким образом, если у вас есть массив arr:

c

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

то имя arr в языке C является указателем на первый элемент массива. Таким образом, arr и &arr[0] являются эквивалентными выражениями и оба они возвращают адрес первого элемента массива.

## сброс указателя

удаление данных в массиве: delete [] arr;

ptr = NULL или ptr =0;

## Арифметика указателей

#include <iostream>

int main() {

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

int length = sizeof(array) / sizeof(int);

int\* ptr = array;

for (int i = 0; i < length; i++) {

std::cout << \*ptr << " ";

ptr++;

}

return 0;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Работа с файлами

## инфо

TXT нУжно переводить в ANSII

1. открытие файла fopen —> fopen\_s

FILE\* ptr ; создание указателя на адрес функции для открытия конкретного файла

// ptr = fopen(“путь файла”, “режим”); → устарело

int err = fopen\_s( &ptr, "путь", "режим");

функция fopen\_s() возвращает ненулевое значение, если произошла ошибка при открытии файла. Если открытие файла прошло успешно, функция возвращает 0.

\*ПУТЬ УКАЗЫВАЕТСЯ ЧЕРЕЗ ДВОЙНЫЕ СЛЕШИ

путь:

"D:\\Visual studio\\projects\\task a+b string\\FILE ptr\\test text.txt"

или

“new\_txt.txt” - создание файла в папке проекта (чтение файла из папки проекта тоже возможно таким способом)

* можно записать полный путь в переменную string

string path = “\*\*\*”;

режимы:

**r** Чтение.  
 **w** Запись. Если файл существует,  
 он будет перезаписан  
 **a** Добавление. Если файл не   
 существует, он будет создан  
 **r+** Открытие файла для  
 чтения и записи  
 **w+** Создание нового файла  
 для чтения и записи  
 **a+** Добавление с  
 возможностью чтения  
 **b** Двоичный файл  
 **t** Текстовый файл

1. работа с указателем. ОЧЕНЬ ВАЖНО СЛЕДИТЬ ЗА УКАЗАТЕЛЕМ.

// Перемещает указатель потока

fseek(FILE\* *файл*, *смещение*, int *привязка*);

привязки: **SEEK\_SET (0) - от начала файла**

**SEEK\_CUR (1) - от текущей позиции**

**SEEK\_END (2) - от конца**

// Возвращает текущую позицию указателя потока от начала файла  
 int offset = ftell(FILE\* *файл*);

// Перемещает указатель потока на начало файла

void rewind(FILE\* *файл*);

1. чтение и запись файла

fgetc(FILE\* ptr); посимвольное чтение // возвращает прочитанный символ

fputc(int c, FILE\* ptr); посимвольная запись

символы в char пишутся в ‘s’, а не в “s”.

char\* ptr2 = char a[100];

возвращает адрес строки в буфере

char\* fgets(адрес массива, кол-во символов чтения, адрес файла);

**char\* fgets(char \*buf, int size, FILE\* *файл*);** построчное чтение

**char arr[20];**

**fgets(arr, sizeof(arr), ptr2);**

если не знаешь сколько символов читать то пиши sizeof(массив).

- - - - - - - - - - - -

возвращает неотрицательное значение в случае ошибки

int fputs(массив строки, адрес файла);

**int fputs(char \*buf, FILE\* *файл*);** построчный ввод

fputs(arr, ptr);

**int fprintf(out, "Дата: %02d.%02d.%d", day, month, year);**

**int fscanf\_s(inp, "Дата: %d.%d.%d", &day, &month, &year);**

\*fscanf() устарел

* Каждая из этих функций возвращает количество успешно преобразованных и назначенных полей; Возвращаемое значение не включает поля, которые были прочитаны, но не назначены. Возвращаемое значение 0 указывает, что поля не были назначены. Если до первого преобразования возникает ошибка или достигается конец потока файла, возвращается значение EOF для fscanf и fwscanf.
* printf() и fprintf() возвращают количество символов, успешно записанных в выходной поток. Если возвращенное значение меньше, чем количество запрошенных символов, то произошла ошибка записи.

