*Предисловие:*

*Это не столько подготовка к билетам, сколько «экспресс» курс по всему семестру и базе языка, по тем темам, которые написаны в билетах. Все что здесь написано – это реальная теория, которую я попытался преподнести через призму моего взгляда с моими комментариями, зачастую выделенными курсивом. Следовать/не следовать по этому руководству – всё решаете вы.*

*«Per aspera ad astra»*

**1) История языка Си.**

**Си** — компилируемый статически типизированный язык программирования общего назначения. Создан Деннисом Ритчи с помощью Брайана Кернигана и Кена Томпсона, как развитие языка Би для написания операционных систем UNIX и для компилирования кода на более старые языки. Согласно дизайну языка, его конструкции близко сопоставляются типичным машинным инструкциям, благодаря чему он нашёл применение в проектах, для которых был свойственен язык ассемблера, в том числе как в операционных системах, так и в различном прикладном программном обеспечении для множества устройств — от суперкомпьютеров до встраиваемых систем. Язык программирования Си оказал существенное влияние на развитие индустрии программного обеспечения, а его синтаксис стал основой для таких языков программирования, как C++, C#, Java и Objective-C.

**2) Компилируемые и интерпретируемые языки. Преимущества и недостатки.**

Компилируемым языком программирования называется язык, исходный код которого преобразуется компилятором в машинный код и записывается в файл с особым заголовком и/или расширением для последующей идентификации этого файла, как исполняемого операционной системой.

Интерпретируемым языком программирования называется язык, исходный код на котором выполняется методом интерпретации. Классифицируя языки программирования по способу исполнения, к группе интерпретируемых относят языки, в которых операторы программы друг за другом отдельно транслируются и сразу выполняются (интерпретируются) с помощью специальной программы-интерпретатора.

Преимущества компилируемых языков:

1) Производительность и скорость выполнения.  
2) Низкоуровневая гибкость при разработке: Больше возможностей у разработчика контролировать происходящее: Взаимодействие аппаратных частей, память и.т.д.  
3) Возможность запускать программу через скомпилированный исполняемый файл. Отсутствует необходимость в дополнительном ПО.

Недостатки:

1) Любые изменения кода требуют рекомпиляции для их внесения.  
2) Платформо-зависимость: Машинный код на разных системах зависит от машины на которой компилируется и исполняется программа.

Преимущества интерпретируемых языков:

1) Кроссплатформенность.  
2) Динамическая типизация.  
3) Меньшие затраты времени на разработку и отладку.  
  
Недостатки:

1) Серьезная разница в скорости выполнения с компилируемыми языками.  
2) Требовательность к наличию интерпретатора.

**3) Байт, бит, общее представление памяти.**

Бит – наименьшая единица измерения информации. Может принимать только два значения:   
1 или 0, True или False и.т.д.

Байт = 8 Бит.

Вся память представляется в виде регистров и адресов.

Нумерация адресов происходит в 16-ричной системе и кол-во адресов процессора зависит от архитектуры (x64 или x86).

*\*Подробнее в 8.2.*

**4) Типы данных, размер, максимальные и минимальные значения, знаковость.**

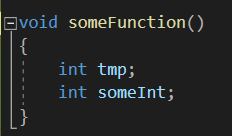
*Сомневаюсь, что есть способ лучше понять эту тему, чем просто +-выучить первую таблицу отсюда:*

[**Система типов Си — Википедия (wikipedia.org)**](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2_%D0%A1%D0%B8)

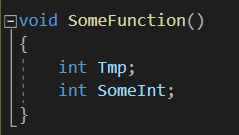
**5) Именование переменных, выравнивание в коде.**

*\*Возможно речь идет про номенклатуру и почерк программиста, хотя я не уверен.   
В любом случае знать не повредит.*

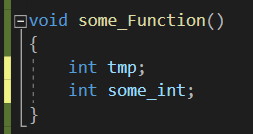
Хороший программист старается придерживаться обще-определённых стилей написания кода *(или, как минимум, старается не отходить от них)*. К этому относится номенклатура чего угодно, адекватные и понятные названия, и визуальный стиль кода.

Три самых популярных номенклатуры:

1) Начало с нижнего регистра, продолжение с верхнего:



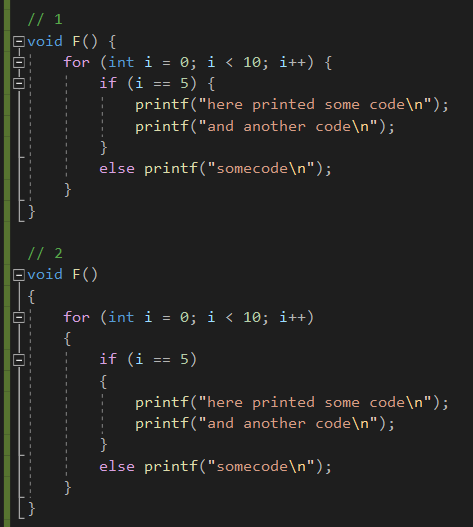
2) Любое слово с верхнего:



3) Использование подчеркиваний:

Визуальный стиль в основномпредставляет способ расставления фигурных скобок:

Выделяют два варианта:



Также можно добавить, что иногда кодстайл могут определять компании или же он может меняться от языка к языку.

*Цитата знающего*: «Например в Java принято называть классы с UpperCamelCase, функции и переменные с lowerCamelCase и статичные константы с UPPERCASE\_WITH\_UNDERSCORES»

*P.S: На самом деле можете писать как вам угодно, здесь нет железных правил. Это стилевые предпочтения, а не религиозные догмы. Но всё-таки придерживаться стандартов – это хороший тон.*

**6) Двоичное представление целых чисел (>0, <0). Таблица ASCII.**

Все целые числа в памяти представляются в виде дополнительного кода.

Дополнительный код для отрицательного числа можно получить инвертированием его двоичного модуля (все 0 превращаются в 1, и наоборот) и прибавлением к инверсии единицы.

Порядок операций взаимозаменяем:  
Инвертация числа + добавление единицы == Вычитание единицы + инвертация числа

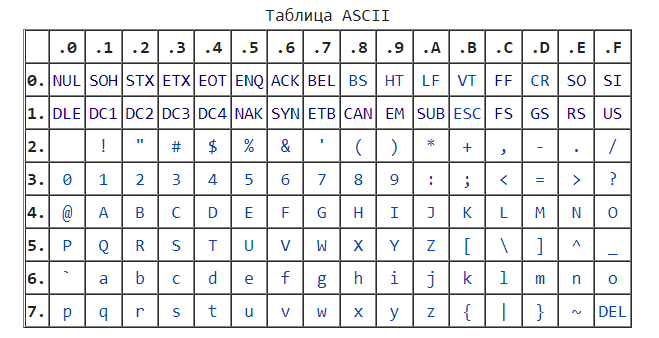
При записи числа в дополнительном коде старший разряд является знаковым. Если значение старшего разряда равно 0, то это значит, что в остальных разрядах записано положительное двоичное число, совпадающее с прямым кодом.

int = 4 байта = 32 бита = ±0(знаковый бит, 0 – положительное, 1 - отрицательное) 00… 00 –еще 31 разряд. Из того и следует граница значений: [-231;231-1].

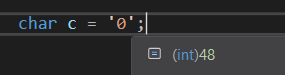
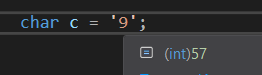
**ASCII** - (***A****merican****s****tandard****c****ode for****i****nformation****i****nterchange*) — название таблицы (кодировки, набора), в которой некоторым распространённым печатным и непечатным символам сопоставлены числовые коды. Таблица была разработана и стандартизирована в США, в 1963 году.

Таблица ASCII определяет коды для символов:

* десятичных цифр;
* латинского алфавита;
* национального алфавита;
* знаков препинания;
* управляющих символов.

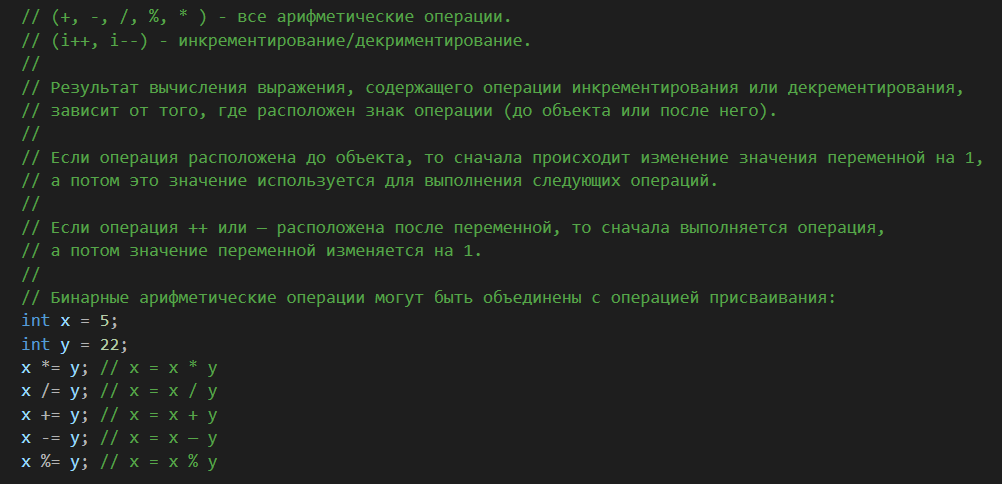


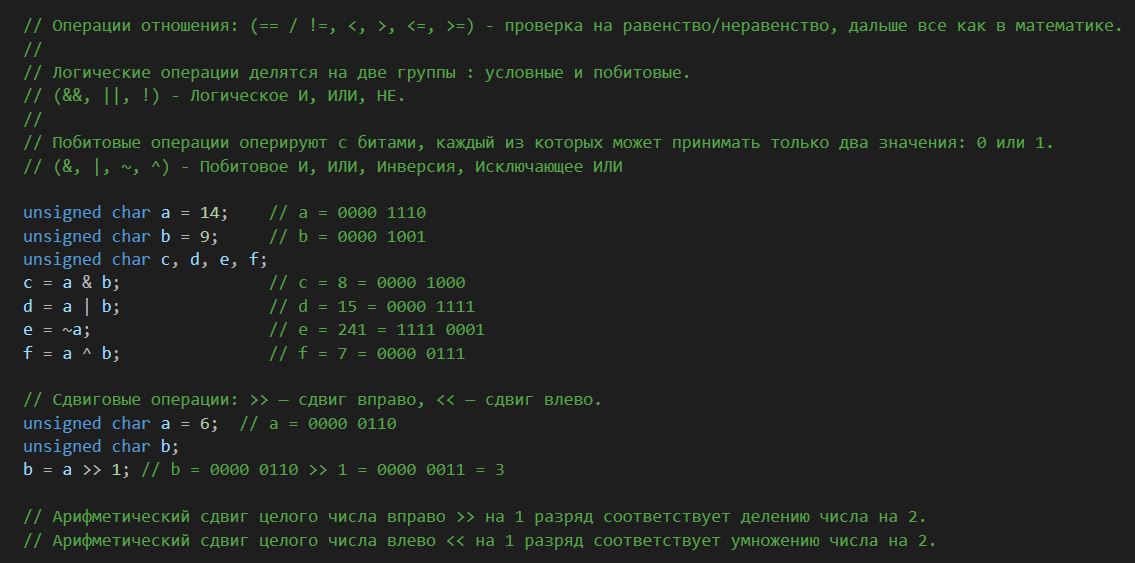
Например все символы цифр расположены по кодам от 48 до 57:

\*Unicode наследует таблицу ASCII поэтому первые 128 символов у них совпадают.

