**Тема №1: Алгоритм и его свойства**

**Алгоритм** - это конечная последовательность шагов, каждый из которых

выполняется за конечное время и приводит к решению конкретной задачи. В

языке программирования С++, алгоритмы широко используются для обработки данных, поиска решений и других задач. Важными свойствами алгоритма являются корректность, эффективность и понятность.

**Основные свойства алгоритма:**

1. **Корректность("Correctness"):**

• Алгоритм считается корректным, если он решает поставленную задачу для

всех

возможных входных данных.

• Корректность означает, что алгоритм ведет себя так, как ожидается, и даёт

правильные результаты.

1. **Определённость ("Definiteness"):**

• Каждый шагалгоритма должен быть однозначно определен и понятен. Нет

неоднозначности в том, что нужно делать.

1. **Входные данные ("Input'):**

• Алгоритм должен иметь определенные входные данные. Это могут быть

параметры функции или данные, с которыми алгоритм работает.

1. **Выходные данные ("Output"):**

• Алгоритм должен производить определенные выходные данные, связанные

входными данными и решением задачи.

1. **Конечность ("Finiteness"):**

• Алгоритм должен завершить выполнение за конечное время. Он не должен

зацикливаться или выполняться бесконечно.

1. **Эффективность ("Efficiency"):**

• Алгоритм считается эффективным, если он использует ресурсы (время и/или

память) эффективно, и его производительность удовлетворительна для данной

задачи.

1. **Понятность ("Clarity"):**

• Алгоритм должен быть понятным и легко читаемым. Чем проще

структура, тем легче его поддерживать и модифицировать.

и яснее его

1. **Универсальность ("Generality"):**

• Алгоритм может быть применен к различным входным данным или

использован для решения более общих классов задач.

1. **Модульность ("Modularity"):**

• Алгоритм может быть разбит на отдельные блоки или модули, что упрощает

понимание и поддержку кода.

1. **Оптимальность ("Optimality"):**

• Алгоритм может быть оптимизирован с целью улучшения

производительности, но оптимизация не должна приводить к ухудшению

читаемости или понимания кода.

**Тема №2: Этапы компиляции.**

1. **Препроцессинг ('Preprocessing'):**

* В этом этапе обрабатываются директивы препроцессора, такие как "#include", #define", #ifdef и другие.
* Производится подстановка содержимого файлов, указанных в директивах #include".
* Выполняются макроопределения с помощью "#define".

1. **Компиляция('Compilation'):**

* На этом этапе преобразуется код на С++ в язык ассемблера или в

промежуточный код (в зависимости от используемого компилятора).

* Происходит проверка синтаксиса и создание объектного файла (\*\*.obj",0").

1. **Aссемблирование ( Assembly ):**

* Ассемблер преобразует код на языке ассемблера в машинный код.
* Создается объектный файл, который содержит исполняемый код, но еще не содержит информации о размещении в памяти.

**4. Линковка ('Linking'):**

* На этом этапе объединяются все объектные файлы программы, а также используемые библиотеки.
* Разрешаются внешние ссылки (ссылки на функции или переменные, которые определены в других файлах).
* Формируется исполняемый файл, который может быть запущен.

**5.Оптимизация ('Optimization'):**

* Некоторые компиляторы предоставляют этот этап, в ходе которого проводятся оптимизации кода для улучшения его производительности.
* Это может включать в себя удаление недостижимого кода, улучшение последовательности инструкций и другие оптимизации.

**Тема №3: Заголовочные файлы, директивы препроцессора.**

Заголовочные файлы и директивы препроцессора используются для организации и структурирования кода, а также для подключения библиотек и определения макросов.

**Заголовочные файлы**

**Использование:**

* Заг. Файлы содержат объявление функций, констант, классов и других элементов, которые используются в программе.
* Обычно имеют расширение h или hpp.

**Структура заг. Файла:**

* В заг. Файле обычно размещают объявления, а определения помещают в соответствующие исходные файлы (cpp).
* Пример заг. Файла: example.h

**Директивы препроцессора:**

1. #include - Используется для включения содержимого других файлов в текущий.
2. #define – для определения макросов
3. #ifdef, #ifndef, #endif – для условной компиляции. Проверяют были ли уже определены макросы.
4. #pragma – для предоставления доп. Указаний компилятору.
5. #undef – для отмены определения макроса.

**Тема №4: Типы данных и роль их в программировании. Классификация типов данных.**

Типы данных играют важную роль в программировании. Они определяют, какие виды данных могут хранится в переменных и как эти данные могут быть обработаны. Типы данных так же позволяет компилятору понять, сколько байт выделить для каждой переменной, как правильно интерпретировать биты в этих байтах и как выполнить операции с этими данными.

Целочисленные: int (целое число), short (короткое целое число), long (длинное целое число), char (символ 8 бит).

Вещественные: float (одинарная точность), double (двойная точность), long double (расширенная точность).

Символьные: char (символ, как правило 8 бит).

Логические: bool (логическое значение true/false).

Перечислимые типы данных: enum (перечисление).

Составные: массивы (array – последовательность элементов одного типа), структуры (struct – объединяет несколько переменных разных типов в одну структуру), классы (class – основа для объектно-ориентированного программирование в С++).