FSCANF() ПОСЛЕ ЧТЕНИЯ СТРОКИ НЕ ВИДИТ \n, и \n нужно читать fgetc()

1. закрытие файла

fclose(ptr);

конец строки - \n

конец файла - EOF/ feof

строки в файле выглядят как:

первая строка\nвторая строка\n

* - - - - - - - - - - - - - - - - – – - - - - - - - - - - - - - – - - - - - - - - - - - - - - - – - - - - - - - - -

## Функции для файлов

int feof(FILE \*stream);

Аргумент stream является указателем на файловый поток, для которого нужно проверить наличие конца файла.

Функция возвращает целое значение, которое является ненулевым, если достигнут конец файла, и нулевым в противном случае.

Функция feof() не читает данные из потока, а только проверяет, достигнут ли конец файла.

## Процедуры в чтении файлов

if (feof(stdin)) { } //не работает / while(feof(FILE\*ptr)) { }

if (feof(stdin)) {

printf("В stdin нет остаточных данных\n");

}

else {

printf("В stdin есть остаточные данные\n");

}

## Ofstream библиотека

#include<fstream>

3 класса:

fstream

ifstream ( поток входа - in stream)

ofstream ( поток вывода - out stream)

ofstream:

ofstream file;

file.open(“myfile.txt”); // файл в папке с проектом

file.is\_open() // проверяет открыт ли файл

поток вывода:

file << “data” << endl;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Технология

прочитать строки из файла, сохранить в памяти + что то потом сделать.

разобраться (массив указателей на строки)

<https://poe.com/s/AG2ZLMcu4ibAIeVfQh18>

# Динамическое выделение памяти

динамическое выделение памяти (после компиляции)

1. создание указателя на область дополнительной памяти (выделение памяти под нашу сущность)

int \*ptr = new int; // выделяет столько памяти сколько требует тип int + указатель на

// это место памяти

динамическая память работает именно через указатель на доп. место памяти в heap’e

2. Удаление выделенной памяти delete

3. Очистка адреса указателя ptr = NULL / ptr = nullptr для предотвращения ошибок обращения по указателю

* запись NULL это макросс #define NULL 0 ; - одно и тоже что и 0.
* nullptr - отдельный тип данных, сделанный специально, чтобы не путать с int 0.
* если int a;

int \*ptr = &a;

\*ptr = 10;

ptr = nullptr; - то данные по адресу не исчезают и занимают бесполезное место

// ptr имеет адрес: 00000000, а \*ptr = ??? - связь со значением потеряна и эта операция невыполнима

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## New Delete

1. const char\* string = new char[100]; //

string = "LaSteriod Daddy Hawk the IV"; // Ошибка на этапе delete

printf("%s", string); //

delete[] string; //

Ваша проблема заключается в том, что после выделения памяти с помощью new[] для массива символов, вы изменяете указатель string, присваивая ему новое значение, которое является строковым литералом.

string = "LaSteriod Daddy Hawk the IV";

Это приводит к потере указателя на выделенную память и приводит к нарушению памяти (memory leak), так как вы больше не имеете доступа к выделенной памяти для освобождения ее путем оператора delete[].

Чтобы исправить проблему, вам нужно скопировать содержимое строки "LaSteriod Daddy Hawk the IV" в выделенную память, вместо того чтобы изменять указатель на строку:

const char\* string = new char[100];

strcpy\_s(const\_cast<char\*>(string), 100, "LaSteriod Daddy Hawk the IV");

3. Также проблема возникает из за смешивания C операторов realloc и подобных и C++ операторов new[ ] delete[ ].

