Оглавление

[Переменные 3](#_Toc137312819)

[Const,constexpr 4](#_Toc137312820)

[Условия и циклы 5](#_Toc137312821)

[Отладка 8](#_Toc137312822)

[Массивы 9](#_Toc137312823)

[Функции 10](#_Toc137312824)

[Встраиваемая функция (**inline)** 11](#_Toc137312825)

[Перегрузка функций 11](#_Toc137312826)

[Шаблоны функции 12](#_Toc137312827)

[Область видимости/продолжительность жизни 15](#_Toc137312828)

[Указатели 17](#_Toc137312829)

[Ссылки 20](#_Toc137312830)

[Директивы препроцессора 23](#_Toc137312831)

[Динамическая память. new c++ 26](#_Toc137312832)

[Динамический массив 28](#_Toc137312833)

[String 32](#_Toc137312834)

[Приведение типов 33](#_Toc137312835)

[argc и argv + различие void main/ int main 34](#_Toc137312836)

[Словари и множества 35](#_Toc137312837)

[модификаторы Const 36](#_Toc137312838)

[Цикл foreach (range-based циклы) 37](#_Toc137312839)

[Лямбда-выражения 39](#_Toc137312840)

[Операторы 40](#_Toc137312841)

[Генератор случайных чисел 42](#_Toc137312842)

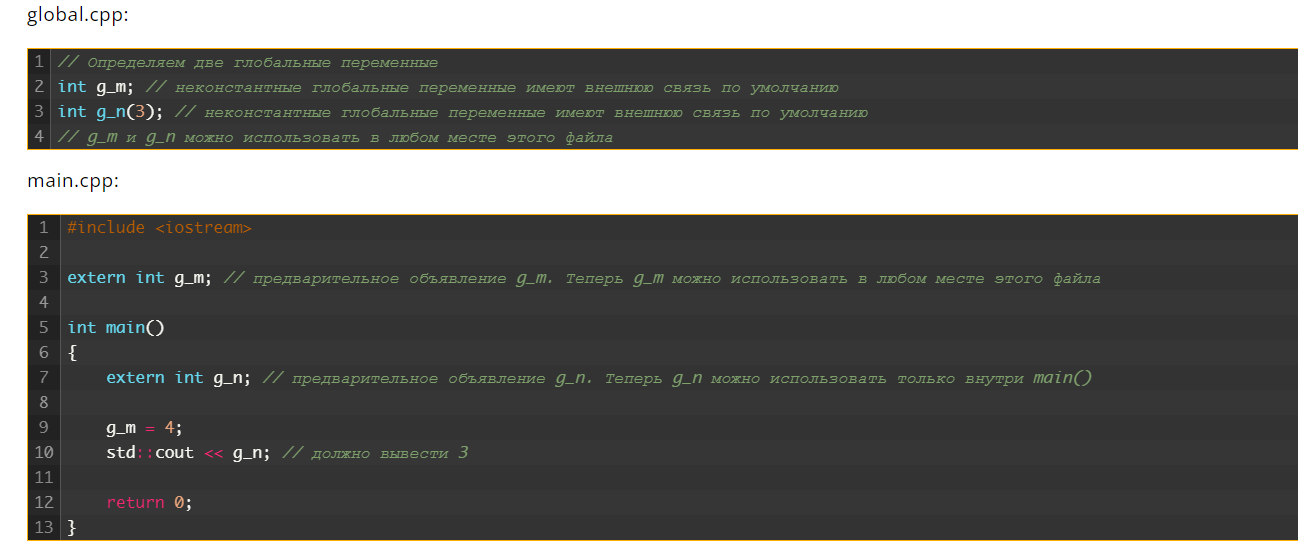
# Переменные

Переменные – именованная область памяти

**Глобальные переменные**

* Определены вне блока и имеют статистическую продолжительность жизни (создаются при запуске – уничтожаются при завершении).
* Можно использовать в ЛЮБОМ месте файла после объявления
* Объявляются по правилам ниже директив #include
* Если существуют две переменные (локальная и глобальная) с одинаковыми именами, то в приоритете будет локальная. Вызов глобальной осуществуляется при помощи оператора области видимости **::**
* Существуют внешние и внутренние переменные. По умолчанию неконстантные – внешние, константные – внутренние. Всё это относится и к функциям
  + Внутренняя – может использоваться в любом месте текущего файла, но не может использоваться вне его
    - Ключевое слово **static**
  + Внешняя – может использоваться и в файле, в котором определена, и вне его.
    - Ключевое слово **extern**
* Если предварительное объявление прописано вне блока, то применяется ко всему файлу, иначе – только к блоку

Пример



# Const,constexpr

Существуют два вида констант:

1. **Константа времени компиляции** – значение определяется во время компиляции программы (например, когда константу сразу задали: const n=10. Стоит учесть, что размер стекового массива константен времени компиляции)
2. **Константа времени выполнения** – значение определяется во время выполнения программы (например, когда мы сперва вводим с клавиатуры данные, заносим их в переменную, а потом присваиваем это значение нашей константе)

Ключевое слово **constexpr** – принудительно заставляет компилятор проверить на константность времени компиляции

# Условия и циклы

**if & switch-case**

* **Классический**

if (condition){

…

} else if (condition) {

…

} else if (condition && condition) { // логичекое и

…

} else if (condition || condition) { // логическое или

…

} else {

…

}

* **if в одну строку**

( condition ) ? true cond: false cond;

* **конструкция switch-case**

switch (condition){

case значение\_1: инструкция\_1;break;

case значение\_2: инструкция\_2;break;

……………

case значение\_N: инструкция\_N;break;

default: инструкция, если все «кейсы» не сработали;

}

**while**

* **classical while**

while (утверждение) // while True - rerun

{

…

}

* **do-while**

do

{

} while (утверждение); // while True – rerun

**for**

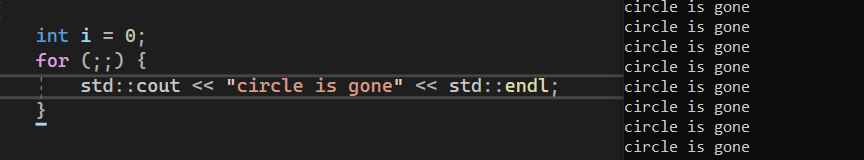
* **Простой цикл for**

for (initialization; condition; action) { // н: for (int i =0, i > 10,i++)

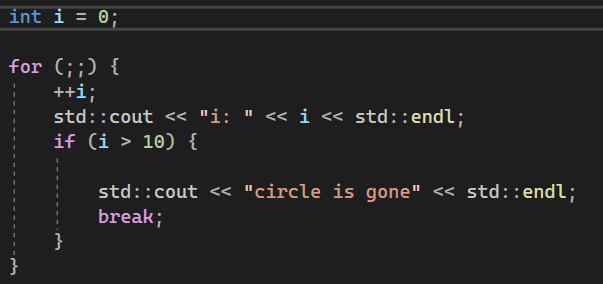
// тело цикла

}

Однако все параметры можно вынести и цикл будет работать или добавить еще больше переменных и условий туда







break – выходит из цикла

continue – переходит на следующую итерацию, пропуская оставшийся код в этой итерации.