**7) Все арифметические, побитовые, логические операции.**





**8.1) Три вида циклов в Си.**

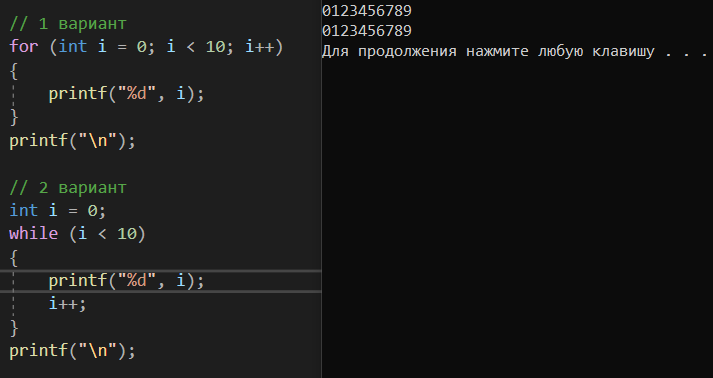
В Си присутствуют два основных ключевых слова для обозначения цикла: for и while.

Первый вид: for – безусловный цикл, ограничиваемый кол-вом иттераций.

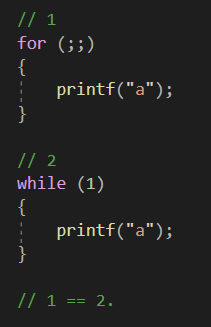
Второй вид: while – цикл с предусловием используется когда кол-во иттераций не известно, и в таком случае цикл ограничивается условием.

Третий вид: do+while – цикл с постусловием, его особенность от обычного while – такой цикл выполнится хотя-бы один раз, т.к проверка условия происходит после того как были выполенены все инструкции в теле цикла.

*На самам деле каждый из них можно реализовать через другой, и вся разница это логическое их использование для той или иной задачи:*



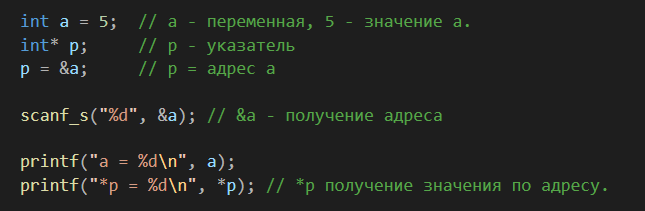
*Пример бесконечного цикла:*

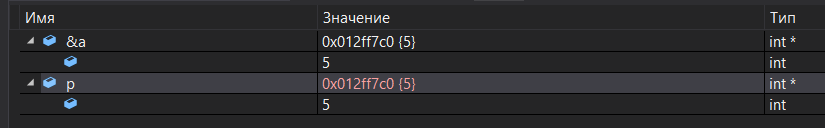


**8.2) Базовое понятие указателя в Си.**

Для любого типа данных будь то int, char или даже void существует тип «указатель на *\*вставьте тип\**». Указатели обозначаются звездочкой «\*» при объявляении переменной.

Указатель – переменная содержащая адрес ***объекта\* (см. конец страницы)***.





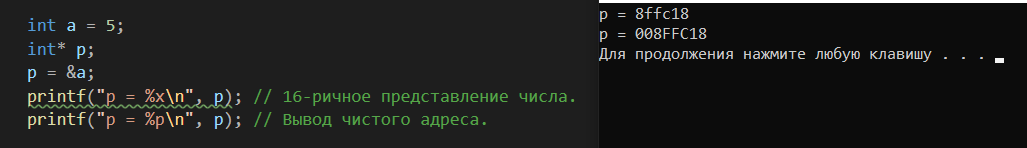
Действий с адресами всего 2: & и \*.

& - получает адрес переменной.

\* - получает значение по адресу.

Printf(“\*p = %d\n”, \*p); - в данной строке \*p означает разыменование указателя – получение значение хранящегося по этому адресу.

Scanf\_s(“%d”, &a); - Получение адреса переменной для записи в него.

Вывод адреса в консоль: 

*\*Под объектами в данном* *случае подразумеваются (пока-что) инты, символы, и.т.д. Хоть это и примитивные типы данных - в дальнейшем я буду всегда стараться использовать слово «объект» для обобщения типов, потому что это семантически верно.   
В языках высокого уровня (имеющих ООП и.т.д) объекты, экземпляры классов и другие понятия – это фундаментальная вещь. Сейчас особо не прошу вас вкладывать в это понятие что-то слишком сложное или абстрактное, но все-же я стараюсь рассказывать так, чтобы вы постепенно привык6али ко всей необъятной подноготной программировния. Если поймёте сейчас – дальше будет легче.*

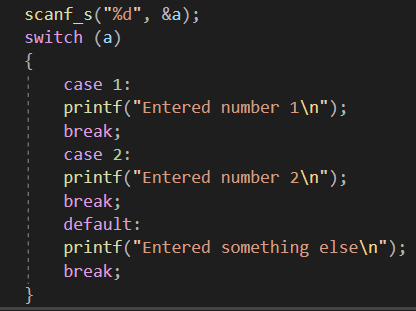
**9) Операторы break, continue, switch. Отладка в VS.**

Операторы break и continue используются в циклах: break для выхода из цикла, и для досрочного перехода иттерации.

Break при срабатывании моментально выходит из текущего уровня вложенности.

Continue при срабатывании моментально завершает текущую иттерацию и переходит к следующей.

Switch – оператор условного перехода, используемый для замены конструкций if, чтобы уменьшить кол-во кода.



Switch «ловит» значение переменной и обрабатывает его с помощью ключевых слов «case».

Все инструкции обрабатываются до ключевого слова break, срабатывание которого идентично поведению в циклах. Если break отсутствует то инструкции продолжат свое выполнение далее, переходя в следующий case.

Инструкции после default выполняются когда нет совпадений ни с каким из кейсов.

**Отладка в VS**

Отладка в Visual Studio работает через расставление точек останова и специальные действия:

Шаг с заходом (F11): Переходит к следующему шагу в коде, каким бы он ни был.

Шаг с обходом (F10): Переходит к следующему шагу в коде, пропуская подпрограммы в этих строчках и выполняя их.

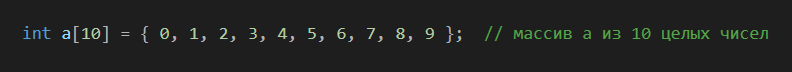
Шаг с выходом (Shift + F11): Выходит из текущей подпрограммы с ее выполнениием.

**10) Статические одномерные массивы. NULL-terminated строки в Си.**

Статические одномерныке массивы представляют собой непрерывный участок памяти строго определенного размера.

Объявление статических одномерных массивов выглядит так: **тип *название***[*размерность*]

Инициализация представляет собой набор начальных значений элементов массива, указанных в фигурных скобках, и разделенных запятыми.



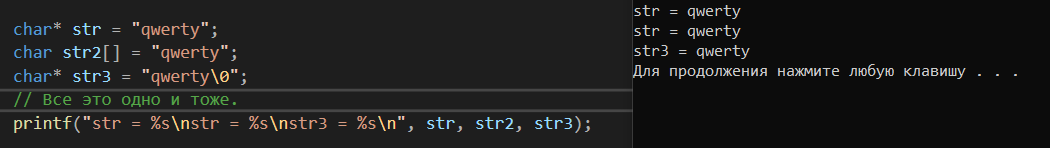
Если количество инициализирующих значений, указанных в фигурных скобках, меньше, чем количество элементов массива, указанное в квадратных скобках, то все оставшиеся элементы в массиве (для которых не хватило инициализирующих значений) будут равны нулю. Это свойство удобно использовать для задания нулевых значений всем элементам массива.



Обращение к элементам массива также указывается через квадратные скобки.

NULL-terminated строки, по другому Си-строки – основной способ представления строк в Си.

Строки в Си являют собой массив символов. Дабы знать когда эти массивы заканчиваются ввелось условие что все они будут заканчиватся символом '\0'.



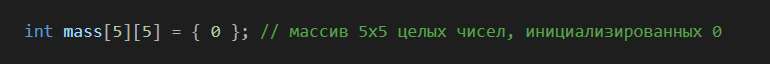
*P.S: Все три объекта являются строковыми литералами и указывают на статическую память. Подробнее в №19.*

**11) Типичные варианты использования массивов.**

*\*Уповаю на вашу фантазию :)*

**12.1) Статические многомерные массивы.**

Статические многомерные массивы являются массивом массивов. Основное отличие от одномерных в том что каждый элемент такого массива является указателем на другой массив. Основной синтаксис совпадает с примером выше, добавляются лишь скобки в зависимости от размерности.



**12.2) Файловый ввод/вывод.**

Работа с файлами в Си организована через потоки данных.

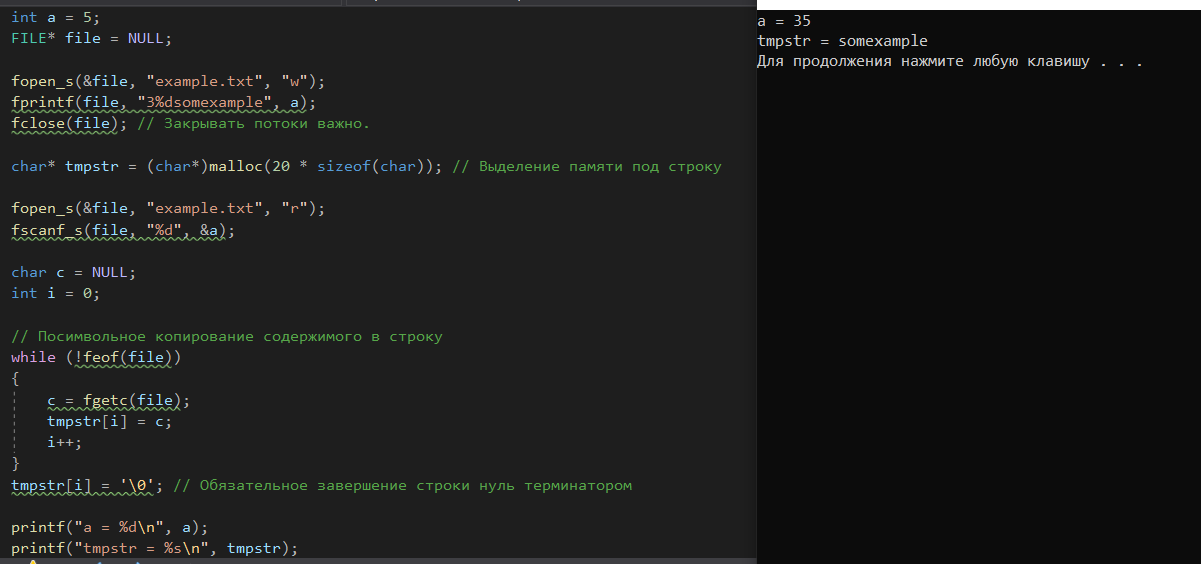
Для обработки файлового потока используются объекты-указатели на структуру FILE.

Сама структура FILE обращается к соответствующему файловому дескриптору.

Для открытия потока используются соответствующие функции fopen и fclose *(безопасные версии fopen\_s, fscanf\_s, fprintf\_s описаны Microsoft и являются модификациями исходных функций. Их существование за пределами Visual Studio не гарантированно\*)*

Работа с потоком (Чтение/Запись) происходит через определенные функции например как fgets, fgetc и др.