Если создать доп. память с помощью C операторов, то удалять тоже нужно с помощью С операторов ( free(void\*))

## realloc

1. существующий массив - int m[ ]

изменяем размер выделенной памяти - m = (int \*)realloc(m, 2000\* sizeof(int))

// массив становится на 2000 элементов

2. у функций типа void\* вроде malloc или realloc перед использованием нужно писать:

line = (char\*)realloc(line, 2\* sizeof(char));

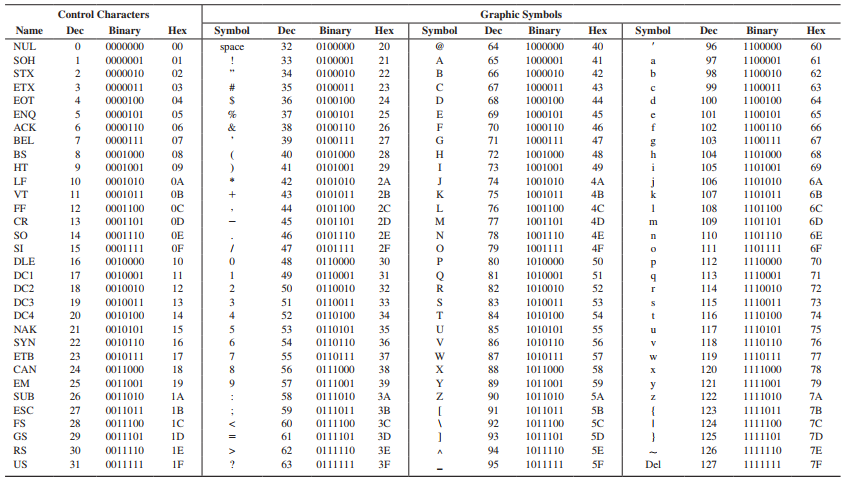
# Массивы

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

## info

В Дебаге char массив (в атмел студио) отображается в ascii кодировке

[ASCII Code – VLSIFacts](https://www.vlsifacts.com/ascii-code/)



int arr[10];

arr - указатель на адрес массива (указатель на нулевой элемент массива)

arr[0] - нулевой элемент массива

\*arr - также нулевой элемент массива

arr[i] - значение элемента массива

arr - указатель ⇒ arr[i] - значение

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

int a;

int \*ptr = &a;

cout << ptr << ptr +1 // + 4 это плюс четыре единицы занимаемой памяти типа

данных

То есть: int = 4 byte ⇒ ptr = 0xf131rf1q00 ; ptr +1 = 0xf131rf1q04

* - - - - - - - - - - - - - - - - - -- - - - - - - - - - - - - - - - - -- - - - - - - - - - - - - - - - - -- - -

arr[i]= 1 // эквивалентно \*(arr+i) =1

int arr[2], arr2[2];

arr[0] = 0;

arr[1] = 1;

\*arr2 = 2;

\*(arr2 + 1) = 3;

cout << arr << '\n' << \*arr << arr[1] << '\n' << \*arr2 << \*(arr2 + 1) << '\n';

## Вывод значений и адресов массивов

const char \* str;

cout << "Вывод строки: " << str << '\n';   
cout << "Адрес строки (первого элемента) :" << static\_cast<const void\*>(str) << '\n';  
cout <<"Адрес указателя на строку:" << &str1 << "\n";

## Передача массива в функцию

[Шаг 8 – Указатели и массивы – Stepik](https://stepik.org/lesson/539/step/8?unit=862)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Функции

void foo()

1. a) Передача указателя на указатель int \*\*arr может использоваться, когда мы хотим изменить сам указатель на массив в функции. Например, если мы хотим выделить память под массив внутри функции и вернуть его как выходной параметр. При этом нам нужно передать в функцию указатель на указатель на массив, чтобы мы могли изменить значение указателя на массив в самой функции.

б)В случае передачи ссылки на указатель int \*&arr мы передаем указатель на массив по ссылке, что позволяет изменять его значение внутри функции, сохраняя тем самым изменения за пределами функции. Это может быть полезно, если мы хотим изменить значение указателя на массив в функции, а не сам массив.