# Отладка

f5 – начать отладку

f10 – перейти на следующую строку

Если навестись на переменную, то можно её закрепить на экране

Если имеется несколько точек останова, то можно еще раз нажать f5,чтобы идти не пошагово, а сразу перейти к следующей.

f11 – шаг с заходом – если при выполнении отладки нам нужно получить не сразу результат функции, а зайти в неё и посмотреть как она работает.

# Массивы

* Статический массив

тип данных[количество элементов];

int arr[5];

Второй вариант инициализации

тип данных название[] = {элементы}

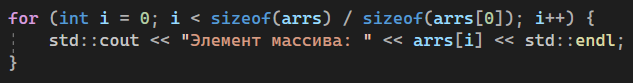
int arr[] = {1,2,3,4,5}

Автоматически опрелит размер массива (можно без равно!)

тип данных название{элементы}

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

Вариант через цикл



# Функции

Возвращаемое\_значение имя\_функции (параметры){

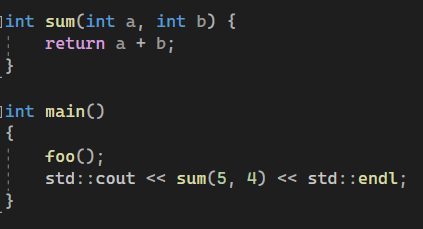
// тело функции //

}

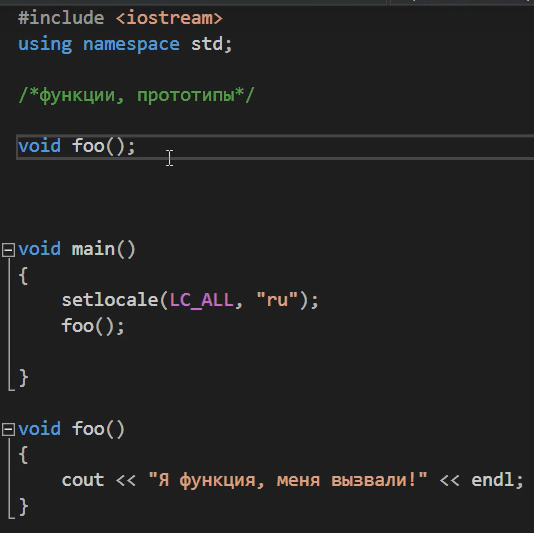
Если возвращаемое значение типа void, то функция ничего не возвращает

Если int, то вернет инт. Если double – вернет double и т.д…

Передачи параметров по значению



* Передается не сама переменная, а лишь значение. В самой функции создается копия с тем же значением. Воздействие на переменную в функции не несут для переменной извне.

 Прототипы функции

Объявляется имя функции и её тип в начале

заголовка, без её реализации. Необходимо для того,

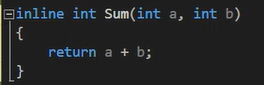
избежать ошибок, которые возникают при вызове

функции, когда та еще не реализована.

Прототип функции должен содержать

параметры функции.

## Встраиваемая функция (**inline)**



Используется для маленький функций. Не скомпилирует большую функцию.

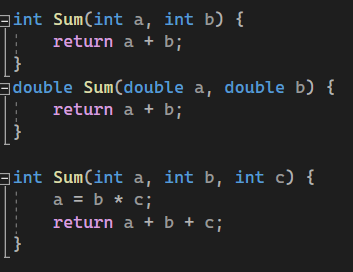
Иногда компилятор сам применяет функцию inline (зачастую, он делает это лучше, чем человек)

## Перегрузка функций

Когда у одного имени функции (или просто функции) более, чем одна реализация

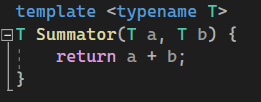
Могут отличаться внутри реализацией.

Может работать с разными типами (в примере int и double)



## Шаблоны функции

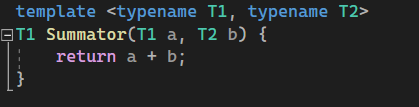
Используются, когда неизвестен зараннее тип. (даже собственно созданные типы)



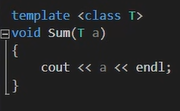
В данном примере функция отработает корректно, если в типе T, который мы будем передавать будет перегружен оператор сложения (+).

Стоит учесть, что T – это один тип, и он не может быть одновременно и int и double

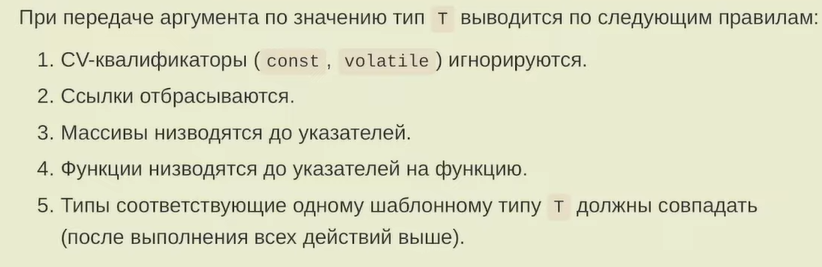
Пример реализации для двух типов данных.



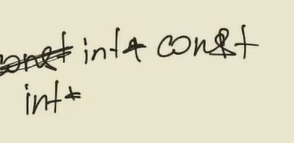
«typename» можно заменить словом «class», т.к. первое из языка С, а второе уже из С++



Стоит учесть, что при передачи по значению тип Т выводится по следующим правилам

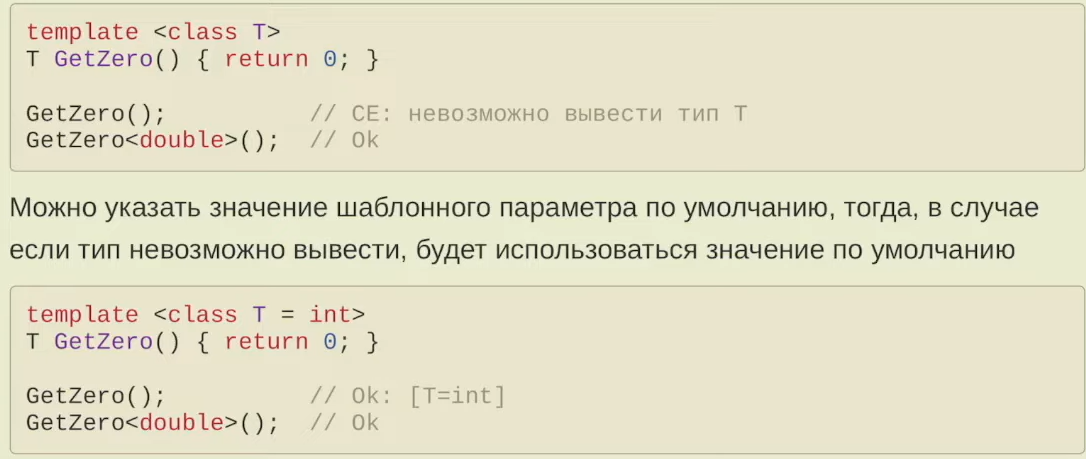


Однако, есть нюанс, связанный с тем, что отбрасываются только константы верхнего уровня.