*Пример работы с файлами:*



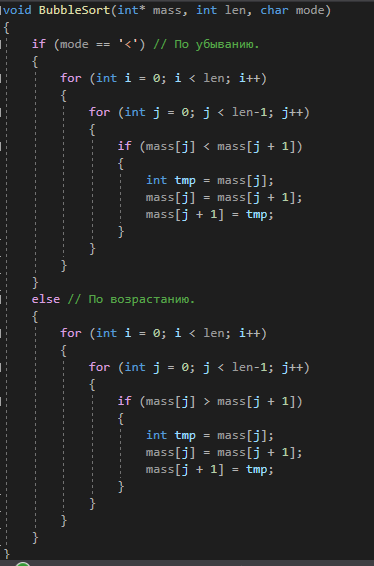
Подбронее о файловом вводе/выводе, параметрах, режимах открытия и.т.д:

[fopen\_s, \_wfopen\_s | Microsoft Docs](https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/reference/fopen-s-wfopen-s?view=msvc-170)

[Файловый ввод-вывод в языке Си — Википедия (wikipedia.org)](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB%D0%BE%D0%B2%D1%8B%D0%B9_%D0%B2%D0%B2%D0%BE%D0%B4-%D0%B2%D1%8B%D0%B2%D0%BE%D0%B4_%D0%B2_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B5_%D0%A1%D0%B8)

**13) Сортировка пузырьком.**

Смысл пузырька в том что за каждую иттерацию сортировка находит наибольший(наименьший) элемент и тот как бы «всплывает» как пузырек на воде, от того и название. Является самой медленной из всех возможных сортировок (сложность n2).



**14) Указатель. Разыменование указателя. Размер указателя.**

*\*Все в 8.2*

Размеры указателей определяются разрядностью системы. В 32-битной это 4 байта, в 64-битной это 8 байт.

**15)** **Операторы &, \*, вывод адреса в консоль. Отличия указателей на разные типы.**

*\*Все в 8.2*

Фактический тип данных для всех указателей, будь то целое число, float, character или other, является тем же самым, длинным шестнадцатеричным числом, которое представляет адрес памяти. Единственное различие между указателями разных типов данных - это тип данных переменной или константы, на которые указывает указатель.

**16.1)** **Динамические одномерные массивы. malloc.**

Динамические одномерные массивы – те же массивы, только их размеры зависят от выделенной для них памяти. Работа с ними построена на указателях.

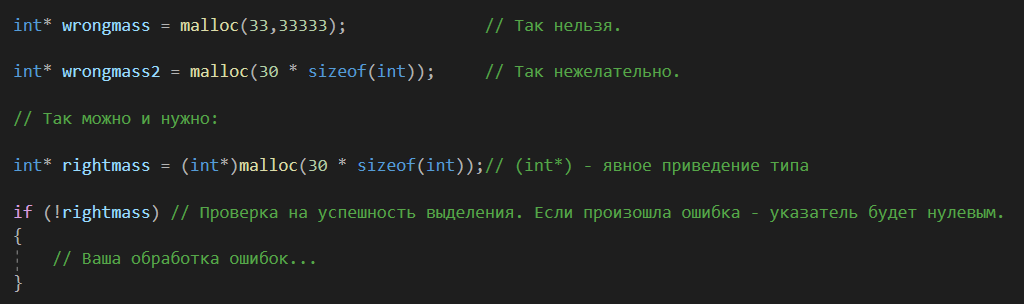
Для работы с динамическими массивами требуется определенный размер памяти, которая выделяется системой в куче(heap) – динамической памяти.

Эту память выделяет функция malloc (которая является более глубоким системным вызовом). Аргументом передаётся кол-во памяти в байтах. **Вместе с выделенной памятью malloc даёт гарантию того, что блок памяти по этому адресу не будет отдан другим приложениям.**

***Важное замечание:***

**1) Кол-во памяти в байтах всегда нужно выравнивать под тот тип данных, память для которого вы выделяете. Для этого и нужен оператор sizeof(тип данных).**

**2) malloc возвращает указатель void в случае успеха, или NULL, если работа функции завершилась с ошибкой – поэтому очень важно использовать явное приведение типов, и проверять успешность выделения памяти для дальнейшей работы с ней.**



**Для каждого вызова malloc после того как вы завершили использование памяти нужно вызывать free. Утечка памяти это плохо, и это может стать критической ошибкой. Хоть вся память которая выделена программе в любом случае очистится при завершении ёё работы, чистить память все равно важно!**

free получает в аргументах указатель на ранее выделенный блок памяти, далее чистит ровно столько, сколько было указано при вызове malloc, и **снимает гарантию** того что этот блок памяти не будет отдан другим приложениям.

Несмотря на то что free по завершении работы никак не трогает указатель на тот блок памяти, который он почистил, использовать память по этому указателю **нельзя.**

***Если речь идет про работу со строками, важно следить чтобы все символы которые вы расставляете всегда находились в пределах выделяемой для них памяти, иначе вызов free повредит кучу.*** ***Да и в принципе важно следить за тем чтобы все, чем вы хотите оперировать, находилось в границах выделяемой памяти.***

Подробнее:

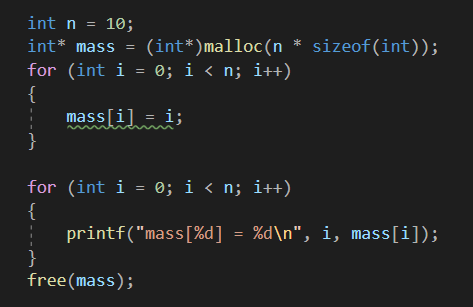
[malloc | Microsoft Docs](https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/reference/malloc?view=msvc-160) [free | Microsoft Docs](https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/reference/free?view=msvc-160)

**16.2) Арифметика указателей.**

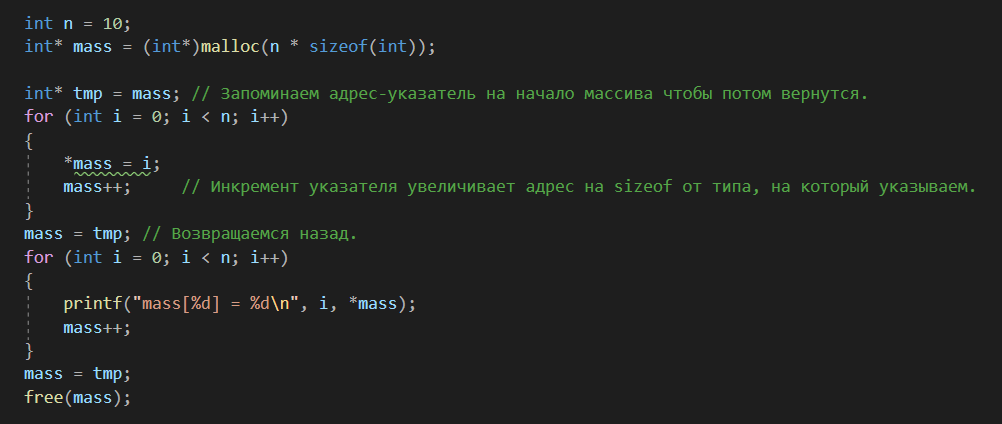
Работа с массивами предполагает использование элементов хранящихся в нем по их индексам.

Важно понимать что квадратные скобки к которым все привыкли – на самом деле не более чем синтаксический сахар для облегчения восприятия кода.

Т.е мы пишем:



Что происходит на самом деле (программа это делает за вас, и это довольно грубый пример):



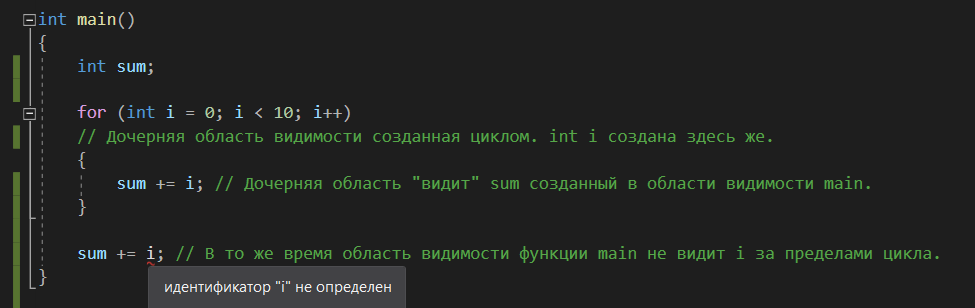
**17.1)** **Область видимости переменных, примеры.**

Областью видимости называется участок кода в котором тот или иной объект или функциональная часть могут быть доступны для любых других объектов или частей.

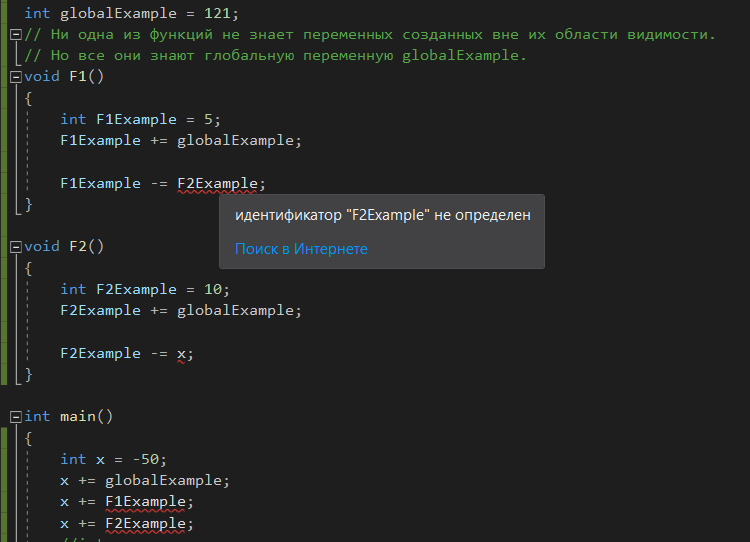
Основной способ выделения областей видимости – фигурные скобки.