1. Передача значения по ссылке int &a используется для того, чтобы мы могли изменить значение переменной a внутри функции и сохранить изменения за пределами функции. Это может быть полезно, если мы хотим избежать копирования больших объектов при передаче их в функцию.

Передача значения по значению int a создает копию значения переменной a, которая передается в функцию. Любые изменения, сделанные внутри функции, не будут отражены в исходной переменной.

Обычно передача по значению используется для передачи простых типов данных, таких как int, float, bool, а передача по ссылке - для передачи сложных объектов, таких как массивы или пользовательские структуры.

## Изменение глобальных переменных в функции

В вашем примере переменная time останется неизменной, потому что вы передаете в функцию foo значение time, а не его адрес. Это значит, что функция foo будет работать с копией значения time, и изменения переменной time внутри функции не будут отражаться на переменной time в main.

Чтобы функция могла изменить значение переменной time, вам нужно передать в функцию адрес переменной time (указатель) вместо значения. Вот пример:

*1*#include <iostream>

*2*

*3*void foo(int\* time) {

*4* (\*time)++;

*5*}

*6*

*7*int main() {

*8* int time = 0;

*9* foo(&time);

*10*

*11* std::cout << "New time value: " << time << std::endl;

*12* return 0;

*13*}

*14*

В данном коде мы передаем адрес переменной time в функцию foo, и функция изменяет значение переменной time по этому адресу. Теперь значение time будет увеличено на 1 и будет отражено при выводе в main.

### 

### Операции с \*arg

Если передавать какой то объект в аргумент функции по ссылке

например словарь: int func(unordered\_map<string, int> obj),

то при выполнении операций с ним нужно \*obj

например операция: (\*obj).[“apple”] = 24

0. ИНКРЕМЕНТЫ В ФУНКЦИЯХ ++i : иногда инкременты не работают сразу в функциях, но всегда работает ( i + 1) // немного фейк

# Шаблонные функции

исходная функция:

void USARTTransmitChar(char c) {

while(!( UCSRA & (1<<UDRE)));

UDR = c;

}

шаблонная версия этой функции:

template <typename T>; // шаблон любого типа данных (typename

// можно заменить на class, чтобы

// собственные типы данных использовать)

T USARTTransmitChar(T c) {

while(!( UCSRA & (1<<UDRE)));

UDR = c;

}

* Если в функции нужны несколько разных типов данных, то нужно создать несколько темплейтов: template <typename T1, typename T2>;

# Мои функции

добавление доп. ячеек памяти к массиву через new[]:

функция для temp указателей под realloc НУЖНО СДЕЛАТЬ ШАБЛОННОЙ

char\* TEMPptr(char\* str, char\* tempstr)

{

tempstr = str;

return tempstr;

}

Алгоритм работы со С-style строками при realloc:

const char\* original = "a";

char\* line = (char\*)malloc((strlen(original) + 1) \* sizeof(char));

strcpy\_s(line, sizeof(line), original);

line = (char\*)realloc(line, 2 \* sizeof(char));

printf("строка: %s\nадрес: %p", line, (void\*)line);

free(line);

return 0;

Алгоритм работы с нулевыми строками при realloc ВАЖНО СЛЕДИТЬ ЗА TEMPptr

после realloc

setlocale(LC\_ALL, "ru");

char\* line = new char[0];

printf("адрес: %p\n\n");

line = (char\*)realloc(line, 2 \* sizeof(char));

char\* TEMPline = line;

\*TEMPline++ = 'a';

\*TEMPline = '\0';

printf("строка: %s\nадрес: %p", line, line);

delete [] line;

return 0;

Алгоритм выделения памяти для строки

char\* line = new char[1];

\*line = '\0';

int i = 1;

char s = 'q';

line = (char\*)realloc(line, (i+1) \* sizeof(char));

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

char\* TEMPline = line + i-1;