Указатели (pointer): переменные, содержащие адреса памяти.

Пользовательские типы данных: ключевое слово - typedef или using.

Возвращаемые типы функций: которые возвращают функцию.

Нулевой тип (void): для указания на отсутcтвие типы, для функций, которые

ничего не возвращают например.

**Тема №5: Операции с простыми типами данных.**

C++ поддерживает различные простые типы данных, такие как:

Целочисленные типы данных: int, char, short, long.  
Вещественные типы данных: float, double.

Символьный тип данных: char.

Логический тип данных: bool.

Арифметические операции: a + b; a – b; a \* b; a / b;

Остаток от деления: a % b;

Инкремент и декремент: a++; b--;

Сравнение: (x == y); (x != y); (x > y); (x < y);

Логические операции: bool1 && bool2(и); bool1 | | bool2 (или); !bool1(не)

**Тема №6: ввод/вывод данных**

В ЯП с++ взаимодействие с консолью осуществляется с помощью потоков std::cin и std::cout, хранящихся в библиотеке <iostream>.

cin:

.get() - считывание одного символа из консоли

.getline() - считывание строки из консоли, до заданного символа (символ конца строки, по умолчанию) или первые n символов.

.fail() - флаг ошибки ввода. по умолчанию - false, в случае ошибки - true

.clear() - сброс флага ошибки на false

.ignore() - стирает содержимое буфера ввода (используется в случае ошибки ввода)

cout:

.endl() - очищает буфер потока и осуществляет "перевод каретки" на следующую строку.

.fill() - устанавливает символ заполнитель.

.width() - задает ширину поля

.setf(ios::flagname1 | ios::fligename2) - установка флагов вывода

.unsetf() - снятие флагов вывода

популярные флаги вывода:

boolalpha - вывод булевой переменной в виде (true/false)

right - выравнивание по правой границе

left - выравнивание по левой границе

особенности вывода массивов:

чтобы вывести массив нужно использовать цикл. Для матрицы или массивов большей вложенности потребуется использование вложенных циклов.

исключение:

строки cstring выводятся полностью за один вывод.

особенности вывода структур:

невозможно вывести структуру полностью одной командой. Для вывода структуры предпочтительно написать отдельную функцию, выводящую все ее поля.

**Тема №8,10: Массивы двумерные**

**1) статические массивы - матрицы**

синтаксис:

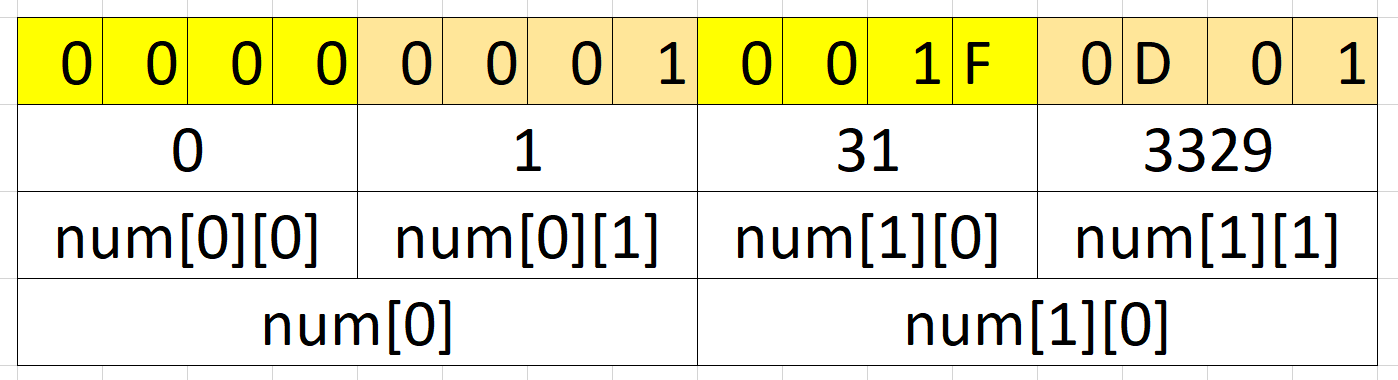
тип\_массива имя\_массива[количество\_строк][количество\_столбцов]

статический массив в c++ может представлять из себя исключительно матрицу прямоугольной формы (все строки равны по длинне).

в памяти статический двумерный массив

int num[2][2]{{0,1},{31,3329}};

представлен сделующим образом:



элементы каждой строки записываются последовательно. Таким образом, пользуясь арифметикой указателей можно получить доступ к любому элементу любой строки массива.