**Созданные в своей области видимости объекты не могут быть доступны другим объектам за ее пределами.**

**Но в то же время дочерние области видимости могут использовать объекты родительской области:**



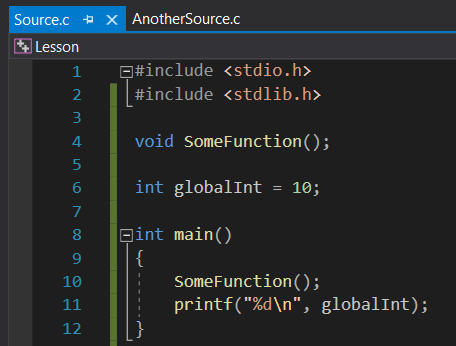
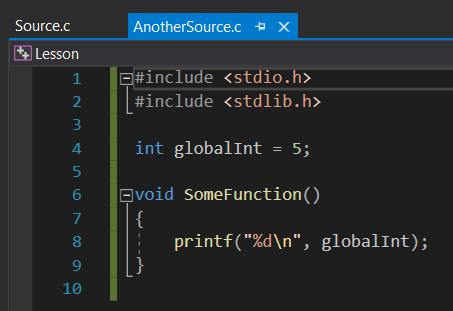
Еще пример:

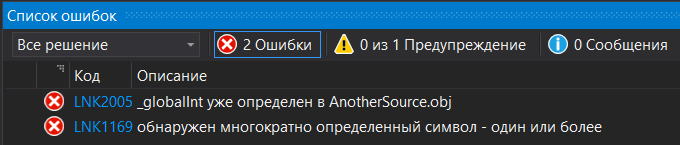


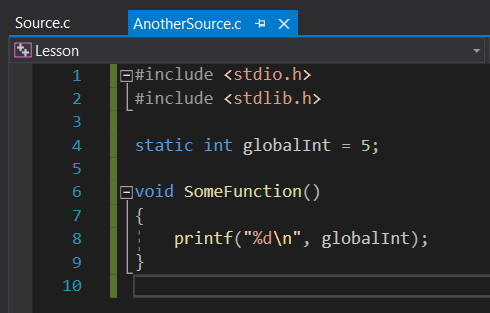
Одно из применений ключевого слова **static** связано с областью видимости:

Глобальные переменные и функции имеющие модификатор **static** могут быть видны **только в пределах данного файла исходного кода*.***

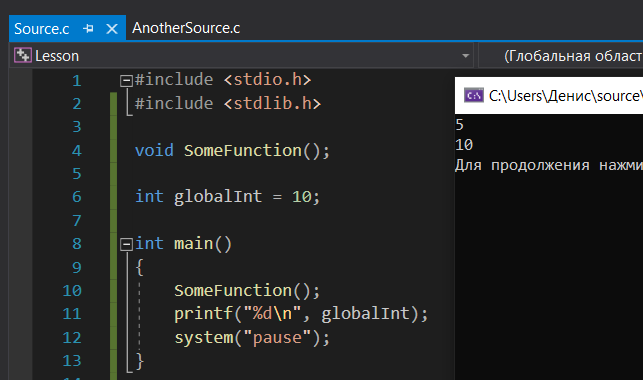
Пример:

Данный код приводит к появлению ошибки **переопределения** – одной из повседневных ключевых ошибок: 

Один из способов решения (*необязательно всегда в первую очередь использовать именно его)*: модификатор **static** у одной из глобальных переменных в одном из файлов:

globalInt во втором файле стал виден только в нем и первый файл перестал его видеть.

Результатом такого изменения становится корректное выполнение программы:

**17.2) Случайные числа, srand, rand.**

Функция **rand** генерирует псевдослучайное число в промежутке [0;32767].

Функция **srand** задает семя случайности на основе которого **rand** генерирует псевдослучайное число.

Для повседневных задач использование **rand** при **srand** по семени функции **time** для случайных чисел допустимо, но оно не допустимо для использования в качестве криптографической функции.

Подробнее:

[rand | Microsoft Docs](https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/reference/rand?view=msvc-160&viewFallbackFrom=vs-2019)

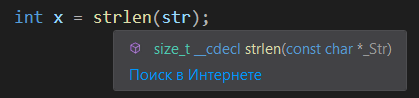
**18)** **Базовое понятие функций.**

Функция – блок кода выполняющий определённую функцию, созданный для логического разделения задач, дабы не выполнять весь код в точке входа.

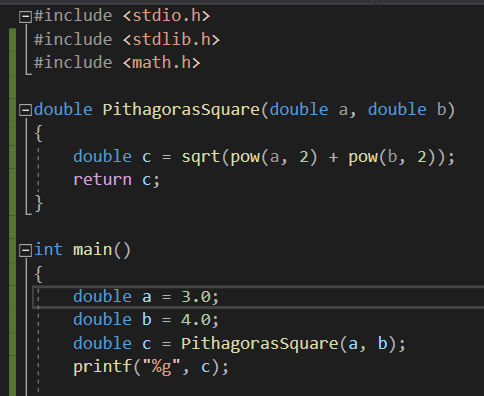
Функции характеризуются двумя параметрами: возвращаемым значением, и аргументами.

**Одинаковые типы данных можно приравнивать между собой. Если ваша функция возвращает тип данных int – её можно приравнять к переменной, типа int. Если char\*\* - её можно приравнять к char\*\*. *Тоже самое для любых типов данных.***

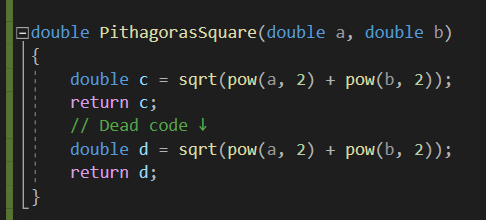
***Примечание: В основном приравнивание не одинаковых типов данных влечет за собой неопределенное поведение. Также передача в функции не тех параметров для каких они предназначены тоже влечет за собой неопределенное поведение. Старайтесь не допускать этого.***

***Типы возвращаемого значения и аргументов можно узнать при наведении на функцию:***

Пример использования функции:

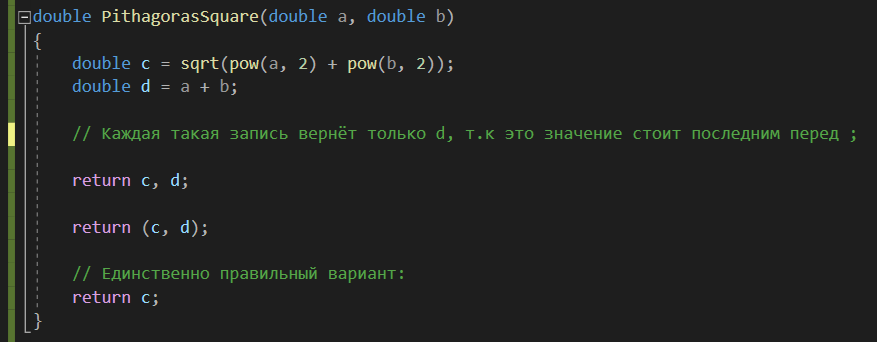
Функция при вызове приостанавливает ход текущей подпрограммы на время своего выполнения.

**Каждая функция для передачи управления в ту точку, откуда ее вызвали, использует ключевое слово return.**

**Return является точкой завершения работы любой функции. Любой код написанный после него является бесполезным.**

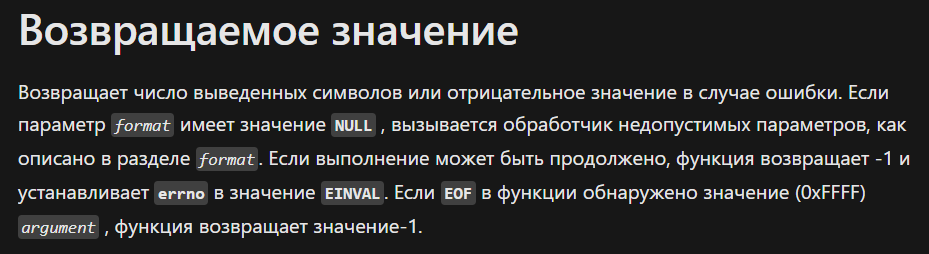
Пример:

Return возвращает (*если возвращает*) *только один* объект**:**

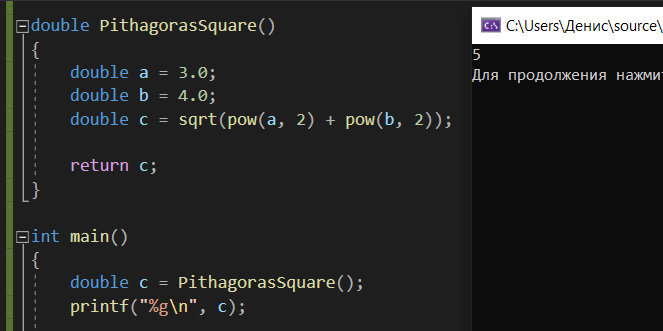


Любую функцию, можно вызвать даже не приравнивая её возвращаемый тип к какой-нибудь переменной такого же типа, тогда если в результате её работы было что-то получено, это значение просто потеряется. Функции с типом void можно вызывать просто по ходу программы, не приравнивая их ни к чему.

*Большая часть функций стандартных библиотек имеют в качестве возвращаемого значения int, или синонимичное к нему обозначение. Зачастую нужно это для отлова ошибок, если таковые случились в результате выполнения функции. Все возвращаемые значения, (если таковые есть) описаны в документации.  
Пример: возвращаемые значения функции стандартной библиотеки printf:*



Аргументами функции может передаваться что – угодно, включая «ничего». В таком случае вызов функции происходит через пустые круглые скобки.



**19)** **Виды памяти: стек, глобальная/статическая память, куча.**

Любая программа или приложение тем или иным способом **всегда** задействует память.

**Стек:**

Память стека используется для хранения любых адресов объектов и точек возврата(*Подробнее в 22-23*).

Как только шаг программы попадает на введение нового объекта, – в стеке выделяется адрес для доступа к этому объекту. Переменные создаваемые в текущей области видимости добавляются в стек, при выходе из этой области – удаляются.

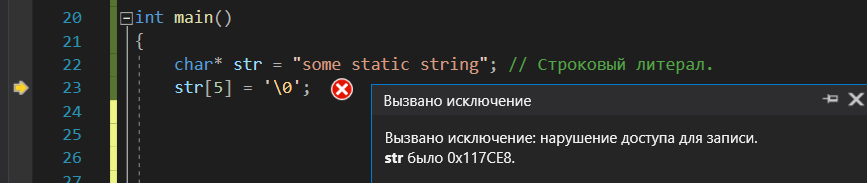
**Статическая память:**

Статическая память – это область памяти, выделяемая при запуске программы до вызова функции main из свободной оперативной памяти для размещения глобальных и статических объектов.

Примером использования статической памяти служат статические массивы (№10, №12.1), и **строковые литералы**.

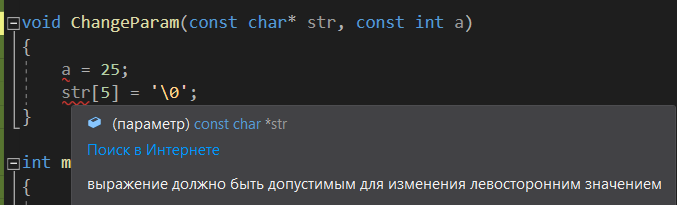
**Строковыми литералами** называются статические строки – строки с содержимым, сохранённым в статической памяти.

Попытка изменить строковый литерал влечет за собой **ошибку доступа (*записывать за пределы массивов, или в статическую память - плохо*)**:

**Замечание: все созданные объекты и адреса для обращения к ним существуют в стеке. Разница лишь в том НА какую память они указывают.**  
Подробнее: [Модель памяти в языке Си — Википедия (wikipedia.org)](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8_%D0%B2_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B5_%D0%A1%D0%B8)

*Также под частичное определение статической памяти могут попадать константы:*

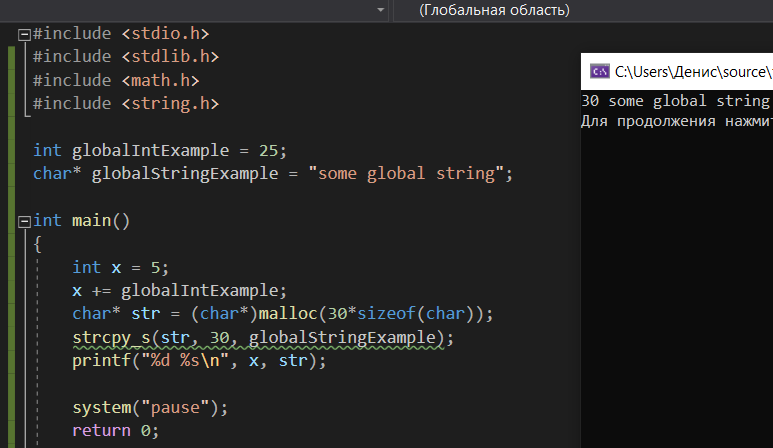
**Константы** – объекты имеющие при объявлении ключевое слово **const** сообщающее о том что такие объекты не изменяются и не должны изменятся извне на протяжении своего существования:



**Глобальная память:**

Такая память хранит в себе объекты наивысшей области видимости, и их могут видеть любые другие объекты и взаимодействовать с ними. Объекты в такой памяти объявляются вне любых других областей видимости.