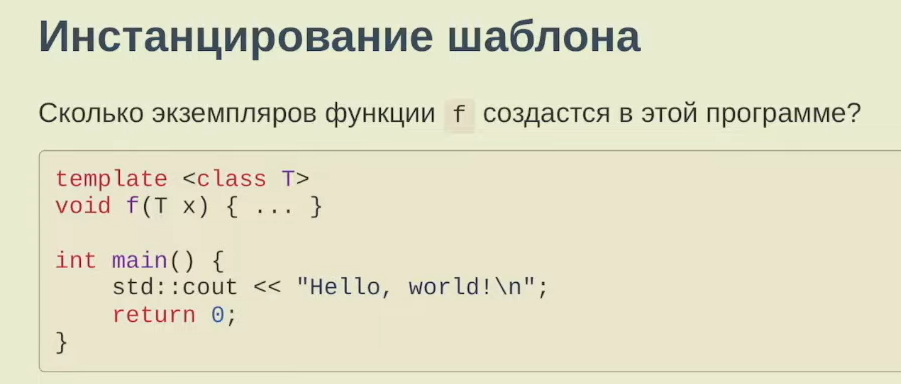
int \* const – преобразуется в int\*, но const int\* не преобразуется.

Можно задавать тип по умолчанию. Пригождается в тех случаях, когда мы вызываем шаблонную функцию без параметров



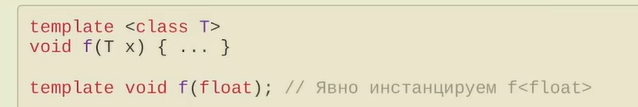
**Инстанцирование шаблонов**

Получение некого экземпляра шаблона



При написании кода и запуске программы обычная функция создает 1 свой экземпляр, даже если она не вызывается, т.к. она сразу перекомпилируется в ассемблерный код. В случае с шаблоными функциями, экземпляр **не создается**, пока мы не вызовем эту самую функцию, т.к. складывать int+int не то же самое, что int + double (ассемблер не будет знать как ему дейстовать просто напросто).

Можно явно инстанцировать шаблон



Зачем это нужно?

В многофайловых программах код сперва компилируется в каждом отдельном файле, а потом собирается целиком (линковка). То есть, если у нас в каждом отдельном файле используется void f(), то он будет создавать заново эту функцию в каждом файле. Для решения этого можно в отдельном файле инстанцировать данную функцию и уже её передать в остальные файлы . Иначе код будет «раздутым» и медленее компилироваться.

Проверка на корректность осуществляется в 2 этапа, т.к. до вызова нельзя проверить на корректность

1. Базовый синтаксис до инстанцирования
2. Полный код – уже после инстанцирования

Резюмируем:

* Шаблон функции – это не функция
* Генерация исполняемого кода не происходит, если шаблон ни разу не вызван
* В случае вызова шаблонной функции создается лишь нужная версия (с вызываемыми параметрами)
* Процесс генерации кода из шаблона называетс **инстанцированием шаблона**

# Область видимости/продолжительность жизни

**Область видимости** определяет, где можно использовать переменную

**Продолжительность жизни** (или «время жизни») определяет, где переменная создается и где уничтожается

**Связь** переменной определяет, относятся ли несколько упоминаний одного идентификатора к одной и той же переменной или нет.

Переменная, имеющая внутренние связи, называется **внутренней** **переменной** (или «статической переменной»). Она может использоваться в любом месте файла, в котором определена, но не относится к чему-либо вне этого файла. Ключевое поле **static.**

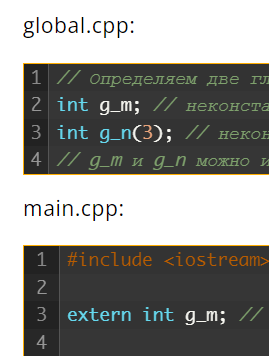


Переменная, имеющая внешние связи, называется **внешней** **переменной**. Она может использоваться как в файле, в котором определена, так и в других файлах.



Переменная *без связей* — это переменная с *локальной* областью видимости

чтобы использовать внешнюю глобальную переменную, которая была объявлена в другом файле, нужно записать предварительное объявление переменной с использованием ключевого слова **extern**

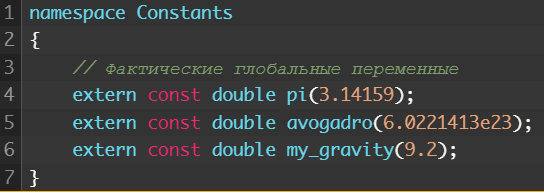


**Функции** имеют такие же свойства связи, что и переменные. По умолчанию они имеют **внешнюю связь**, которую можно сменить на внутреннюю с помощью ключевого слова **static**

**Файловая область видимости –** внутренние глобальные переменные

**Глобальная облатсь видимости** – внешние глобальные переменные

Есть одна неприятная особенность при подключении файлов заголовков. Если один и тот же заголовок подключить к 20 разным файлам, то он 20 раз скомпилируется. Чтобы избежать этого, в примере с константными переменными, достаточно объясвить их extern



* Глобальные переменные имеют глобальную область видимости и могут использоваться в любом месте программы. Подобно функциям, вы должны использовать предварительные объявления (с ключевым словом extern), чтобы использовать глобальную переменную, определенную в другом файле.
* По умолчанию, глобальные неконстантные переменные имеют внешнюю связь. Вы можете использовать ключевое слово static, чтобы сделать их внутренними.
* По умолчанию, глобальные константные переменные имеют внутреннюю связь. Вы можете использовать ключевое слово extern, чтобы сделать их внешними.
* Используйте префикс g\_ для идентификации ваших неконстантных глобальных переменных.

**Правила:**

1. Определяйте переменные в наиболее ограниченной области видимости

# Указатели

Указатели – это ссылочный тип данных, особенность которого заключается в том, что они содержат адрес другой переменной.

тип указателя должен совпадать с переменной, на которую он указывает.