2) динамический массив

синтаксис:

тип\_ссылки\_на\_ссылку \*\* имя\_массива = new тип\_массива \* [количество\_строк]{};

for (int i{}; i < количество\_строк; i++){

имя\_массива[i] = new тип\_массива[количество\_элементов\_строки];

}

в памяти массив

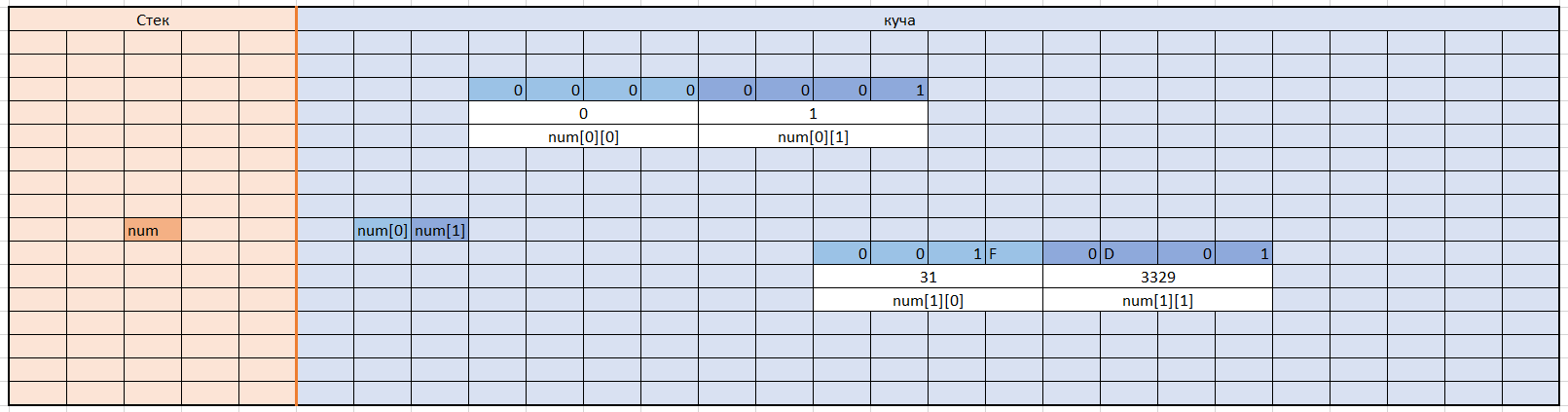
int\*\* num = new int\* [2]{};

for (int i{}; i < 2; i++){

num[i] = new int[2];

}

представлен следующим образом:

****

**В стеке создается переменная (num), в которой хранится адрес массива, содержащего адреса строк двумерного массива. (просто посмотрите на картинку, так понятнее)**

**как можно заметить, применение динамического массива облегчает работу, тк позволяет производить логическую замену элементов (вместо перезаписывания строки поэлементно можно перезаписать указатель)**

**Тема №11: Указатели и ссылки. Примеры.**

**Указатель — это обычная переменная, которая хранит адрес какого - то объекта. Указатель, как и любая другая переменная имеет свой адрес в памяти.**

// создание нулевого указателя

int \*ptr\_1{};

int \*ptr\_2 = nullptr;

**Для создания нескольких указателей, перед каждым именем нужно ставить оператор звëздочки.**

// создание нескольких указателей

int \*ptr\_1{}, \*ptr\_2{}, \*ptr\_3{};

// содание одного указателя

int \*ptr\_4{}, var\_1{}, var\_2{}

**Отличие const int \* от int \* const**

|  |  |
| --- | --- |
| const int \*ptr\_1 = &my\_var\_1; | int \* const ptr\_1 = &my\_var\_1; |
| Чтобы создать указатель, хранящий адрес константы, нужно чтобы этот указатель имел тип указателя на константу | Если мы не хотим менять значение указателя в течении выполнения программы, нужно создать константный указатель |

**Ссылка - это некий псевдоним для переменной (второе имя), не является типом переменной. Под капотом является константным указателем.**

// создание ссылки

int &link\_1 = my\_var\_1;

\* Данные взяты из «Лекция 11» Пчелинцевой Н. И.

**Передача по указателю и передача по ссылке**

В следующей таблице перечислены основные различия между методами передачи по указателю и передачи по ссылке.

| **Параметры** | **Передача по указателю** | **Передача по ссылке** |
| --- | --- | --- |
| **Передача аргументов** | Мы передаем адрес аргументов в вызове функции. | Мы передаем аргументы в вызове функции. |
| **Доступ к значениям** | Доступ к значению аргументов осуществляется через оператор разыменования **\*** | Ссылочное имя можно использовать для неявной ссылки на значение. |
| **Переназначения** | Переданные параметры могут быть перемещены/переназначены в другое место в памяти. | Параметры не могут быть перемещены/переназначены на другой адрес памяти. |
| **Допустимые значения** | Указатели могут содержать значение NULL, поэтому переданный аргумент может указывать на значение NULL или даже на мусорное значение. | Ссылки не могут содержать значение NULL, поэтому оно гарантированно имеет какое-то значение. |

**Разница между ссылочной переменной и переменной-указателем**

Ссылка — это тот же объект, только с другим именем, и ссылка должна ссылаться на объект. Поскольку ссылки не могут иметь значение NULL, они более безопасны в использовании.

* Указатель может быть переназначен, а ссылка — нет, и должен быть назначен только при инициализации.
* Указателю может быть присвоен NULL напрямую, в то время как ссылке — нет.
* Указатели могут перебирать массив, мы можем использовать операторы инкремента/декремента для перехода к следующему/предыдущему элементу, на который указывает указатель.
* Указатель — это переменная, содержащая адрес памяти. Ссылка имеет тот же адрес памяти, что и элемент, на который она ссылается.
* Указатель на класс/структуру использует '->' (оператор стрелки) для доступа к своим членам, в то время как ссылка использует '.' (оператор точки)
* Указатель должен быть разыменован с помощью \*, чтобы получить доступ к области памяти, на которую он указывает, в то время как ссылка может быть использована напрямую.