Пример использования:



**Куча или Динамическая память:**

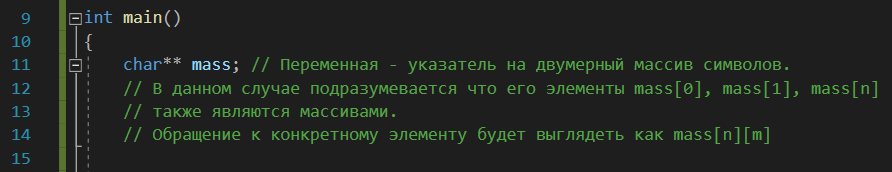
Самая большая по допустимому для использованя объёму память, использование её подразумевает системные запросы (malloc, realloc) на ее выделение для тех или иных целей. Предоставляемая системой память всегда требует очистки.

**20) Указатель на указатель. Многомерные динамические массивы.**

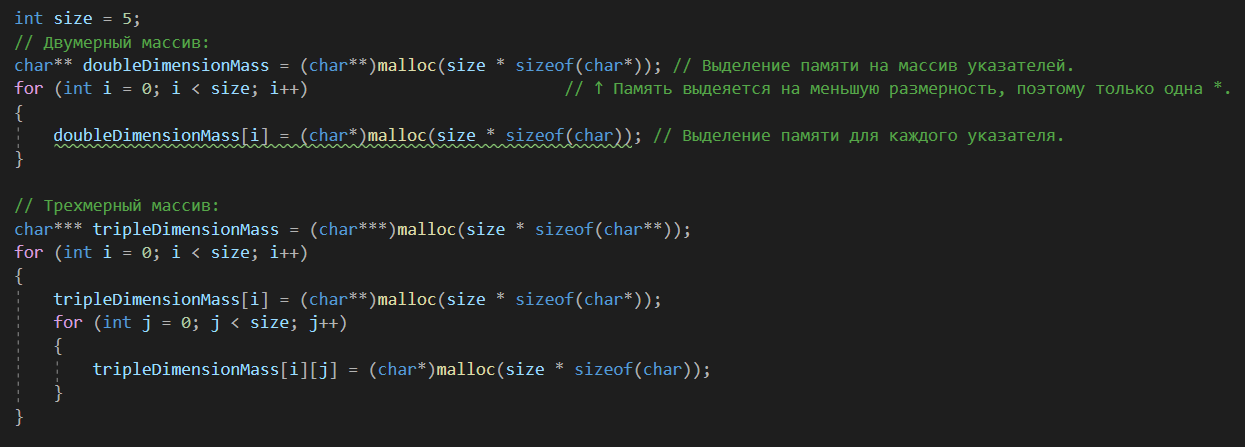
Многомерные динамические массивы, например трехмерные, являются массивами массивов массивов, иначе говоря, указателем на указатель на указатель. Для двумерного это просто указатель на указатель.

*Если речь идет про работу с массивами, воспринимать кол-во звездочек у типа переменной легче как размерность массива: char\* - одномерный массив символов, char\*\* - двумерный массив символов, char\*\*\* - трехмерный и.т.д.*

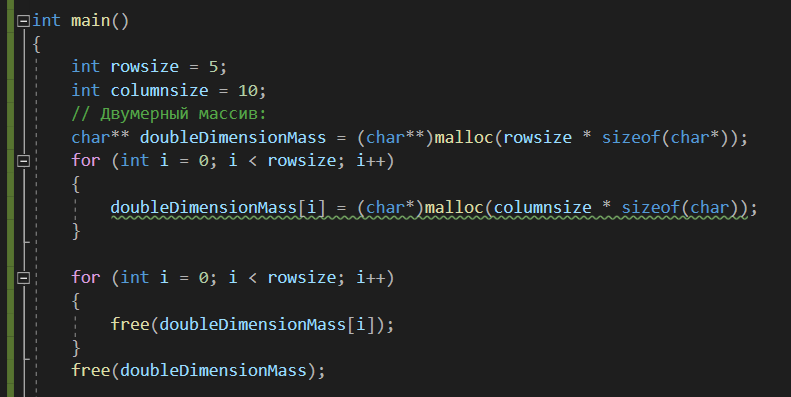
Если речь идет про двумерный массив, это значит что каждый элемент такого массива – является указателем на одномерный массив.

  
Для работы с n-мерными массивами требуется выделение памяти под каждый его элемент кроме последнего.

Пример:



Очищается память в **обратном порядке** относительно её выделения. Сначала память очищается по указателям в индексах, и только после очищается сам указатель на массив.



В таком случае память чистится по той же переменной, по которой она выделялась. В примере это rowsize.

Если вы будете чистить её больше – вы сломаете кучу. Если меньше – будет утечка памяти. Если чистить с начала весь указатель – вы просто не доберетёсь до памяти по индексам. Следите за этим.

**21) Параметры функции main: argc,argv,env.**

Исходная функция main, которая является точкой входа по умолчанию, тоже имеет в себе аргументы. Эти аргументы становятся валидными когда программа запускается через коммандную строку как exe файл с далее перечисляемыми параметрами.

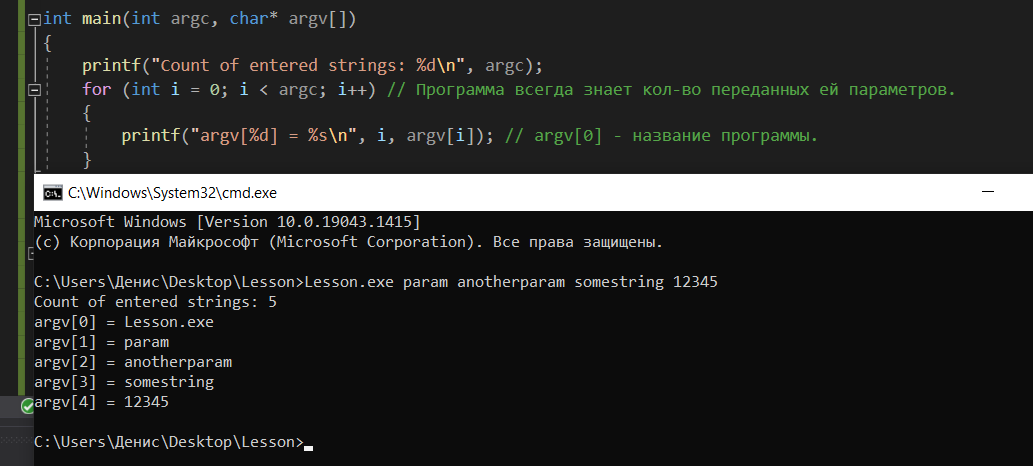
Int argc (argument count) – Название параметра принятого по умолчанию, который хранит в себе кол-во далее идущих строк.

Char\* argv[] (argument vectors) – Название параметра принятого по умолчанию, который хранит в себе переданные параметры в виде строк.

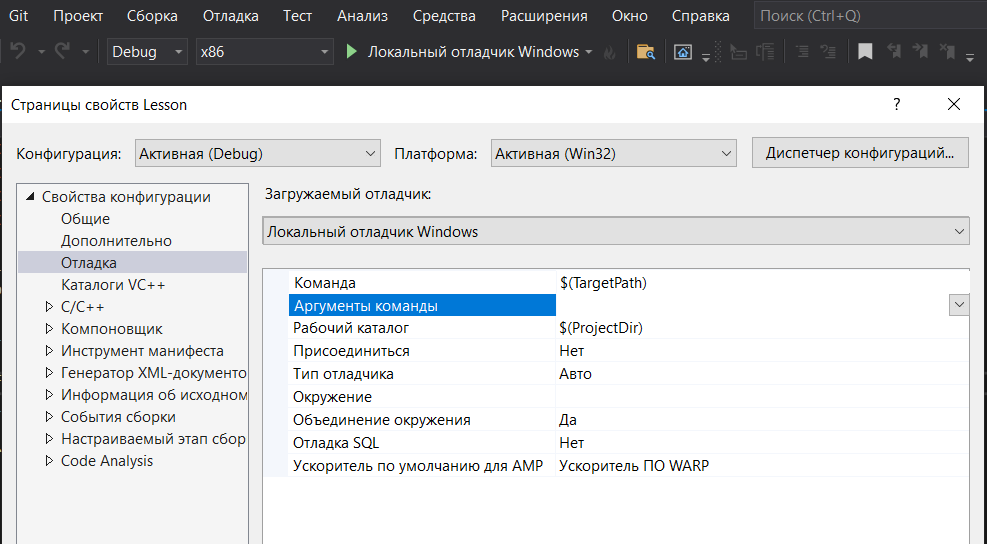
Env – параметр окружения, доступный в для использования в Visual Studio, хранящий в себе данные о системных параметрах системы, которые она использует:

[Переменная среды — Википедия (wikipedia.org)](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D1%8B)

Пример:



Запускать для отладки через Visual Studio программу, использующую аргументы командной строки, можно через настройку этих параметров:

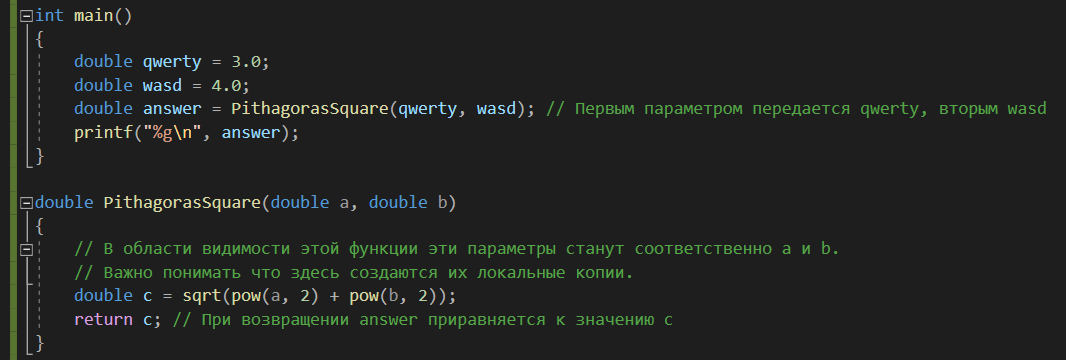
Проект->Свойства->Отладка->Аргументы команды

**22) Функции. Передача параметров по-значению. Передача массива в функцию.**

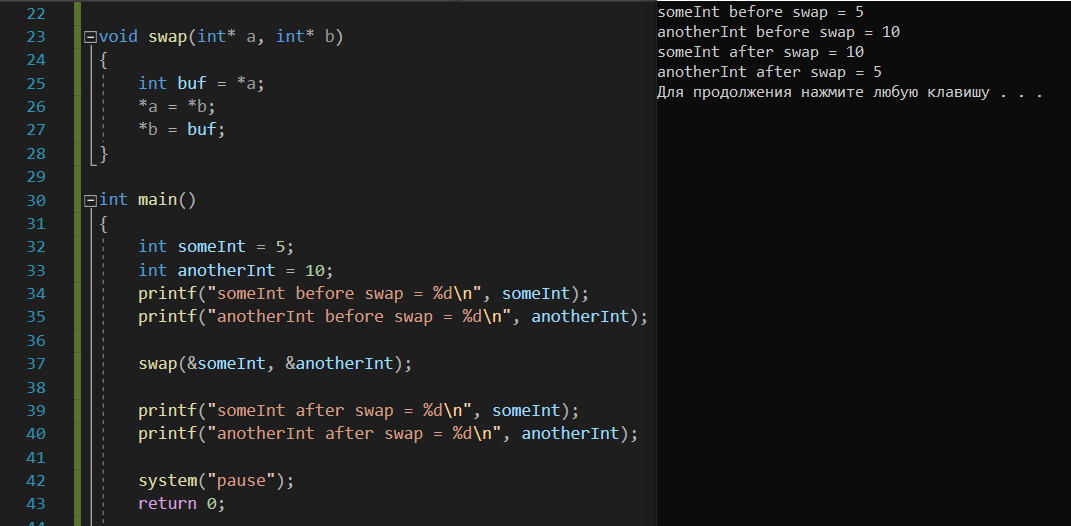
*\*Основное в 18*

Обычно подпрограммы могут передавать аргументы двумя способами. Первый называется передачей по значению. Данный метод копирует содержимое аргумента в формальный параметр подпрограммы. Изменения, сделанные в параметре, не влияют на значение переменной, используемой при вызове.