\* - разименовывает указатель

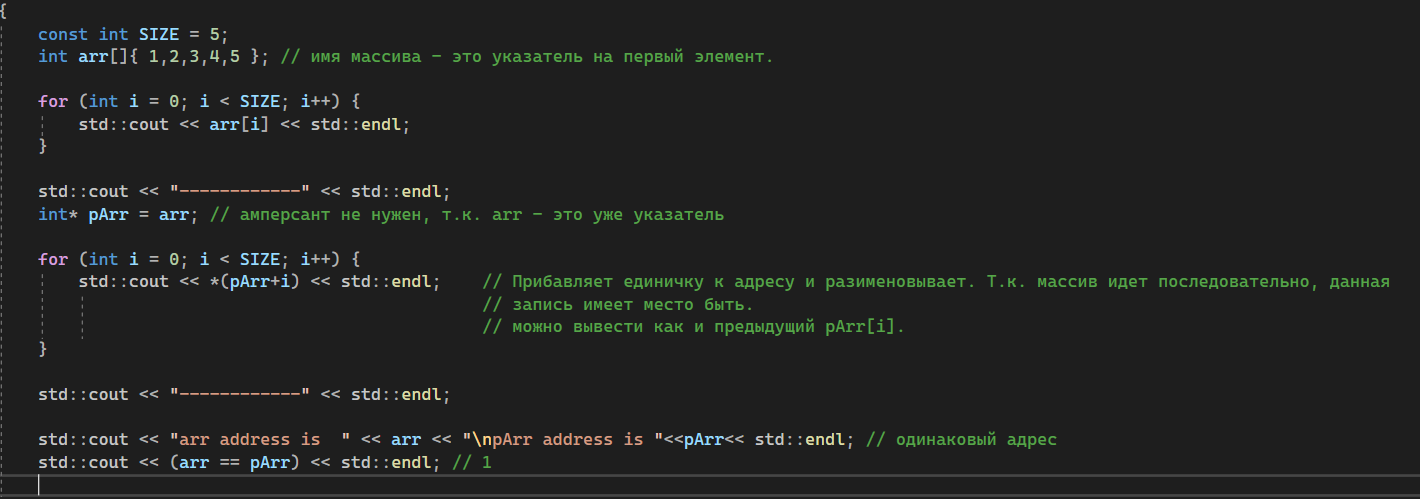
Указатели и массивы

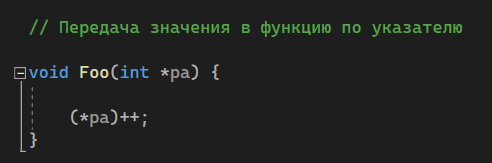
Имя массива – это указатель на первый элемент

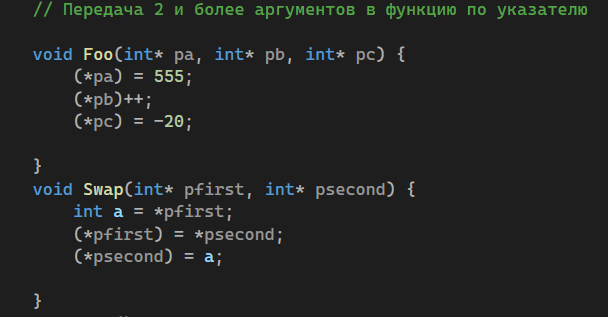
Делая указатель на массив не нужно ставить амперсант у правого операнда

Запись arr и pArr (\*pArr = arr) эквивалентна при работе с массивом

При прибавлении 1, то будет шаг соответствующий типу данных. (для инт 4 байта)





 Передача аргументов по указателю





Используются для того, чтобы выделять память динамически.

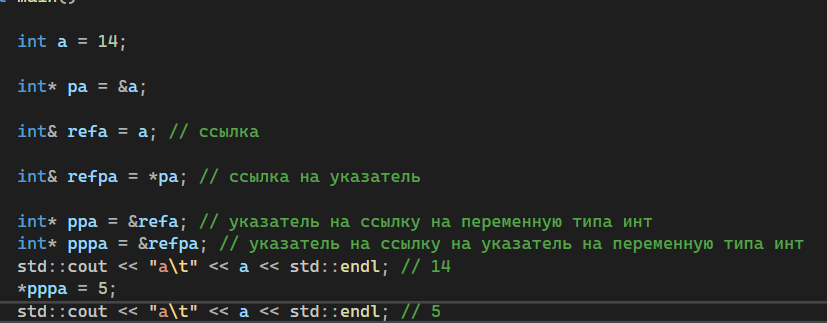
# Ссылки

Ссылка – это лишь другое имя нашей переменной.

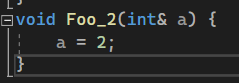
На данном скриншоте видна инициализации ссылки переменной и указателем

Ссылка ВСЕГДА должна быть инициализирована.

Не нужно разименовывать

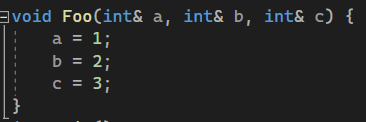


Передача параметра по ссылке в функцию



Как и с обычной переменной

C несколькими переменными также

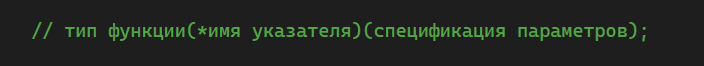




**Указатель на функцию**

указатель на функцию – это фактически переменная, которой мы можем присовоить ссылку на функцию

Эффект похожий на полиморфизм в ООП

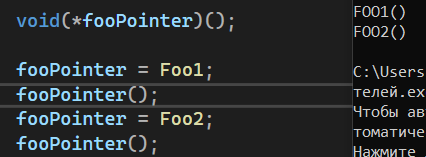
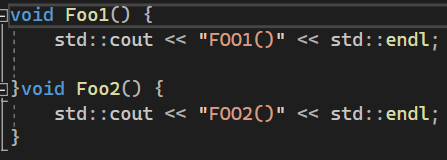


1 в 1 как с обычной функцией, только добавляется еще имя указателя

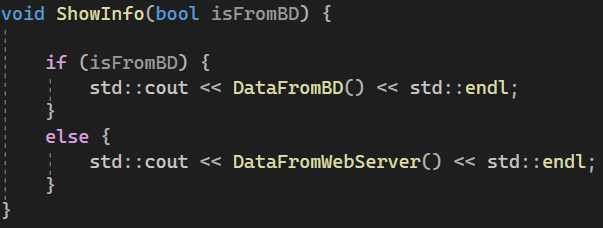
Тип функции должен совпадать с типом указателя. Например, если у нас тип функции = void, то мы не можем из неё ничего возвращать функцией strlen() и др.

Параметры в функции и параметры у указателя на функцию должны совпадать

1. Объявляем функцию void(\*fooPointer)(), которая не принимает никаких параметров (вторая скобочка)

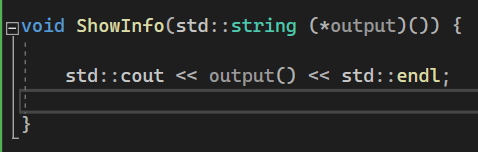


1. Помещаем в наш указатель \*fooPointer функцию Foo1;
2. Вызываем её
3. Помещаем в наш указатель \*fooPointer функцию Foo2;
4. Вызываем её
   1. Справа виден результат работы программы

Используется для масштабирования. Допустим, нам необходимо выводить данные из двух источников. Мы написали следующий код:

Однако перед возникает ситуация, что впоследствии будут добавляться источники данных. Это можно решить через switch-case, но придется вручную дописывать код, к тому же нужно будет другим сотрудникам знать, какой именно код (цифра) соответствует какому выводу, что неудобно.