\* Данные взяты с сайта [Передача по указателю и передача по ссылке в C++ - GeeksforGeeks](https://www.geeksforgeeks.org/passing-by-pointer-vs-passing-by-reference-in-cpp/?ref=lbp)

**Тема №12: Указатели и ссылки. Арифметика указателей. Примеры.**

Арифметика указателей означает выполнение арифметических операций над указателями. Это относится к операциям, которые допустимо выполнять с указателями.

**Ниже приведены арифметические операции, допустимые для указателей на C ++:**

1. Увеличивающие и уменьшающие указатели
2. Добавление константы к указателям
3. Вычитание константы из указателей
4. Вычитание двух указателей одного типа
5. Сравнение указателей

1. Увеличение и уменьшение указателей

Увеличение или уменьшение указателя приведет к тому, что он будет ссылаться на адрес следующих или предыдущих данных в памяти. Этот процесс отличается от увеличения или уменьшения числовых данных.

**Увеличение указателя**

Увеличение указателя будет зависеть от типа адреса переменной, хранящейся в указателе. Если указатель хранит адрес переменной целочисленного типа, то размер целочисленного указателя может составлять 4 или 8 байт в зависимости от 32-разрядной или 64-разрядной машины соответственно. Итак, теперь, если мы увеличим переменную целочисленного типа, она увеличится на 4 или 8 байт в зависимости от ее размера.

### Уменьшение указателя

Когда мы применяем операцию уменьшения к указателю, указатель уменьшается на 4 или 8 байт в зависимости от машины.

int main()

{

    int num = 27;

    // Storing address of num in num\_pointer

    int\* num\_pointer = #

    // Print size of int

    cout << "Size of int: " << sizeof(int) << endl;

    // Print the address stored at num\_pointer

    cout << "Before Increment: " << num\_pointer << endl;

    // Increment pointer

    num\_pointer++;

    cout << "After Increment: " << num\_pointer << endl;

    // Print the address stored at num\_pointer

    cout << "Before Decrement: " << num\_pointer << endl;

    // Decrement pointer

    num\_pointer--;

    cout << "After Decrement: " << num\_pointer << endl;

    return 0;

}

Вывод:

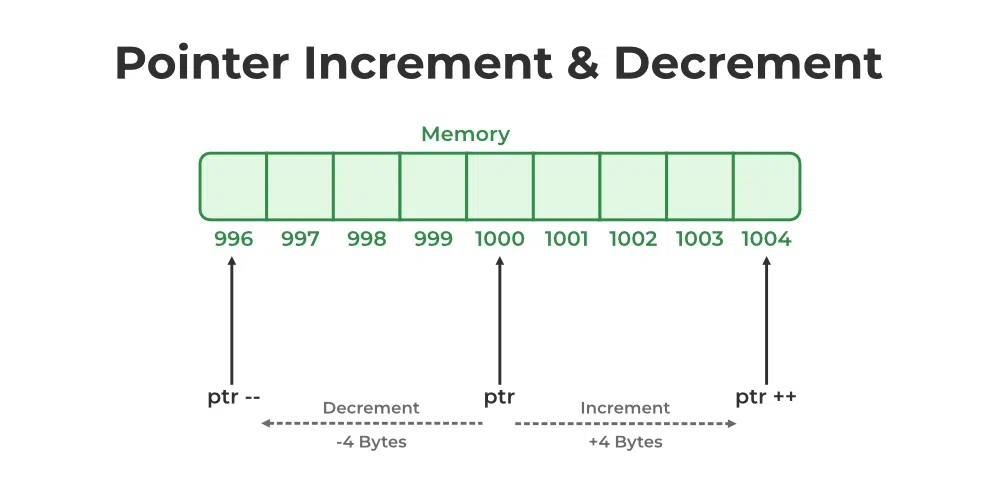
Size of int: 4

Before Increment: 0x7ffe3e7f56d4

After Increment: 0x7ffe3e7f56d8

Before Decrement: 0x7ffe3e7f56d8

After Decrement: 0x7ffe3e7f56d4



## 2. Добавление константы к указателям

Мы можем добавлять целочисленные значения к указателям, и указатель корректируется в зависимости от размера типа данных, на который он указывает. Например, если целочисленный указатель хранит адрес 1000, и мы добавляем значение 5 к указателю, он сохранит новый адрес как:

1000 + (5 \* 4(size of an integer)) = 1020

int main()

{

    int num = 20;

    int\* ptr = #

    cout << "Address stored in ptr: " << ptr << endl;

    // Adding the integer value 1 to the pointer ptr

    ptr = ptr + 1;

    cout << "Adding 1 to ptr: " << ptr << endl;

    // Adding the integer value 2 to the pointer ptr

    ptr = ptr + 2;

    cout << "Adding 2 to ptr: " << ptr << endl;

    return 0;