Пример передачи по значению:

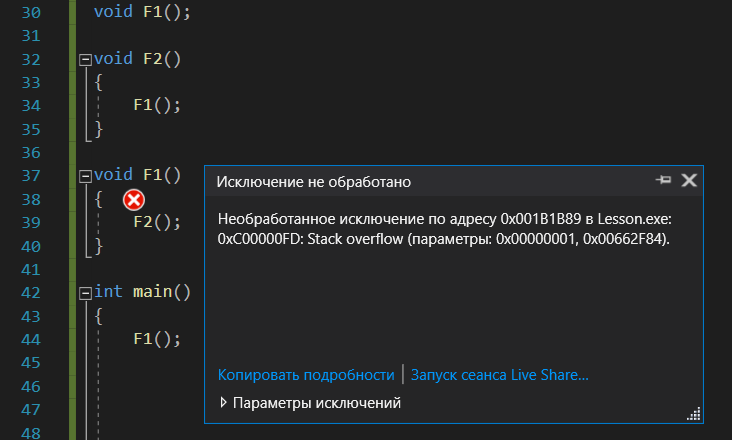


Передача по указателю является вторым способом передачи аргументов. В данном методе копируется адрес аргумента. В подпрограмме адрес используется для доступа к настоящему аргументу, используемому при вызове. То есть, изменения, сделанные в параметре, влияют на содержимое переменной, используемой при вызове.

Пример передачи по указателю: 

**Про «точки» возврата:**

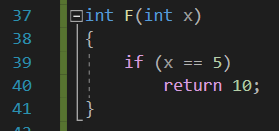
Каждая функция при своем вызове, должна знать куда ей вернуться после того как она завершает свое выполнение: для этого она оставляет в стеке «точку» возврата.

Именно по этой причине вы не можете делать так: 

Стек переполняется точками возврата подпрограмм и дальнейший ход программы нарушается.

Ровно по той же причине нельзя вызывать main() во время работы всей программы (*даже если у вас там красочно расписано меню какого – нибудь приложения и.т.п, и вам кажется, что это хорошая идея – поверьте, так делать не стоит.*)

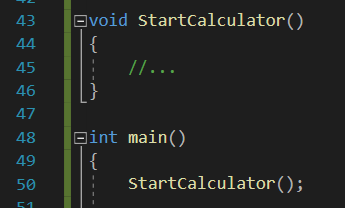
*Личные советы:  
1) Если ваша функция подразумевает условные конструкции: следите за тем чтобы ваша функция в* ***любом*** *случае что-нибудь возвращала(если она что-то возвращает), во избежание неопределенного поведения. Вот так делать не стоит:*



*(Если после какой нибудь составной части, будь то условная конструкция или цикл стоит всего одна инструкция – фигурные скобки для такого можно опускать)*

*2) Нет смысла в двух разных функциях, если всё, чем они различаются – локальные наименования аргументов/переменных (см Передачу по значению). Специализировать функции под конкретные аргументы – очень редкое, и не факт, что нужное решение.*

*3) Логическое разделение проекта не только на файлы, но и на void функции - переходы может повысить(а может и понизить) читамость кода:*



*В таком случае важно следить за понятной и адекватной номенклатурой.*

*4) Функции это хорошо, но старайтесь не переусердствовать с ними: не нужно сносить каждые две строчки в функцию, ровно как и не следует все делать в main.*

*Они не должны быть функциями типа (return a+b) но и не должны быть в 500 строк (хотя, конечно, для всего могут быть частные случаи).*

*5) Чаще читайте документацию, старайтесь разобраться в том или ином поведении. Помимо того что вы будете её постепенно учить(чтобы в дальнейшем открывать её реже) вы также повысите свои навыки в решении задач: решение будет приходить в голову быстрее и вы будете тратить на задачи меньше времени.*

[Средства разработчика, техническая документация и примеры кода | Microsoft Docs](https://docs.microsoft.com/ru-ru/?view=msvc-160)

*6) Часто может случится так что для конкретной задачи уже существует нужная вам функция (либо их комбинация) или библиотека. Старайтесь искать (или спрашивать) и грамотно их использовать, цените своё время, велосипед изобретать не нужно. За вас (вероятно) это давно уже сделали.*

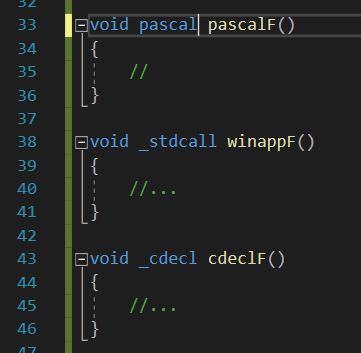
*7) Если у вас что-то не работает: в первую очередь проверяйте объявления и определения функций, импорт библиотек, возвращаемые значения.   
Часто в том, что программа не работает, виновата далеко не она :)*

**23) Различные соглашения о вызовах функций в Си. Подробно cdecl.**

*Если к этому моменту прочтения вы уже почувствовали себя джедаем – можете попробовать понять это, иначе вернитесь чуть сюда позже:*

[Соглашение о вызове — Википедия (wikipedia.org)](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B3%D0%BB%D0%B0%D1%88%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BE_%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2%D0%B5)

Типизация соглашения о вызове для каждой функции определяется в её начале:



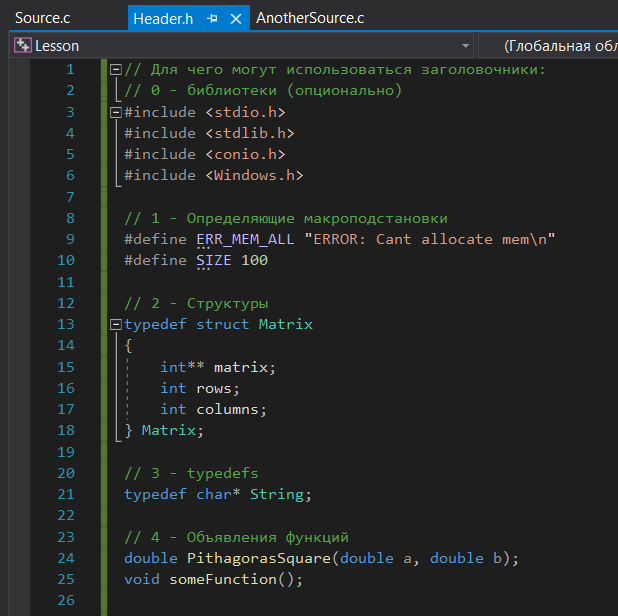
**24) Многофайловая разработка. Раздельная компиляция. Include Guards.**

Многофайловая разработка нужна для повышения качества. От архитектуры которая закладывается еще на этапе проектирования зависит многое, в том числе читаемость кода и качество всего проекта, и одна из её частей – многофайловая разработка.

Файлы в Си разделяются на два типа – файлы исходного кода с расширением '.с', и заголовочные файлы '.h'

Исходный код разделяется на блоки по функционалу или другим частям, зависит от архитектуры.

Заголовочные файлы созданы для хранения ресурсов, которые используют файлы исходного кода.

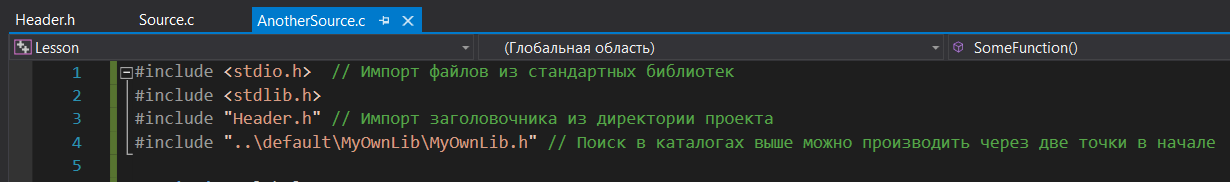
Основное использование заголовочников: 

**Директивы препроцессора:**

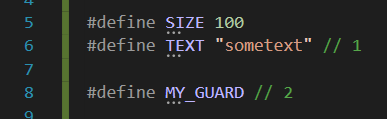
#include – Директива которая копирует содержимое файла к которому обращается и просто вставляет его в данный файл.

Если имя файла написано в угловых скобках <> препроцессор будет искать его в стандартных директориях.  
Если имя файла указано в кавычках “” препроцессор будет искать его среди файлов проекта.

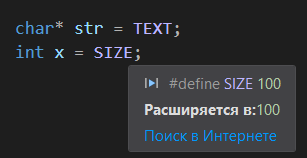
Пример:



#define – Директива макроподстановки или определения:



В первом случае обе директивы являются макроподстановками, иначе говоря – все появления далее в коде слов SIZE и TEXT будут заменены на соответственные значения 100 и строку “sometext”:



(*Может быть полезным почитать*): [Макросы в С и С++ / Хабр (habr.com)](https://habr.com/ru/post/546946/)

Во втором случае MY\_GUARD является просто определенным макросом и приравнять его к чему-нибудь не получится.

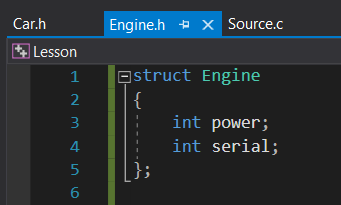
Определяющие директивы #define являются очень важным и гибким механизмом в языке. Например с их помощью можно защищать проекты от переопределения и использовать кроссплатформенную компиляцию.

**Include Guards или Макрозащита**

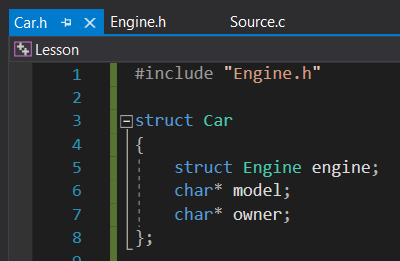
Как уже писалось ранее: переопределение – ошибка которая возникает, когда ваш проект при сборке находит два одинаковых имени, и не может определится какое из определений является единственно верным.

Импорт незащищённых заголовков может вызывать переопределение, если этот заголовок включается сразу в несколько файлов исходного кода.