Добавим указатель на функцию как параметр в нашу основную функцию ShowInfo()



Теперь, чтобы вызвать вывести данные на экран нам необходимо вывести эту функцию и написать в параметрах имя вызываемой функции.



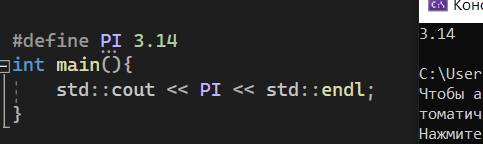
# Директивы препроцессора

Препроцессор – программа, которая проводит манипуляции с кодом еще до того, как он будет скомпилирован (предварительная обработка кода)

**define**

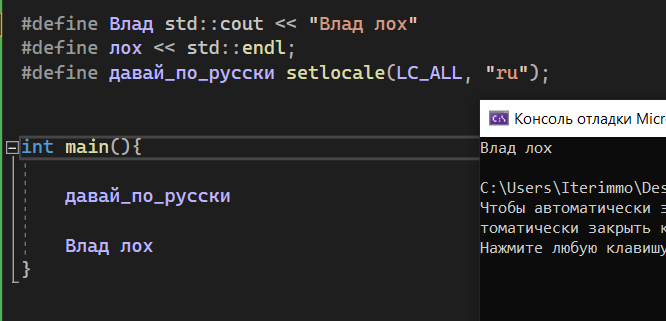
#**define** – заменяет символы на другие до того, как код скомпилируется.

Пример: код вывел 3.14 (справа)



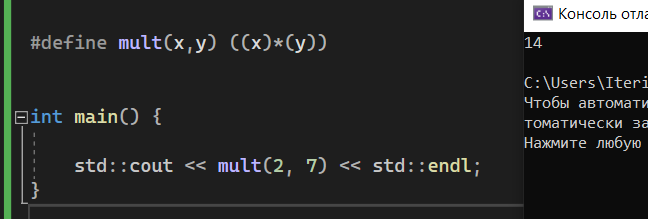
Можно использовать define для объявления констант

Еще один пример



!!! Важно следить за тем, чтобы наш define не был именем какой-либо функции, т.к. наш препроцессор заменит её имя на то, что содержится в define !!!

**Макрос-функция** (через define)



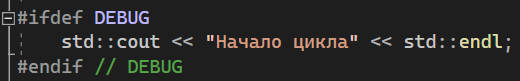
**Условная компиляция**

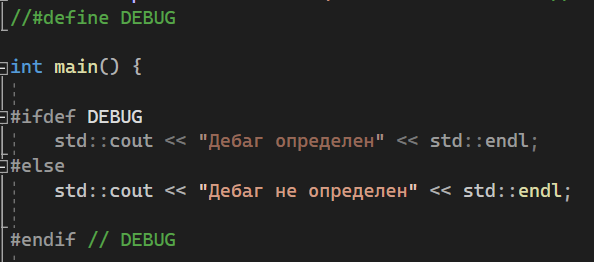
#ifdef-#endif (#else, #ifndef)

Используется для комментирования большого куска кода.

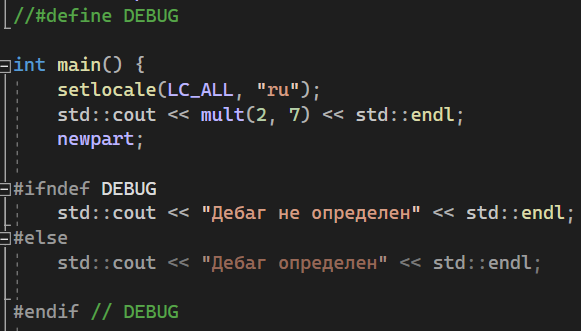
* Фактически показывает определен ли наш define с таким именем или нет. Если определен возвращает True, если нет, возвращает False. (для ifndef наоборот)

Для отладки, условно, нам необходимо вывести что-то, но это не должно содержаться в итоговом проекта, однако может понадобиться в будущем.

* + 1. Определяем в дефайне наш дебаг
    2. Пишем в тех местах, где нужно закомментировать #ifdef. Вторая часть – endif проставится автоматически
    3. Комментируем наш define DEBUG
    4. Код между #ifdef и #endif не будет отображаться
* Если мы хотим сделать развилку, то можно вставить #else, как и с обычным if-ом

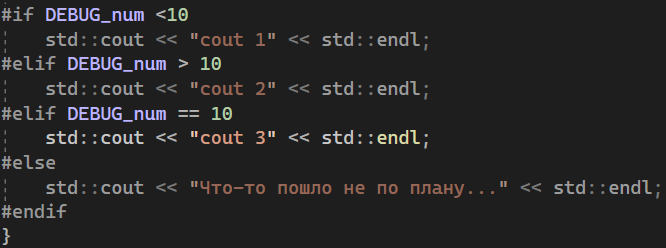


* если написать #if**n**def, то у нас будет ровно всё наоборот: если define определен мы выполним код, иначе попадем в else (если он сущетвует)



* Можно заменить ifdef на if. С обоими ифами можно сделать ветвление на #elif-ах, если у нас у define-переменной есть какое-либо значение





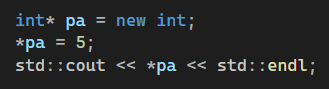
# Динамическая память. new c++

Взаимодействие с памятью.

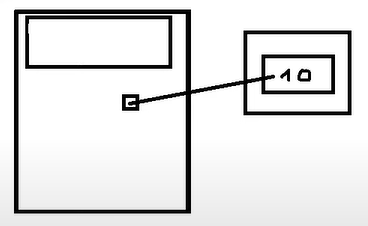
Не имеет автоматичекого сборщика мусора

**new**

Взаимодействие с памятью вне программы. Запрашивает доп.память у оперативки.



int \*pa – указатель на место в памяти вне нашей программы (обычно так хранят динамические массивы)

Далее мы разименовываем значение и даем ему значение 5

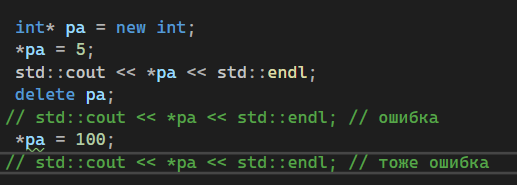
Выглядит это примерно так

**delete**

Очистка памяти.

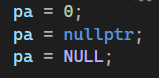


Ключевое слово «delete» “уничтожает” данные, которые хранятся в переменной, но не саму переменную.

Однако с этой переменной нельзя работать. Cам адрес сохранятся.

Система может взаимодействовать с этим адресом для других программ. Код ниже выведет адрес.

Чтобы очистить еще и адрес, необходимо обнулить наш указатель – Сделать это можно 3-мя способами:



Все они не идентичны по своей структуре.