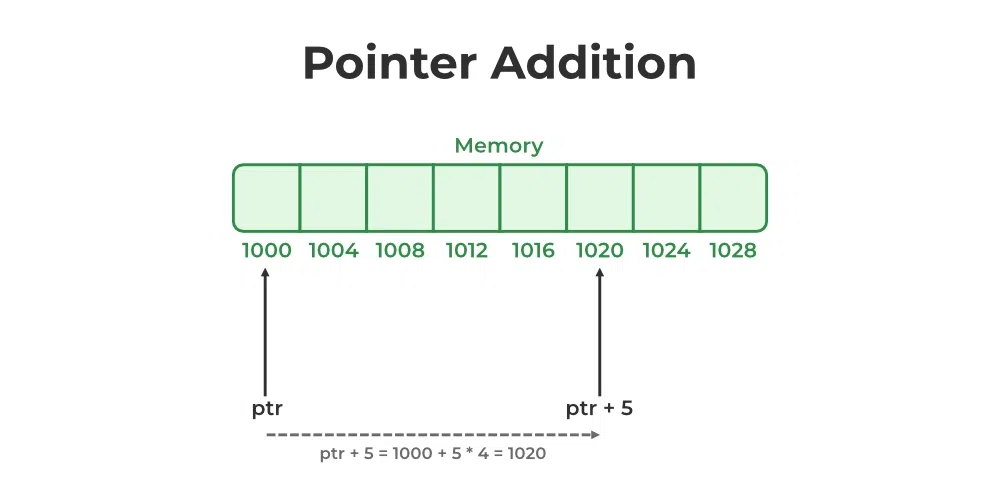
}

Вывод:

Address stored in ptr: 0x7ffdb8634a94

Adding 1 to ptr: 0x7ffdb8634a98

Adding 2 to ptr: 0x7ffdb8634aa0



## 3. Вычитание константы из указателей

Мы также можем вычитать константу из указателей, и это то же самое, что добавление константы к указателю. Например, если целочисленный указатель хранит адрес 1000, и мы вычитаем значение 5 из указателя, он сохранит новый адрес в виде:

1000 - (5 \* 4(size of an integer)) = 980

int main()

{

    int num = 100;

    int\* ptr = #

    cout << "Address stored in ptr: " << ptr << endl;

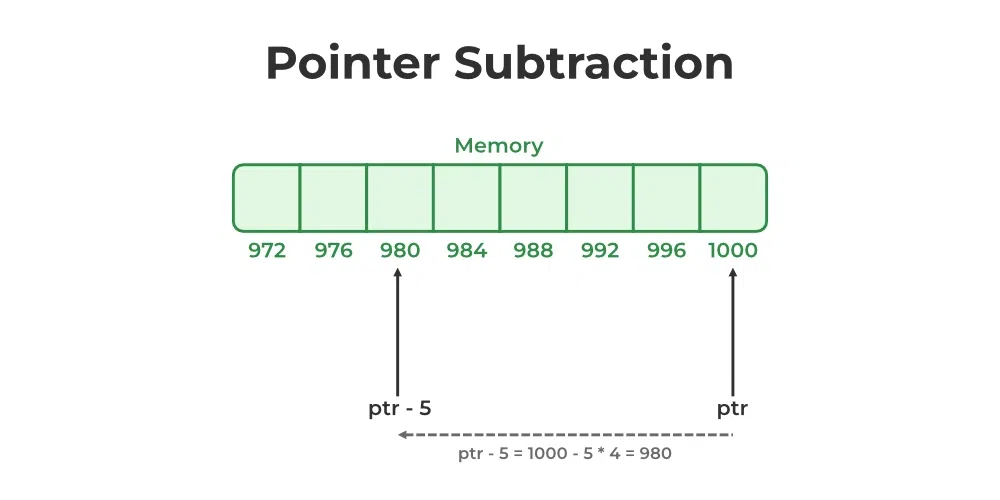
    // Subtracting the integer value 1 from pointer ptr

    ptr = ptr - 1;

    cout << "Subtract 1 from ptr: " << ptr << endl;

    return 0;

}



## 4. Вычитание двух указателей одного и того же типа данных

Вычитание двух указателей может быть выполнено только тогда, когда оба указателя имеют один и тот же тип данных. Вычитание двух указателей дает количество элементов, присутствующих между двумя указателями.

int main()

{

    int num = 45;

    int\* ptr1 = #

    // Adding 4 to ptr1 and stored in ptr2

    int\* ptr2 = ptr1 + 4;

    cout << "Address stored in ptr1:" << ptr1 << endl;

    cout << "Address stored in ptr2:" << ptr2 << endl;

    // Subtracting ptr2 from ptr1

    cout << "ptr2 - ptr1 = " << ptr2 - ptr1 << endl;

    return 0;

}

Вывод:

Address stored in ptr1:0x7ffdd3f5673c

Address stored in ptr2:0x7ffdd3f5674c

ptr2 - ptr1 = 4

## 5. Сравнение указателей

В C ++ мы можем выполнить сравнение между двумя указателями, используя операторы отношения ( **>, <, >=, <=, ==, !=**). Обычно мы используем эту операцию, чтобы проверить, указывают ли два указателя as на одну и ту же ячейку памяти или нет.

int main()

{

    // declaring some pointers

    int num = 10;

    int\* ptr1 = #

    int\*\* ptr2 = &ptr1;

    int\* ptr3 = \*ptr2;

    // comparing equality

    if (ptr1 == ptr3) {

        cout << "Both point to same memory location";

