Конечно, вот ответы на все вопросы из списка:

1. История возникновения языка программирования C++. Версии стандарта языка. IDE для программирования на C++. Средства отладки приложений в IDE.

   Ответ: Язык программирования C++ был разработан Бьярне Страуструпом в 1983 году как расширение языка C. Основная цель создания C++ заключалась в добавлении объектно-ориентированного программирования к C, сохранив при этом его эффективность. Первым стандартом языка стал C++98, утвержденный в 1998 году. Далее последовали C++03, C++11 (который внес значительные изменения и улучшения), C++14, C++17 и последний стандарт на момент 2023 года — C++20. Популярные IDE для C++ включают Visual Studio, Code::Blocks, CLion и Eclipse CDT. Эти IDE предоставляют инструменты для написания, компиляции и отладки программ. Средства отладки в IDE обычно включают точки останова (breakpoints), просмотр переменных в реальном времени, трассировку выполнения программы и профилирование производительности.

2. Переменные и константы, их объявление и определение. Правила задания идентификаторов. Базовые типы данных, спецификаторы типов. Суффиксы для литералов.

   Ответ: Переменные в C++ объявляются следующим образом: тип имя_переменной;, например, int x;. Константы объявляются с использованием ключевого слова const: const int y = 10;. Идентификаторы должны начинаться с буквы или подчеркивания и могут содержать буквы, цифры и подчеркивания. Например, myVar, _temp, value1. Базовые типы данных включают: int, float, double, char, bool. Спецификаторы типов (модификаторы) могут быть: signed, unsigned, short, long. Суффиксы для литералов используются для указания типа литерала: u или U для unsigned, l или L для long, ll или LL для long long, f или F для float.

3. Функции преобразования типа. Явные и неявные, безопасные и небезопасные неявные преобразования. Функция static_cast. Оператор приведения типа в стиле C.

   Ответ: Преобразование типов в C++ бывает явным и неявным. Явное преобразование выполняется программистом с использованием операторов приведения, например, static_cast<int>(x). Неявное преобразование выполняется компилятором автоматически, когда это возможно. Безопасные преобразования не приводят к потере данных или изменению значения, например, преобразование int к float. Небезопасные преобразования могут приводить к потере данных или изменению значения, например, преобразование float к int. static_cast используется для явного преобразования одного типа в другой, когда преобразование логически допустимо, например, int к float. Пример использования: int

x = static_cast<int>(3.14);. Оператор приведения в стиле C выглядит как (type)value, например, (int)3.14.

4. Математические функции и функции округления, тригонометрические функции. Принцип хранения действительных чисел в памяти компьютера. Математические функции языка C для параметров с типами int, float и double.

   Ответ: В C++ доступны различные математические функции, такие как sqrt (квадратный корень), pow (возведение в степень), abs (модуль числа). Для округления используются функции ceil (округление вверх), floor (округление вниз), round (обычное округление). Тригонометрические функции включают sin, cos, tan. Действительные числа хранятся в формате с плавающей точкой, который делится на три части: знак, экспонента и мантисса. Математические функции в C++ поддерживают параметры типов int, float, double, например, int x = abs(-5); float y = sqrt(25.0f); double z = pow(2.0, 3.0);.

5. Основные арифметические операции, оператор присваивания, цепочка присваиваний, модификации оператора присваивания, префиксные и постфиксные инкремент и декремент. Приоритет выполнения операций. Комментарии в C++.

   Ответ: Основные арифметические операции включают: сложение (+), вычитание (-), умножение (*), деление (/), остаток от деления (%). Оператор присваивания = используется для присвоения значения переменной, например, int x = 5;. Цепочка присваиваний позволяет присваивать одно и то же значение нескольким переменным: int a, b, c; a = b = c = 10;. Модификации оператора присваивания включают +=, -=, *=, /=, %=. Префиксные (++x, --x) и постфиксные (x++, x--) операторы инкремента и декремента увеличивают или уменьшают значение переменной на единицу. Приоритет выполнения операций определяется правилами языка: сначала выполняются операции в скобках, затем инкремент и декремент, умножение и деление, и наконец сложение и вычитание. Комментарии в C++ могут быть однострочными (//) и многострочными (/* ... */).