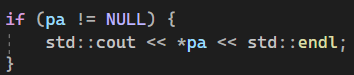
* nullptr – это отдельный тип данных. Лучше использовать его при обнулении, чтобы не вызвать какую-либо ошибку, связанную с тем, что в указатель запишется интовый ноль, а не произойдет обнуление.
* NULL – это директива. Она меняет на NULL на 0. Т.е. запись 1 и 3 идентичны в машинном коде.

 После инициализации

 После delete

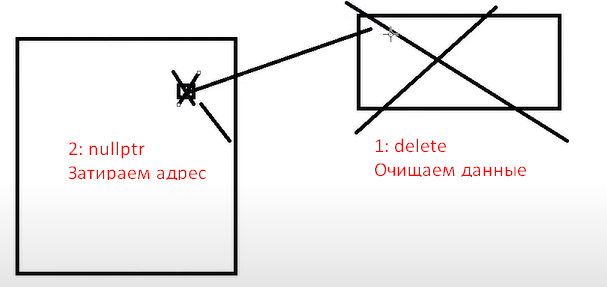
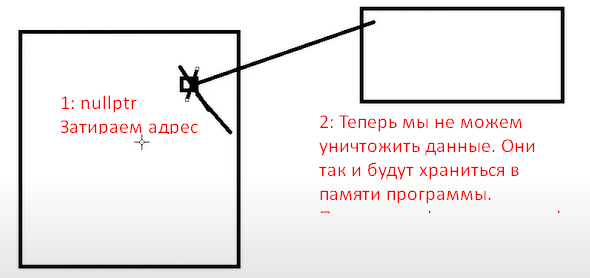
После обнуления (нулевой указатель)

Проверка на ненулевой указатель



!!! если удалить, но не обнулить переменную, то будет вызвано исключение, ибо удаленный (delete) указатель НЕ ЯВЛЯЕТСЯ нулевым.

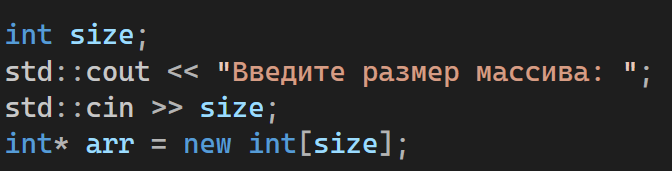
!!! Сперва delete, потом обнуление , иначе произойдет **утечка памяти** !!!



# Динамический массив

Инициализируем наш массив

Заметим, что переменная size не является константой и вводится с клавиатуры.

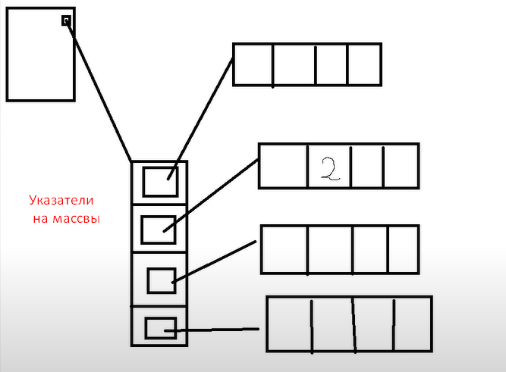


Удаляем наш массив. Как с указателями, но с добавлением []

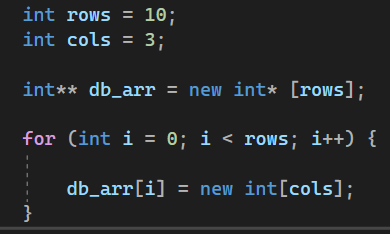


! Есть одна особенность, которая заключается в том, что массив с данными должен быть непрерывен в памяти. Т.е, если он не вместиться в определенный промежуток памяти, он будет исктаь другой свобожный, которого может не оказаться. Например, у нас массив на 100`000 элементов, а в памяти уже столько маленьких массивов поменьше (или простых переменных/указателей), что нет непрерывных 100`000 свободных адресов, из-за чего возникнет ошибка !

**Двумерный массив (динамической)**

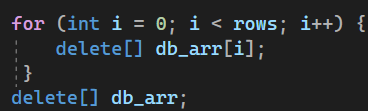
Двумерный массив – это одномерный массив одномернных массивов. Вместо переменной содержится указатель на другой массив, который содержит в себе несколько переменных

Инициализация двумерного массива будет иметь

следующий вид

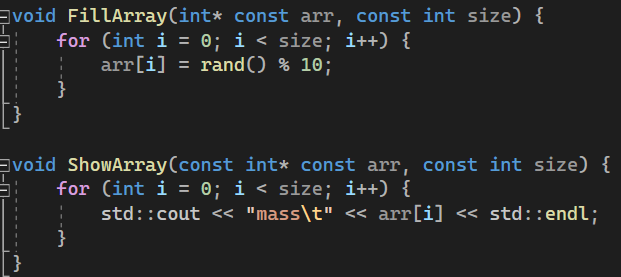
Сперва мы создаем указатели на указатели, а потом в эти указатели помещаем массивы

Чтобы не было переполнения очистка массива будет происходить следующим образом:



Сперва удаляются массивы-колонки, а уже потом основной массив.

**Передача в фунцию динамического массива**



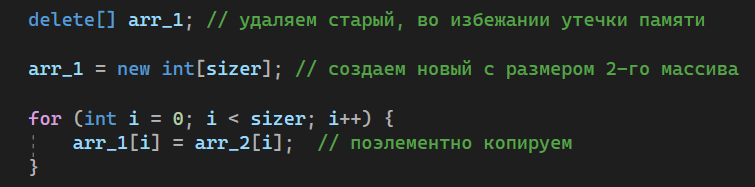
**Копирование динамического массива**



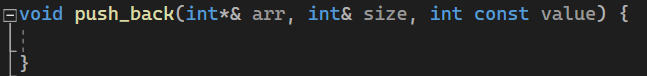
Происходит копирование следующим образом

1. удаляем старый, во избежании утечки памяти
2. создаем новый с размером 2-го массива
3. поэлементно копируем

В коде это выглядит следующим образом



**Увеличение динамического массива**



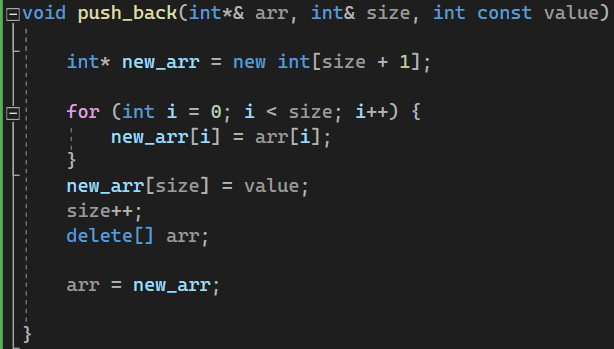
Изменение происходит удалением старого массива и созданием нового.