    }

    else {

        cout << "ptr1 points to: " << ptr1 << endl;

        cout << "ptr3 points to: " << ptr3 << endl;

    }

    return 0;

}

Вывод:

Both point to same memory location

### Сравнение с NULL

Мы можем сравнить указатель типа с NULL. Эта операция помогает нам определить, указывает ли данный указатель на некоторый адрес памяти или нет. Это помогает нам контролировать ошибки, такие как ошибки сегментации.

int main()

{

    int num = 10;

    // assigning null in case we dont use pointer

    int\* ptr = NULL;

    ptr = #

    // checking if the pointer is in use or not

    if (ptr == NULL) {

        cout << "No value is pointed";

    }

    else {

        cout << "The value pointed is " << \*ptr;

    }

    return 0;

}

Вывод:

The value pointed is 10

\*Данные взяты с сайта https://www.geeksforgeeks.org/cpp-pointer-arithmetic/

**Тема №13: Обход массивов с помощью указателей.**

**Указатели на массивы**

Имя массива всегда хранит адрес самого первого элемента. И нередко для перемещения по элементам массива используются отдельные указатели:

|  |
| --- |
| int nums[] {1, 2, 3, 4, 5};  int \*ptr {nums};  int num3 = \*(ptr+2);  std::cout <<  "num3: " << num3 << std::endl;  // num3: 3 |

Здесь указатель ptr изначально указывает на первый элемент массива. Увеличив указатель на 2, мы пропустим 2 элемента в массиве и перейдем к элементу nums[2].

Можно сразу присвоить указателю адрес конкретного элемента массива:

|  |
| --- |
| int nums[] {1, 2, 3, 4, 5};  int \*ptr {&nums[2]};    // адрес третьего элемента  std::cout << "\*ptr = " << \*ptr  << std::endl; //\*ptr = 3 |

С помощью указателей легко перебрать массив:

|  |
| --- |
| #include <iostream>    int main()  {      const int n = 5;      int nums[n]{1, 2, 3, 4, 5};        for(int \*ptr{nums}; ptr<=&nums[n-1]; ptr++)      {          std::cout << "address=" << ptr << "\tvalue=" << \*ptr << std::endl;      }  } |

Так как указатель хранит адрес, то мы можем продолжать цикл, пока адрес в указателе не станет равным адресу последнего элемента.

Аналогичным образом можно перебрать и многомерный массив:

|  |
| --- |
| #include <iostream>    int main()  {      int nums[3][4] { {1, 2, 3, 4} , {5, 6, 7, 8}, {9, 10, 11, 12}};      unsigned int n { sizeof(nums)/sizeof(nums[0]) };         // число строк      unsigned int m { sizeof(nums[0])/sizeof(nums[0][0]) };   // число столбцов       // указатель на самый последний элемент 0 + 3 \* 4 - 1 = 11      int \*end {nums[0] + n \* m - 1};        int \*ptr {nums[0]};                // указатель на первый элемент      for( unsigned i{1}; ptr <= end; ptr++, i++)      {          std::cout << \*ptr << "\t";          // если остаток от целочисленного деления равен 0,          // переходим на новую строку          if(i%m == 0)          {              std::cout << std::endl;          }      }  } |

Поскольку в данном случае мы имеем дело с двухмерным массивом, то адресом первого элемента будет выражение a[0]. Соответственно указатель указывает на этот элемент. С каждой итерацией указатель увеличивается на единицу, пока его значение не станет равным адресу последнего элемента, который хранится в указателе end.

\*Данные взяты с сайта [C++ | Указатели и массивы (metanit.com)](https://metanit.com/cpp/tutorial/4.5.php?ysclid=lr20omqpdy407682824)

**Тема №14: Динамически одномерные массивы. Создание, удаление и обработка.**

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

### Как работают динамические массивы

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

## Как создать динамический массив в C++

Чтобы **создать динамический массив** мы будем использовать конструкцию ниже:

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

* <тип данных> - без разницы какой тип данных тут будет находиться, но лучше тот, который будет совпадать с типом переменных.
* <тип переменных> - указанный сюда тип и будут иметь ячейки массива.
* <количество ячеек> - здесь мы задаем размер массива (например [n] или [25]).

Динамический массив полностью идентичен обычному массиву, кроме:

* Своей инициализации
* Возможностью своевременно освободить память.

Давайте рассмотрим пример с использованием динамического массива:

int main() {

setlocale(0, "");

int n;

cout << "Введите количество чисел, которое вы хотите ввести: ";

cin >> n;

cout << "Введите " << n << " чисел: ";

int \*dinamich\_array = new int [n]; // создаем

// динамический массив

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

cin >> dinamich\_array[i]; // считываем числа в ячейки массива

}

cout << "Теперь давайте выведем элементы массива в обратном порядке: ";

for (int i = n - 1 ; i >= 0; i--) {

cout << dinamich\_array[i] << " "; // выводим значение всех ячеек

}

cout << endl << "Удаляем массив!";

delete [] dinamich\_array; // удаляем динамический массив

return 0;

}

Вот что будет при выполнении программы:

**Задайте количество чисел, которое вы хотите ввести: 5**

**Введите 5 чисел: 2 4 6 8 16**

**Теперь давайте выведем элементы массива в обратном порядке: 16 8 6 4 2**

**Удаляем массив!**

**Process returned 0 (0x0) execution time : 0.010 s**

**Press any key to continue.**

## Удаление динамического массива

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

delete [] <имя массива>;

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

\*Данные взяты с сайта [Динамические массивы и переменные в C++: легко и просто! (codelessons.dev)](https://codelessons.dev/ru/dinamicheskie-massivy-i-peremennye-vse-samoe-glavnoe/)

**Тема №16: Строки (с-строки). Функции для работы со строками.**

**Символ** – элементарная единица, некоторый набор которых несет определенный смысл. В языке программирования С++ предусмотрено использование символьных констант. Символьная константа – это целочисленное значение (типа int) представленное в виде символа, заключённого в одинарные кавычки, например 'a'.

Вес одного символа – 1 байт или 8 бит.

Строки в С++ представляются как массивы элементов типа char, заканчивающиеся нуль-терминатором \0 называются С-строками или строками в стиле С.

Объявление строки:

char str[10];

str – имя строки

10 – размер массива, то есть в данной строке может поместиться 9 символов, последнее место отводится под нуль-терминатор.

Инициализация строки:

char string[10] = "abcdefghf";

char string[10] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'f', '\0' };

char string[] = "abcdefghf";

Также можно ввести строку с клавиатуры при помощи инструкции cin, которая считывает только одно слово. Еще можно использовать функцию getline().

Объявление массивов строк в языке С также возможно. Для этого используются двумерные массивы символов, что имеет следующий синтаксис:

char имя[количество][длина];

Первым размером матрицы указывается количество строк в массиве, а вторым – максимальная (с учетом завершающего нуля) длина каждой строки При объявлении массивов строк можно производить инициализацию:

char имя[количество][длина] = {стр. литерал №1, ..., cтр. литерал №N};

Функции C-style строк:

cin.getline(string, streamsize, separator); – считывание строки

где string – переменная типа char, в которую запишется строка, streamsize - максимально количество символов, которое может быть записано в строку, и separator – строковый разделитель, показывающий на конец строки.

strcpy(str1, str2)

Выполняется побайтное копирование символов из строки str2 в строку str1. Копирование прекращается только в случае достижения символа '\0' (нуль- терминатор). Перед копированием необходимо проверить выполнение условия, что длина str2 меньше или равна длине str1. В противном случае возможно возникновение ошибок, связанных с наложением данных.

strlen(s)

Функция strlen вычисляет количество символов в строке до первого вхождения символа конца строки. При этом символ конца строки не входит в подсчитанное количество символов.

strcat(s1, s2)

Данная функция объединяет s2 и s1

strcmp(str1, str2)

Функция имеет тип int, так как данная функция выполняет сравнение двух строк str1 и str2, различая прописные и строчные буквы, в результате сравнения возвращает одно из следующих значений:

Если результат функции < 0, то str1 < str2.

Если результат функции = 0, то строки эквивалентны

Если результат функции > 0, то str1 > str2

strchr(s, ‘symbol’)

Производит поиск символа в строке, указывает место первого вхождения символа в строку. Если символ не найден, функция возвращает NULL.

strrchr() – возвращает указатель на последний, совпадающий с заданным символом, символ в строке. Если символ не найден, возвращается значение NULL.

strlwr(s) – преобразует прописные буквы в строчные

strupr(s) – преобразует строчные буквы в прописные

strrev(s) – реверсирование строки, т.е. изменяет порядок следования символов на обратный.

stricmp(s) – тоже самое, что и strcmp() только не берет в учет регистр.

strcspn(str1,  str2 )

Функция strcspn выполняет поиск первого вхождения в строку string1 любого из символов строки string2, и возвращает количество символов до найденного первого вхождения.

Поиск учитывает и завершающий нуль-символов, поэтому, если функция возвращает длину строки string1, это значит, что  ни один из символов строки string2 не входит в состав string1.

strtok(string, delim)

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

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

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

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

**Тема №17: Структуры. Обработка полей структуры.**

**Структура** – это пользовательский тип данных, который описывает какую-либо сущность

Структуры помогают в организации сложных данных (особенно в больших программах), поскольку позволяют группу связанных между собой переменных трактовать не как множество отдельных элементов, а как единое целое.

Для определения структуры применяется ключевое слово struct.

Как и в случае с обычными переменными, определение переменной типа структуры выделяет память для этой переменной.

Создание структуры:

struct Employee {

int id;

int age;

double wage;

};

Создание переменной с типом структуры:

Employee joe{};

Обработка полей структуры:

joe.id = 14;

joe.age = 32;

joe.wage = 24.15;

Инициализация:

Employee joe{ 1, 32, 60000.0 };

**Тема №18: Обработка полей структуры с помощью указателя.**

Создадим переменную с типом структуры Employee:

Employee kai{};

kai.id = 9;

kai.age = 19;

kai.wage = 53;

А теперь создадим указатель на переменную kai:

Employee \*kai2 = &kai;

Чтобы изменить какое-либо из полей переменной kai с помощью указателя kai2, вместо точки нужно будет использовать стрелку (->)

kai2->age = 20;

kai2->wage = 54;

**Тема №19: Перечисления. Объединения**

**Перечисления.**

Перечислением называется тип данных, который включает множество именованных целочисленных констант. Именованные константы, принадлежащие перечислению, называются перечислимыми константами.