\*TEMPline++ = s;

\*TEMPline = '\0';

printf("%s", line);

TEMPline = nullptr;

free(line);

line = nullptr;

## Управление консолью

очистить консоль:

#include "windows.h"

system(“cls”);

очистить конкретную строку:

#include < windows.h >

HANDLE hd = GetStdHandle(STD\_OUTPUT\_HANDLE);

COORD cd;

....

cd.X=0;

cd.Y=15; //номер строки

SetConsoleCursorPosition(hd,cd); //установили курсор в начало строки

cout < < " "; //затёрлиБили в цикле вывели нужное количество пробелов

SetConsoleCursorPosition(hd,d); //опять установили куда нужно

# Библиотеки

## задержка

\_\_asm("nop"); - задержка на 1 машинный цикл

#include "windows.h"

//..

Sleep(5000); // ждать 5 секунд, время эта функция считает в тысячных долях секунды

# Многопоточное программирование

#include<iostream>

#include<thread>

#include<chrono>

using namespace std;

void func()

{

for(int i=1; i<10;i++)

{

printf("%d\tSIDEthread\t%d\n", this\_thread::get\_id(), i);

this\_thread::sleep\_for(chrono::milliseconds(4000));

}

}

void main()

{

thread th1(func); // после инициализации треда функция, адрес которой передали в аргументе начнет сразу же выполняться. НУЖНО ПЕРЕДАВАТЬ численны е АРГУМЕНТЫ ЧЕРЕЗ ЗАПЯТУЮ если они есть в функции: thread th1(func, 1, 1 ,1)

//th1.detach(); - функция делающая так, чтобы если тред будет паралельно выполняться и закончит свой цикл, то ВСЯ программа завершится

for (int i = 0; i < 10; i++)

{

printf("%d\tMAINthread\t%d\n", this\_thread::get\_id(), i);

this\_thread::sleep\_for(chrono::milliseconds(1000));

}

th1.join(); // заставляет тред после выполнения ждать выполнения основного треда и только по окончанию цикла основного треда программа завершается

}

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Структуры данных С++

## Словарь

Ассоциированный массив: unordered\_map - класс

создание объекта класса:

#include <unordered\_map>

unordered\_map<string, UGOData> data

аргументы:

string - тип данных для ключа словаря

UGOData - пользовательский тип данных для значений для ключа.

### Print

Вывод всех ключей:

unordered\_map<string, int> myMap = {

{"apple", 1},

{"banana", 2},

{"cherry", 3}

};

// Цикл по всем элементам словаря

for (auto it = myMap.begin(); it != myMap.end(); ++it) {

cout << "Ключ: " << it->first << ", Значение: " << it->second << endl;

}

Вывод сложных структур данных на основе словаря:

#include <iostream>

#include <unordered\_map>

#include <vector>

using namespace std;

struct UGOData {

string imgPath; // Путь к изображению

string key;

vector<int> ugoValues;

vector<string> info;

};

void printUGODataMap(const unordered\_map<string, UGOData>& data) {

for (const auto& [key, value] : data) {

cout << "Ключ: " << key << endl;

cout << " imgPath: " << value.imgPath << endl;

cout << " key: " << value.key << endl;

cout << " ugoValues: ";

for (int val : value.ugoValues) {

cout << val << " ";

}

cout << endl;

cout << " info: ";

for (const string& str : value.info) {

cout << str << " ";

}

cout << endl;

}

}

int main() {

unordered\_map<string, UGOData> data;

// Заполняем данные в map

data["image1"] = {"images/image1.jpg", "image1", {1, 2, 3}, {"Описание 1", "Дополнительная информация"}};

data["image2"] = {"images/image2.png", "image2", {4, 5, 6}, {"Описание 2"}};

// Вызываем функцию для вывода данных

printUGODataMap(data);

return 0;

}

## Vector<>

#include <vector>

### Print

std::vector<int> numbers = {1, 2, 3, 4, 5};