Передача массива по ссылке необходима, чтобы сделать так, чтобы массив мог указывать на новый адрес

Когда мы передаем в функцию указатель на массив, то мы можем взаимодействовать с данными массива. Однако сам указатель создается как новая переменная, т.е. это равносильно передачи по значению. Однако если использовать передачу по ссылке, то мы можем передавать адрес массива, где изначально лежали данные (передает адрес указателя самого)

При передачи массива в функцию по указателю в самой функции создается НОВЫЙ указатель, который указывает нам на те же ячейки в памяти, на которые указывал прежний, однако адрес самого указателя будет другой и мы не сможем с ним взаимодействовать (с данными сможем), а так как нам надо изменить и сам указатель и данные, то мы прибегает к использованию ссылки.

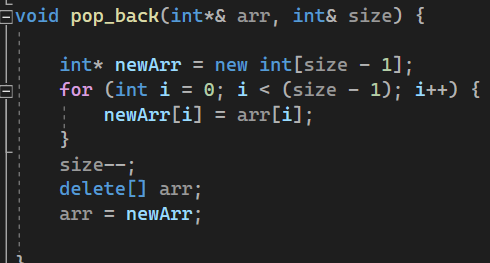
Данная функция будет выглядеть следующим образом

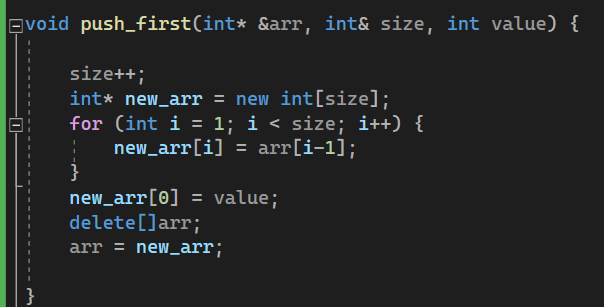
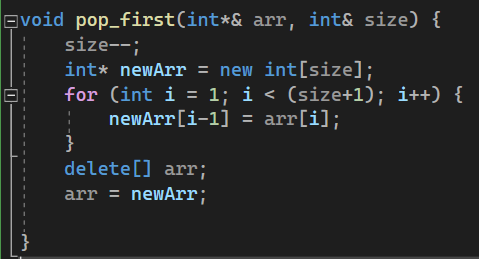


1. Создаем новый массив с размерностью +1 от старого
2. Переписываем все данные из старого в новый
3. На место последнего элемента ставим наш параметр Value
4. Увеличиваем переменную size на 1
5. Удаляем старый массив
6. Присваиваем старому указателю адреса новых значений массива.

**Уменьшение динамического массива**

1 в 1 как и с предыдущим

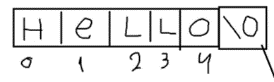


Добавление и удаление первого элемента массива

# String

В char хранится один символ в одинарных кавычках `d`



Класс String (далее тип данных) – это массив из char`ов.

В конце массива char`ов (строки) хранится оператор «тернарный нуль», который указывает на конец массива.

Для строки мы используем двойные ковычки ``symbol ``

Нельзя написать char ``d``, так как такая запись является массивом, в конце которого будет терменирующий нуль, т.е. de facto 2 символа

**Указатели и строки**

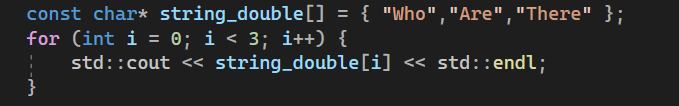
Запись массива через указатель выглядит следующим образом



Для него будут работать все операции с динамическими массивами

Без конст нельзя

Двойной массив



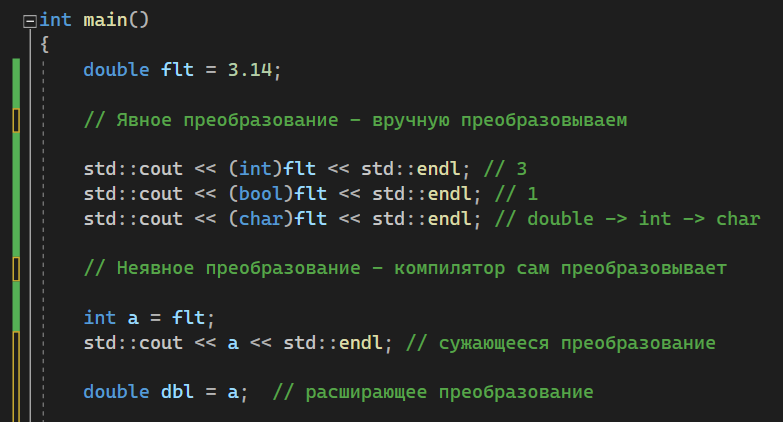
**Конкатенация строк**

Объединение строк: к первому массиву дописываются элементы второго

*strcat\_s(str1,str2);*



# Приведение типов



В арифметических операциях компилятор старается сохранить максимальное количество данных, поэтому будет использовать тип данных члена выражения с наибольшим размером для итогового хранения результата.

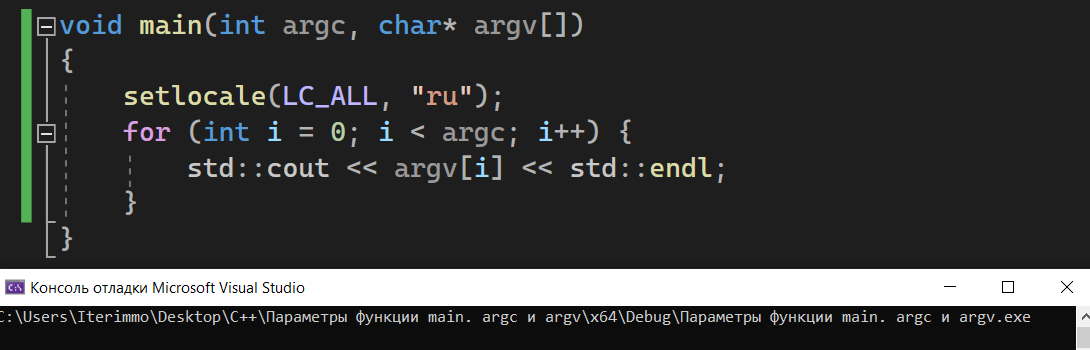
# argc и argv + различие void main/ int main

Необходимо прописывать в main, если мы допускаем, что в нашу программу будут передаваться какие-либо параметры. Если же в этом нет нужды, то и в этих двух переменных тоже её нет.

argc – arguments count – количество аргументов – сколько различных параметров мы передали (>1)

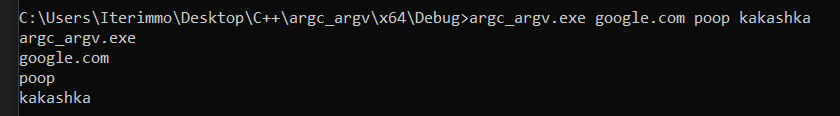
argv – arguments value – массив указателей на чар (строки).

Можно вывести циклом for все параметры, которая принимает программа



!Если вывести «пустую» программу, то она выдаст нам адрес .exe-файла!

Пример передачи параметров в функцию



Разинца между void main() и int main() заключается в том, нужно ли нам возвращать результат работы нашей программы тому объекту, которую нашу программу вызвал.