Объявляются перечисления следующим образом:

enum  color  { r, g, b };

где enum – ключевое слово, color – имя типа перечисления, r, g, b – сами перечислимые константы. При объявлении типа перечисления его значения могут инициализироваться произвольными целочисленными константами или константным выражением. Например,

enum  color  { r = 2, g = r = 2, b = 6 };

Если инициализация отсутствует, то перечислимым константам присваиваются последовательные значения: 0, 1, 2, … Например,

enum  color  { r, g, b }; r = 0, g = 1, b = 2

**Объединения.**

Тип объединение описывает набор данных, которые называются элементами или членами объединения. Объединение позволяет хранить в одной и той же области памяти значения различных типов, которые соответствуют типам элементов объединения. Но в каждый данный момент времени в этой области памяти может храниться только одно значение. Отсюда следует, что длина памяти, распределяемой компилятором под объединение, равна наибольшей из длин членов этого объединения. Как и в случае структуры, члены объединения могут иметь любой тип за исключением типов void и функция.

Объявляются объединения следующим образом:

union  num

{

int  n;

double  f;

};

где union – это ключевое слово, num – имя типа объединения, n, f – члены объединения.

**Тема №20: Виды функций. Механизмы передачи значений в функции. Возвращаемые значения из функций. Вызов функций.**

Функция – это блок кода, который выполняет определенную задачу.  
Разделение сложной проблемы на более мелкие части делает нашу программу простой для понимания и повторного использования. Есть два типа функций:

1) Стандартные библиотечные функции: предопределены в С++.

2) Определяемая пользователем функция (создается пользователями).

Аргументы, которые представляют переменные или константы, могут передаваться в функцию по значению (by value) и по ссылке (by reference).

При передаче аргументов по значению функция получает копию значения переменных и констант. Например:

void square(int m)

{

    m = m \* m;

    cout << "In square: m = " << m << endl;

}

При передаче параметров по ссылке передается ссылка на переменную, через которую мы можем манипулировать самой переменной, а не просто ее значением. Так, перепишем предыдущий пример, используя передачу по ссылке:

void square(int& m)

{

    m = m \* m;

    cout << "In square: m = " << m << endl;

}

Функция возвращает значение с типом, который мы определяем перед именем функции. Например:

void f1(){

cout << “Hi” << endl;

return;

};

int f2(){

return 1;

};

double f3(){

return 3.2;

};

Вызов функции:

int main(){

f1();

cout << f2() + f3();

}

Результат работы программы:

Hi

4.2

**Тема №1: Рекурсивная функция.**

**Рекурсивная функция** — это функция, которая вызывает саму себя.

Рекурсия функции нужна, когда требуется выполнить последовательность из одинаковых действий.

**Тема №2: Стек вызовов.**

**Стек вызовов** — это сегмент данных, используемый для хранения локальных переменных и временных значений.

У стека в C++ можно обратиться к произвольной ячейке. Стек выделяется при запуске программы и обычно небольшой по размеру (4Мб). Все функции хранят свои локальные переменные на стеке. При выходе из функции соответствующая область стека объявляется свободной. Промежуточные значения, возникающие при вычислении сложных выражений, также хранятся на стеке.

**Тема №1: Файлы. Общий подход к работе с файлами.**

**Файл** – **именованный набор байтов, который может быть сохранен на некотором накопителе.** под файлом понимается некоторая последовательность байтов, которая имеет своё, уникальное имя, например **Input.txt**

В одной директории не могут находиться файлы с одинаковыми именами. Под именем файла понимается не только его название, но и расширение, например: **Input.txt** и **Input.bin** **—**разные файлы, хоть и имеют одинаковые названия.

Полное имя файлов – это полный адрес к директории файла с указанием имени файла, например: **D:\\Docs\\Input.txt**

**Для работы с файлами необходимо подключить заголовочный файл <fstream>.**В **<fstream>** определены несколько классов и подключены заголовочные файлы:

**<ifstream> —**файловый ввод

**<ofstream> —**файловый вывод.

Например, необходимо создать текстовый файл и записать в него строку необходимо проделать следующие шаги:

1. создать объект класса **ofstream;**
2. связать объект класса с файлом, в который будет производиться запись;
3. записать строку в файл;
4. закрыть файл.

Функция **open(Имя файла, режим)** – открывает файл для работы**(!!Обязательно всегда нужно после взвимодействий с файлом закрыть его функцией close(Имя файла))**

Для открытия файла в функцию необходимо передать путь к файлу. И также можно указать режим открытия. Список доступных режимов открытия файла:

* **ios::in**: файл открывается для ввода (чтения). Может быть установлен только для объекта ifstream или fstream
* **ios::out**: файл открывается для вывода (записи). При этом старые данные удаляются. Может быть установлен только для объекта ofstream или fstream
* **ios::app**: файл открывается для дозаписи. Старые данные не удаляются.
* **ios::ate**: после открытия файла перемещает указатель в конец файла
* **ios::trunc**: файл усекается при открытии. Может быть установлен, если также установлен режим out
* **ios::binary**: файл открывается в бинарном режиме