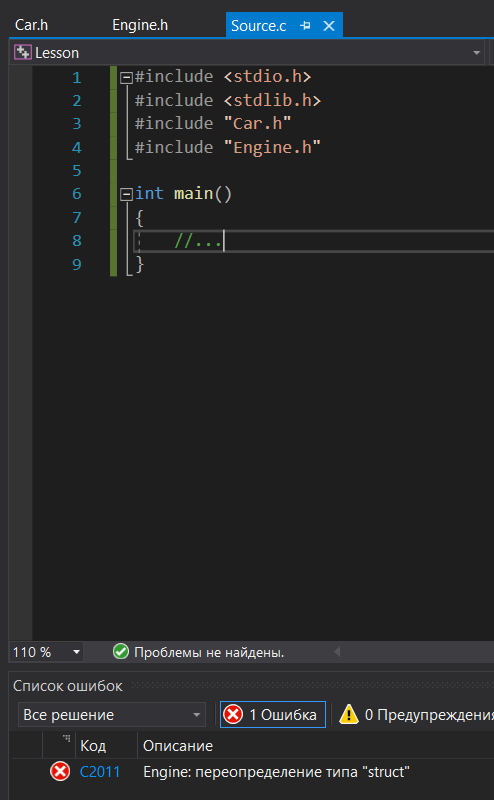
Допустим, есть код который описывает структурами машину и то из чего она состоит: например двигатель, и саму машину:



В таком случае сама машина включает заголовочник двигателя:

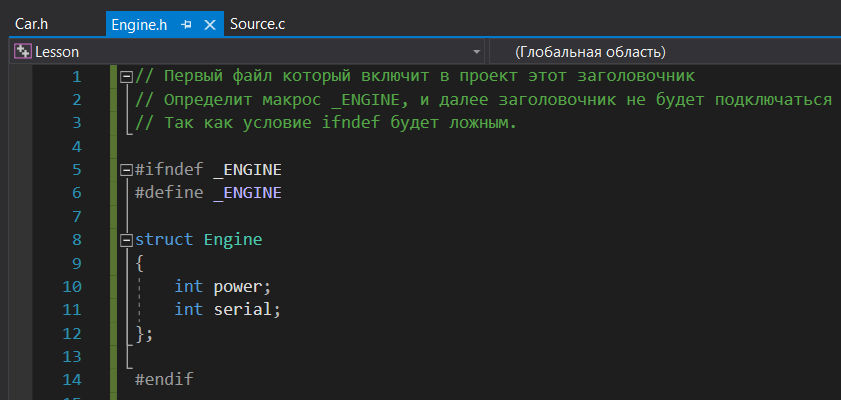


Далее если файлы исходного кода включают оба заголовочника, в результате мы получаем ошибку:



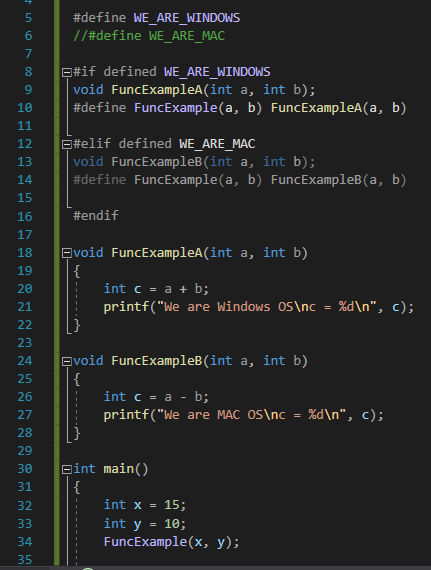
Быстрое решение проблемы вы могли видеть или использовать:   
Директива **#pragma once** следит за тем чтобы файл в котором она использована подключался строго один раз.

Важное для понимания решение проблемы: использование определяющих директив вместе с #ifndef и #endif:



*P.S: Все стандартные библиотеки защищены точно таким же образом, поэтому их подключение в нескольких файлах не вызывает никаких проблем.*

**Кроссплатформенность и условное включение:**

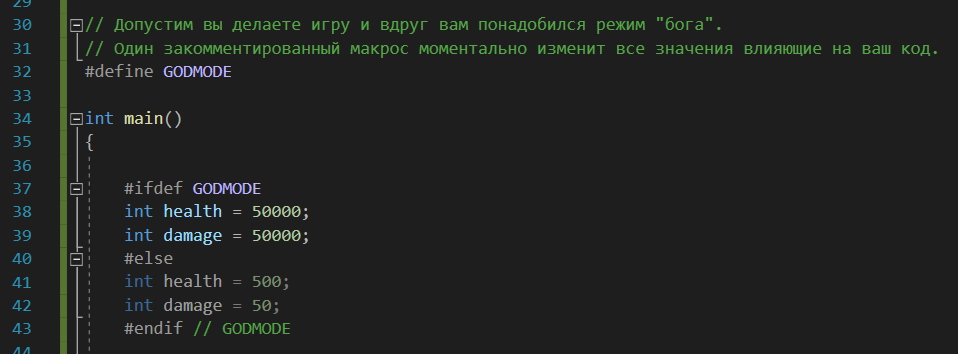
Те же условные конструкции if, else if, else которые мы используем в коде также можно использовать через # вместе с #define, #ifdef, #ifndef для реализации кроссплатформенности:

В данном случае объявляется макрос который (условимся) определяет нашу систему, и в зависимости от него также объявился и **протитип функции**, и макрос на её реализацию.

Стоит закомментировать 5 строку и раскомментировать 6 - и программа без малейших других изменений покажет уже другой результат.

Ровно таким же методом можно делать ваш код условно включаемым.

Например:



**25+26) Компиляция, линковка. Команды компиляции и линковки. Консоль разработчика VS, синтаксис gcc, cl, link.**

*Эти два билета я делаю самыми последними, и мне стало уже очень лень пытаться придумывать умные слова, поэтому давайте за меня это сделает хабр и документация:*

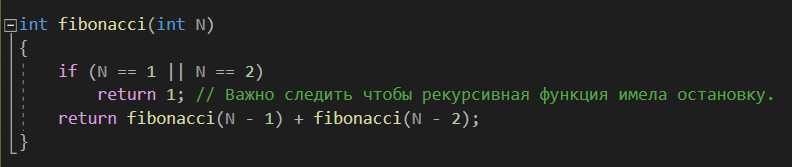
*Этапы компиляции:* [Процесс компиляции программ на C++ / Хабр (habr.com)](https://habr.com/ru/post/478124/)

*Компиляция руками:* [Пошаговое руководство. Компиляция программы на языке C из командной строки | Microsoft Docs](https://docs.microsoft.com/ru-ru/cpp/build/walkthrough-compile-a-c-program-on-the-command-line?view=msvc-160&viewFallbackFrom=vs-2019)

**27) Рекурсивные функции.**

Рекурсивные функции – функции которые вызывают сами себя.

Пример: Функция Фибоначчи.



**28) char\*\*\* mass; Правильное выделение и освобождение памяти.**

*\*Все в 20*

**29) Практика использования heap: 4 варианта int\*\* mass.**

[csu-lang-C/main.c at main · Ginmaru-Gin/csu-lang-C · GitHub](https://github.com/Ginmaru-Gin/csu-lang-C/blob/main/11-Heap-strings-structs/lecture-code/10.11.2021/main.c)

Уменьшение кол-ва вызовов malloc связано с тем что этот системный вызов (в сравнении с другим кодом) очень медленнен, и чем вызовов будет меньше – тем оптимизированнее будет ваша программа.

**30 + 31) Измерение времени выполнения участка кода. clock, time. Перенаправление ввода/вывода (<,>,>>).**

*Я толком и не придумал что мне написать, поэтому я просто оставлю это здесь:*

[csu-lang-C/12-Crossplatform-execution-time/lecture-code/17.11.2021 at main · Ginmaru-Gin/csu-lang-C · GitHub](https://github.com/Ginmaru-Gin/csu-lang-C/tree/main/12-Crossplatform-execution-time/lecture-code/17.11.2021)

[clock | Microsoft Docs](https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/reference/clock?view=msvc-170)

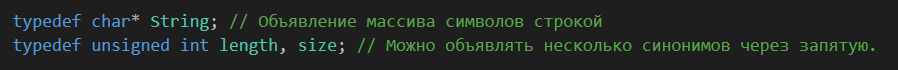
**32) Директивы препроцессора. #include #define #ifdef #ifndef.**

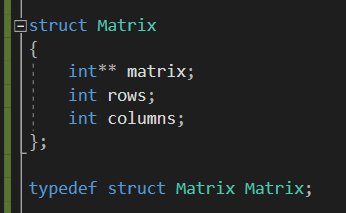
*\*Все в 24*

**33) Оператор typedef. (104,156 K&R).**

**typedef** (type defenition) – оператор объявления «синонимов» или псевдонимов к существующим типам данных. Объявления typedef можно использовать для создания более коротких или более понятных имен для типов, уже определенных в языке C или объявленных пользователем.

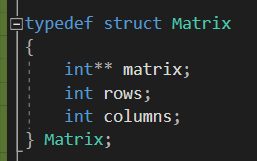
Пример хорошего использования:



Также прекрасным использованием может служить введение синонимов для объявлений структур:

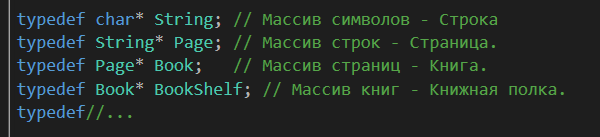
Таким образом объявлять экземпляры структур можно без ключевого слова **struct**, используя только ключевое слово **Matrix**.

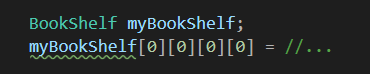
*Но так как все знают что программисты – люди ленивые, есть способ сократить и эту запись:*



*Примечание: Иногда для массивов порядка больше 2(если вдруг потребуется работа с ними) есть резон вводить синонимы для простоты понимания. Т.к воспринимать двенадцатимерные массивы головой физически очень сложно, синонимы могут исправить это дело.*

*Пример:*

*Главное не забывайте о том что на самом деле это все равно четырехмерный массив символов:*



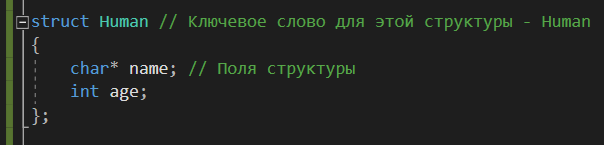
**34) Условное включение. Кроссплатформенность в Си. #if #elif #else (104 K&R).**

*\*Все в 24*

**35) Простые структуры. Ввод/вывод структур.**

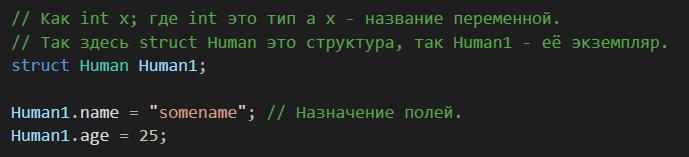
Структуры – сложный тип данных (*и наивысший архитектурный синтаксический инструмент для использования в Си*) являющий собой способ описания объектов на основе примитивов.

Структуры объявляются с помощью ключевого слова **struct** и в качестве определения использют **поля:**



Поля структуры располагаются в памяти в том порядке, в котором они объявлены.

Ввод осуществляется через создание **экземпляров** структуры и назначение полей:

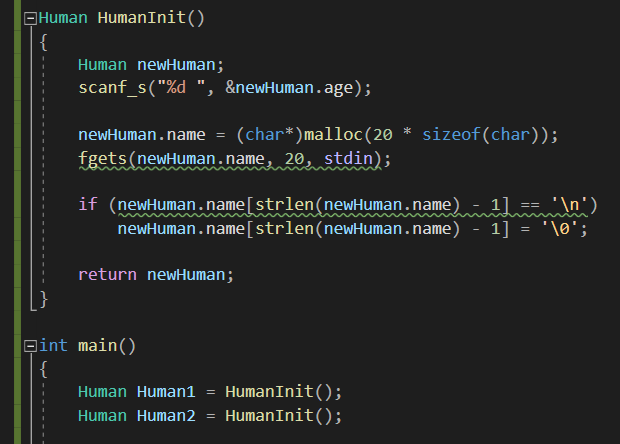


Также поля можно инициализировать сразу же при создании экземпляра (в данном случае уже был использован typedef(№33)):



Принципиального отличия назначения полей между первым и вторым способом нет, главное помнить что порядок при инициализации совпадает с порядком объявления полей в самой структуре.