(если корректно, то 0)

int main() – классика. Необходимо писать именно так, к тому же ставить return 0 в конец программы

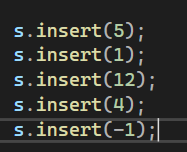
# Словари и множества

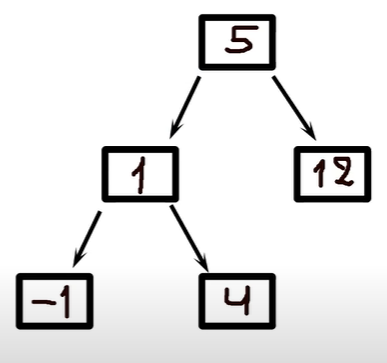
Множество содержит уникальные значения

Не может изменить число, так как это нарушит бинарное дерево, поэтому нам нужно сперва удалить, а потом вставить.

Необходимо подключить <str> через инклюд

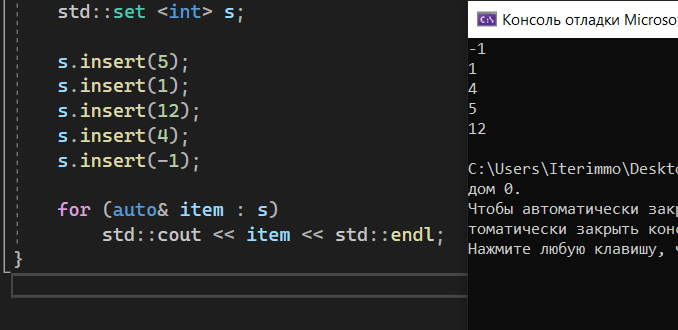
**Добавление элемента в множество**



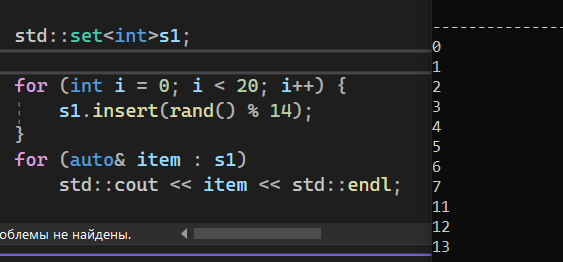
Сам принцип работы множества устроен по принципу бинарного дерева.

Все идет от корня. Если число меньше – налево, больше – направо.

При добавлении сперва 1, а потом 4. Ветвление 4-ки пойдет от единицы.

Вывод будет происходить в порядке возрастания

Пример рандомных чисел и «уникальности» каждого элемента в множестве



# модификаторы Const

При передаче неконстантной переменной в функцию, её можно объявить константной. То же самое относится и к указателям.

В данном случае динамический массив, попадая в функцию FillArray (заполнение массива), становится de facto статическим.



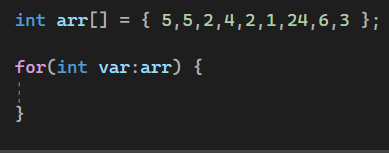
const arr запрещает добавлять новую память к уже имеющемуся массиву, а const int size запрещает изменять его размер.

если указать const int\* const arr – нельзя будет изменить значения массива.

# Цикл foreach (range-based циклы)

(не for\_each)

Имеет следующий синтаксис

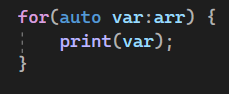


Ищем в какой-то **коллекции объеков** (arr) какой-то **конкретный объект** (var)

Это 1 в 1 как следующий цикл на питоне

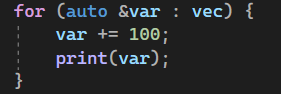
for col in dataset.columns:

Зачастую не пишут явно тип данных, а вставляют ключевое слово «auto»



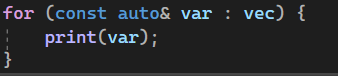
При такой реализации данные передаются **по значению.** Если изменить внутри цикла данные, то они не изменятся в основном контейнере

Решается это очень просто – передаем по адресу наши элементы



К тому же так гораздо выгоднее в плане памяти и времени, т.к. ничего не копируется

Если пробежаться по циклу, но не хотим, чтобы была даже в теории возможность изменять данные, но всё же не желаем копировать лишний раз объекты, то:

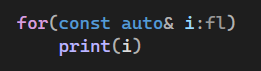


Наглядный пример использования

**Было:**



**Cтало:**



# Лямбда-выражения

Лямбда-функция – анонимная функция.

У неё нет имени, поэтому она и анонимная.

[](){} - внешний вид

* [] -
* () – принимаемые параметры
* {} – тело функции

# Операторы

Унарные – для действий нужен 1 операнд (а = -а)

Бинарные – 2 операнда (а = b + c)

Тернальные – 3 операнда

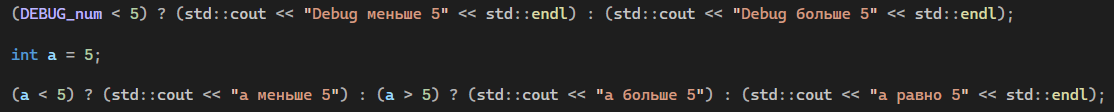
Инкремент ++

Декремент --

Постфикс (a--,a++)

Префикс (++a,--a)

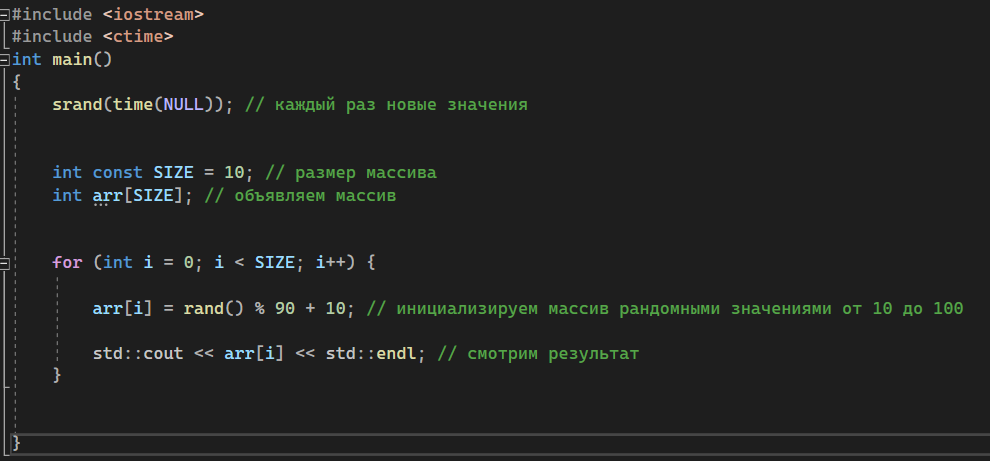
**Тернарный оператор**



(condition)?(true):(false)

Скобки необязательны, но с ними удобнее

# Генератор случайных чисел



Если нужно сделать так, чтобы после каждой отработки случайные значения оставались тем же, необходимо закомментировать srad()