6. Условные операторы и оператор множественного выбора, использование перечисляемого типа в switch. Операторы отношения, логические операции, пустой оператор, составной оператор, его влияние на область видимости переменных и функций.

   Ответ: Условные операторы включают if, else if, else. Пример: if (x > 0) { /* код */ } else { /* код */ }. Оператор множественного выбора switch используется для выбора одного из нескольких вариантов. Пример:

```
enum Color { RED, GREEN, BLUE };
Color c = RED;
switch (c) {
```

```
    case RED: /* код */ break;
    case GREEN: /* код */ break;
    case BLUE: /* код */ break;
}
```

Операторы отношения включают ==, !=, >, <, >=, <=. Логические операции: && (логическое И), || (логическое ИЛИ), ! (логическое НЕ). Пустой оператор ; используется для завершения выражений. Составной оператор { ... } группирует несколько операторов в блок, влияя на область видимости переменных и функций.

7. Вложенные конструкции if-else, использование операторов присваивание и «запятая» в условии оператора if. Побитовые логические операции и модификации оператора присваивания для работы с ними, тернарный оператор условия.

   Ответ: Вложенные конструкции if-else позволяют создавать многоуровневую логику принятия решений. Пример:

```
if (x > 0) {
    if (y > 0) {
        // код
    } else {
        // код
    }
} else {
    // код
}
```

   Операторы присваивания и «запятая» в условии if используются для выполнения нескольких операций. Пример:

```
if ((x = func()) != 0) {
    // код
}
```

   Побитовые логические операции включают & (побитовое И), | (побитовое ИЛИ), ^ (побитовое исключающее ИЛИ), ~ (побитовое НЕ), << (сдвиг влево), >> (сдвиг вправо). Модификации оператора присваивания для побитовых операций: &=, |=, ^=, <<=, >>=. Тернарный оператор ?: используется для краткой записи условий. Пример: int y = (x > 0) ? 1 : -1;.

8. Цикл со счетчиком, цикл с предусловием и цикл с постусловием. Операторы досрочного завершения тела цикла. Использование оператора «запятая» в заголовке цикла. Организация циклов при помощи меток и безусловных переходов.

   Ответ: Цикл со счетчиком for:

   “‘cpp for (int i = 0; i < 10; ++i) { // код }

Цикл с предусловием `while`:
```cpp
while (x < 10) {
    // код
}
```

Цикл с постусловием do-while:

```cpp
do {
    // код
} while (x < 10);
```

Операторы досрочного завершения break и continue:

```cpp
for (int i = 0; i < 10; ++i) {
    if (i == 5) break; // завершает цикл
    if (i % 2 == 0) continue; // переходит к следующей итерации
    // код
}
```

Оператор «запятая» в заголовке цикла:

```cpp
for (int i = 0, j = 10; i < j; ++i, --j) {
    // код
}
```

Организация циклов с использованием меток и безусловных переходов:

```cpp
for (int i = 0; i < 10; ++i) {
loop_start:
    if (i == 5) goto loop_end;
    // код
}
loop_end:
// код после цикла
```

9. Вложенные циклы. Использование операторов прерывания во вложенных циклах. Организация вложенных циклов при помощи меток и безусловных переходов.

   Ответ: Вложенные циклы позволяют выполнять многократные повторения внутри других циклов. Пример:

   ```cpp
   for (int i = 0; i < 10; ++i) {
       for (int j = 0; j < 10; ++j) {
           // код
           if (j == 5) break; // прерывание внутреннего цикла
       }
   }
   ```

   Прерывание во вложенных циклах можно осуществлять с помощью меток и операторов goto:

```
for (int i = 0; i < 10; ++i) {
    for (int j = 0; j < 10; ++j) {
        if (j == 5) goto end_loops; // прерывание обоих циклов
        // код
    }
}
end_loops:
// код после циклов
```

10. Алгоритмы нахождения делителей числа, определение простоты числа, вычисления факториала натурального числа, выделения цифр натурального числа.