Если экземпляров нужно несколько, а назначение их полей происходит (например) через стандартный ввод, то функции могут облегчить эту работу:



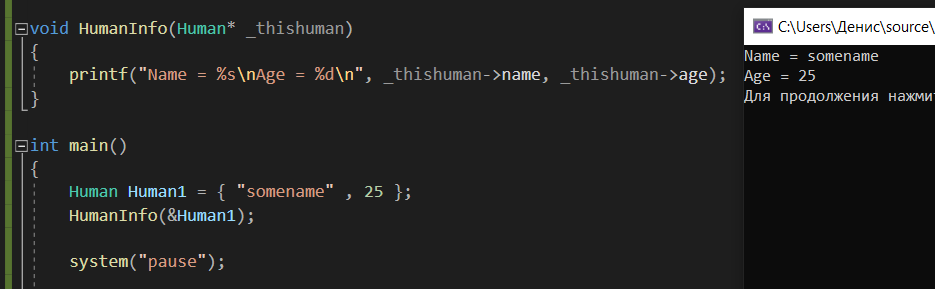
Так как Human это наш новый тип данных - функция может его возвращать.

Почему это работает? Каждый вызов этой функции, будет создавать внутри её области видимости экземпляр структуры, и каждый раз по новому адресу. После назначения полей, при возвращении, этот экземпляр просто приравняется к таким же экземплярам в функции main, но каждый из них будет лежать в разных адресах памяти.

Таким образом мы можем инициализировать сколько угодно экземпляров.

**Передача структур:**

Структуры передаются через указатель:

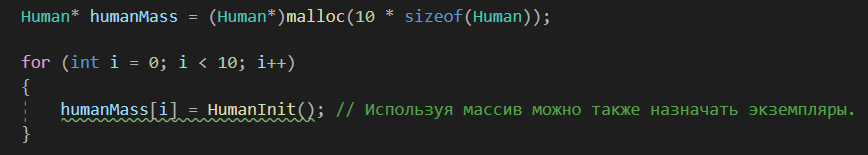


В таком случае обращение к полям тоже происходит через указатель ‘->’

**Массивы структур:**

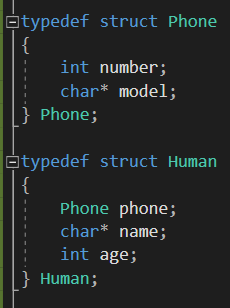
Массивы структур ничем не отличаются от обычных массивов, схема такая же:

Создать указатель и выделить память под тип данных:

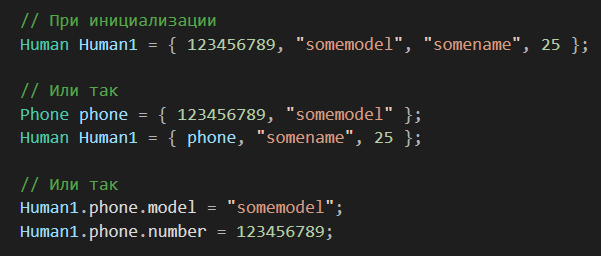


**Вложенные структуры:**

Вложенная структура – структура, полем которой является другая структура:

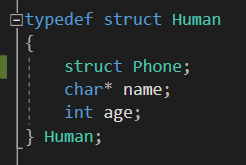


Назначение полей в данном случае имеет все тот же синтаксис:

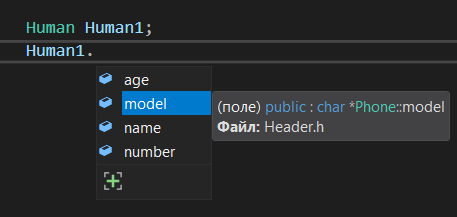


Разница лишь в том что во втором случае для этого создаётся дополнительный объект, к которому можно обращаться не только через вложенную структуру, но и напрямую.

Также поля вложенной структуры можно не объявлять отдельным словом, таким образом родительская структура будет наследовать поля дочерней:

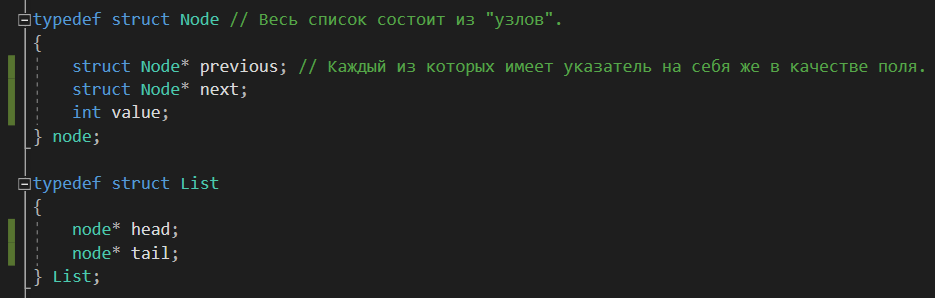


Таким образом можно обращаться к полям напрямую через экземпляр родительской структуры:

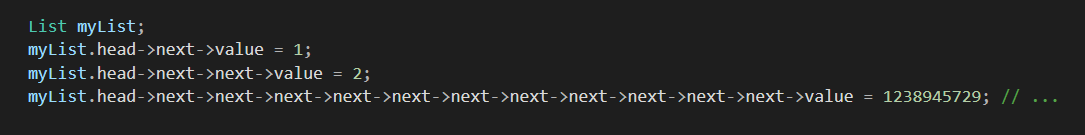


**Структуры с динамическими полями:**

Иначе говоря, структура полем которой является **указатель** на эту же структуру. Примером может служить структура данных Список:



В таком случае назначение/добавление таких полей может продолжаться до бесконечности:



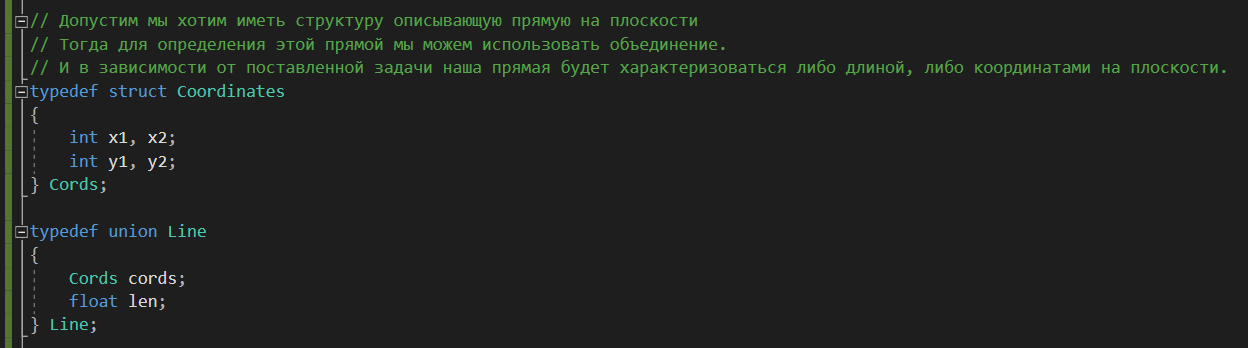
(*Этот код на самом деле не запустится, он здесь только для визуального понимания происходящего*)

**36) Объединения (158 K&R), битовые поля (159 K&R), перечисления (230 K&R).**

**Объединения** — это сложный тип данных, который позволяет различным полям разделять общую память. Таким образом в объединении может храниться только одно из полей. Размер объединения равен размеру наибольшего поля.

Объединения объявляются с помощью ключевого слова **union.**

Пример:

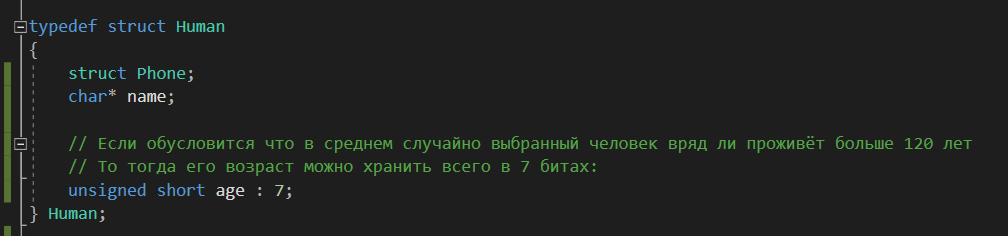


В таком случае оба поля структуры будут лежать в одной и той же области памяти, и использовать в текущий момент времени мы cможем только одно из полей. Важно понимать что тогда занимаемая объединением память будет расчитываться относительно максимально объемного типа данных в нем – в данном случае структуры Coordinates которая занимает 16 байт.

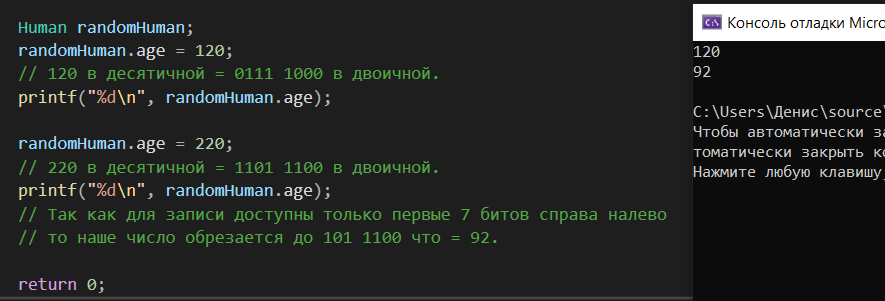
**Битовые поля**

Используя структуры, можно упаковать целочисленные компоненты еще более плотно, чем это было сделано с использованием массива.

Набор разрядов целого числа можно разбить на битовые поля, каждое из которых выделяется для определенной переменной. При работе с битовыми полями количество битов, выделяемое для хранения каждого поля отделяется от имени двоеточием:

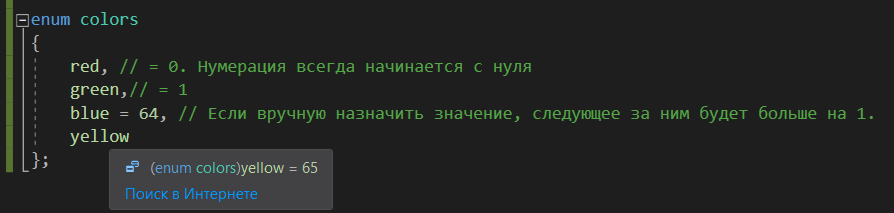


Что будет если постараться записать больше?



**Перечисления**

Перечисления позволяют определять синонимы к целочисленным типам данных и они созданы для улучшения читабельности кода:



**37) Простые алгоритмы. Бинарный поиск. Поворот матрицы.**

[csu-lang-C/main.c at main · Ginmaru-Gin/csu-lang-C · GitHub](https://github.com/Ginmaru-Gin/csu-lang-C/blob/main/17-Structures-pratice/lecture-code/ImportantAlgorithms/main.c)

*Как по мне: другого способа выучить билет, кроме как попытаться понять происходящее – просто нет.*

**38) Структуры с динамическими полями. Массивы структур. Структуры внутри структур.** *\*Все в 35*