// Вывод всех элементов

for (int i = 0; i < numbers.size(); ++i) {

std::cout << numbers[i] << " ";

}

ИЛИ

for (int value : it->second.ugoValues) {

cout << value << " ";

}

# Извлечение файлов С++

если данные имеют сложную структуру и хранятся не только в одной папке и файле, то строит создать структуру данных, позволяющую сделать удобную навигацию для работы с ними

## Структуры

Пример структуры, которая может хранить данные из файлов:

#include<vector>

#include<iostream>

#include<string>

#include<unordored\_map>

struct UGOData {

string imgPath;

string key;

vector<int> ugoValues;

vector<string> info;

};

unordered\_map<string, UGOData> data

## Взаимодействие со struct

операции в цикле

for (const auto& pair : data) {

const string& key = pair.first;

const UGOData& value = pair.second;

…

}

const auto& pair: data здесь мы объявляем переменную pair типа const auto&.

обычно пишут const string& pair: data, но auto сам определяет тип.

const: означает, что значение pair не будет меняться в цикле.

auto: означает, что тип pair будет определен автоматически компилятором на основе типа элементов в data.

&: это ссылка, она позволяет pair ссылаться на элементы data без копирования их.

: data: это ключевое слово range-based for loop, оно означает, что цикл будет перебирать каждый элемент в data.

## Извлечение (чтение)

#include<iostream>

#include<unordered\_map>

#include <fstream>

#include <sstream>

#include <string>

#include <windows.h>

using namespace std;

int read\_data(unordered\_map<string, UGOData> \*data) {

// Путь к папке с изображениями

string imgDir = "D:\\Visual studio\\projects\\CKb LETI\\1 zadanie\\data\\Img\\";

// Чтение файла data.txt

ifstream dataFile("D:\\Visual studio\\projects\\CKb LETI\\1 zadanie\\data\\data.txt");

string line;

while ( getline(dataFile, line)) {

istringstream ss(line);

string key, inf;

ss >> key;

UGOData ugoData;

ugoData.key = key;

ugoData.imgPath = imgDir + key + ".jpg";

while (ss >> inf)

{

ugoData.info.push\_back(inf);

}

// Сохраняем в карту по ключу (например, "R")

(\*data)[key] = ugoData;

}

dataFile.close();

// Чтение файла ugo\_data.txt

ifstream ugoFile("D:\\Visual studio\\projects\\CKb LETI\\1 zadanie\\data\\ugo\_data.txt");

while ( getline(ugoFile, line)) {

istringstream ss(line);

string key;

int value;

ss >> key;

// Записываем значения УГО для соответствующего ключа

while (ss >> value) {

(\*data)[key].ugoValues.push\_back(value);

}

}

ugoFile.close();

return 0;

}

## Вывод структуры данных

void printUGODataMap(const unordered\_map<string, UGOData>& data) {

setlocale(LC\_ALL, "ru\_RU.UTF-8");

for (const auto& pair : data) {

const string& key = pair.first;

const UGOData& value = pair.second;

cout << "Ключ: " << key << endl;

cout << " imgPath: " << value.imgPath << endl;

cout << " key: " << value.key << endl;

cout << " ugoValues: ";

for (int val : value.ugoValues) {

cout << val << " ";

}

cout << endl;

cout << " info: ";

for (const string& str : value.info) {

cout << str << " ";

}

cout << endl;

}

setlocale(LC\_ALL, "ru");

}

ИЛИ

void print\_data(unordered\_map<string, UGOData> data)

{

for (auto it = data.begin(); it != data.end(); ++it) {

//wcout << L"Ключ: " << it->first << L", Значение: \n" << "\t";

for (string str : it->second.info) {

cout << str << " ";

}

cout << endl;

cout << it->second.imgPath << endl;

for (int value : it->second.ugoValues) {

cout << value << " ";

}

cout << "\n\n";

}

}