    Ответ: Нахождение делителей числа:

```
for (int i = 1; i <= n; ++i) {
    if (n % i == 0) {
        // i - делитель n
    }
}
```

    Определение простоты числа:

```
bool isPrime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i <= sqrt(n); ++i) {
        if (n % i == 0) return false;
    }
    return true;
}
```

    Вычисление факториала:

```
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

    Выделение цифр числа:

```
void extractDigits(int n) {
    while (n > 0) {
        int digit = n % 10;
        n /= 10;
        // код для обработки digit
    }
}
```

11. Статические одномерные массивы. Способы инициализации элементов массива. Обращение к содержимому ячейки массива. Оператор и функция sizeof. Ручной ввод-вывод массивов. Заполнение элементов массива случайными числами.

    Ответ: Объявление и инициализация статического одномерного массива:

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

    Обращение к элементам массива:

    ```
    int x = arr[2]; // доступ к третьему элементу
    arr[0] = 10; // изменение первого элемента
    ```

    Оператор и функция sizeof:

    ```
    int size = sizeof(arr) / sizeof(arr[0]); // количество элементов в массиве
    ```

    Ручной ввод массива:

    ```
    for (int i = 0; i < size; ++i) {
        cin >> arr[i];
    }
    ```

    Вывод массива:

    ```
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    ```

    Заполнение случайными числами:

    ```
    #include <cstdlib>
    #include <ctime>
    srand(time(0)); // инициализация генератора случайных чисел
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % 100; // случайное число от 0 до 99
    }
    ```

12. Статические многомерные массивы. Способы инициализации элементов массива. Обращение к содержимому ячейки массива. Оператор и функция sizeof. Ручной ввод-вывод массивов. Заполнение элементов массива случайными числами.

    Ответ: Объявление и инициализация статического многомерного массива:

    ```
    int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
    ```

    Обращение к элементам массива:

    ```
    int x = matrix[1][2]; // доступ к элементу в 2-й строке и 3-м столбце
    matrix[0][0] = 10; // изменение элемента в 1-й строке и 1-м столбце
    ```

Оператор и функция sizeof:

```
int rows = sizeof(matrix) / sizeof(matrix[0]);
int cols = sizeof(matrix[0]) / sizeof(matrix[0][0]);
```

Ручной ввод массива:

```
for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
        cin >> matrix[i][j];
    }
}
```

Вывод массива:

```
for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
        cout << matrix[i][j] << " ";
    }
    cout << endl;
}
```

Заполнение случайными числами:

```
#include <cstdlib>
#include <ctime>
srand(time(0)); // инициализация генератора случайных чисел
for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
        matrix[i][j] = rand() % 100; // случайное число от 0 до 99
    }
}
```

13. Динамические одномерные массивы в языке C++. Понятие указателя, операции над указателями, инициализация указателя. Динамическое распределение памяти для одномерных массивов с использованием операторов new и delete.

    Ответ: Объявление указателя и динамическое распределение памяти:

    ```
    int* arr = new int[10]; // выделение памяти для массива из 10 элементов
    ```

    Обращение к элементам массива через указатель:

    ```
    arr[0] = 1;
    int x = arr[1];
    ```

    Освобождение памяти:

    ```
    delete[] arr;
    ```

14. Динамические одномерные массивы в языке C. Понятие указателя, операции над указателями, инициализация указателя. Динамическое

распределение памяти для одномерных массивов с использованием функций malloc, calloc, realloc, free.

Ответ: Объявление указателя и динамическое распределение памяти:

```
int* arr = (int*)malloc(10 * sizeof(int)); // выделение памяти для массива из 10 элементов
```

Обращение к элементам массива через указатель:

```
arr[0] = 1;
int x = arr[1];
```

Освобождение памяти:

```
free(arr);
```

Использование calloc и realloc:

```
int* arr = (int*)calloc(10, sizeof(int)); // выделение и инициализация памяти
arr = (int*)realloc(arr, 20 * sizeof(int)); // изменение размера массива
free(arr);
```

15. Алгоритмы сортировки элементов массива: пузырьковый, вставкой, выбором.

Ответ: Сортировка пузырьком:

```
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; ++i) {
        for (int j = 0; j < size - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

Сортировка вставкой:

```
void insertionSort(int arr[], int size) {
    for (int i = 1; i < size; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = key;
    }
}
```

Сортировка выбором:

```
void selectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; ++i) {
        int minIndex = i;
        for (int j = i + 1; j < size; ++j) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        std::swap(arr[i], arr[minIndex]);
    }
}
```

16. Алгоритмы поиска минимального (максимального), нахождение суммы (произведения) элементов, сдвиги элементов одномерных массивов

Алгоритм поиска минимального (максимального) элемента Для нахождения минимального или максимального элемента в одномерном массиве используется следующий алгоритм:

1. Инициализировать переменную для хранения минимального (максимального) значения первым элементом массива.
2. Перебрать все элементы массива:
   - Если текущий элемент меньше (больше) сохраненного значения, обновить минимальное (максимальное) значение.
3. После завершения цикла в переменной будет храниться минимальное (максимальное) значение массива.

Пример на языке C++:

```
#include <iostream>
using namespace std;

int main() {
    int arr[] = {4, 2, 8, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    int min = arr[0];
    int max = arr[0];

    for (int i = 1; i < n; i++) {
        if (arr[i] < min)
            min = arr[i];
        if (arr[i] > max)
            max = arr[i];
    }
```

```
    cout << "Minimum: " << min << endl;
    cout << "Maximum: " << max << endl;

    return 0;
}
```

Алгоритм нахождения суммы (произведения) элементов Для нахождения суммы или произведения элементов одномерного массива:

1. Инициализировать переменную для суммы нулем (для произведения единицей).
2. Перебрать все элементы массива:
   - Для суммы: прибавить текущий элемент к переменной суммы.
   - Для произведения: умножить текущий элемент на переменную произведения.
3. После завершения цикла в переменной будет храниться сумма (произведение) всех элементов массива.

Пример на языке C++:

```
#include <iostream>
using namespace std;

int main() {
    int arr[] = {4, 2, 8, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    int sum = 0;
    int product = 1;

    for (int i = 0; i < n; i++) {
        sum += arr[i];
        product *= arr[i];
    }

    cout << "Sum: " << sum << endl;
    cout << "Product: " << product << endl;

    return 0;
}
```

Алгоритм сдвига элементов массива Для сдвига элементов массива влево или вправо:

1. Сдвиг влево:
   - Сохранить первый элемент массива.
   - Переместить каждый элемент массива на одну позицию влево.

- Последний элемент заменить сохраненным первым элементом.

2. Сдвиг вправо:
   - Сохранить последний элемент массива.
   - Переместить каждый элемент массива на одну позицию вправо.
   - Первый элемент заменить сохраненным последним элементом.

Пример на языке C++ (сдвиг влево):

```
#include <iostream>
using namespace std;

void shiftLeft(int arr[], int n) {
    int temp = arr[0];
    for (int i = 0; i < n - 1; i++) {
        arr[i] = arr[i + 1];
    }
    arr[n - 1] = temp;
}

int main() {
    int arr[] = {4, 2, 8, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    shiftLeft(arr, n);

    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

Пример на языке C++ (сдвиг вправо):

```
#include <iostream>
using namespace std;

void shiftRight(int arr[], int n) {
    int temp = arr[n - 1];
    for (int i = n - 1; i > 0; i--) {
        arr[i] = arr[i - 1];
    }
    arr[0] = temp;
}

int main() {
    int arr[] = {4, 2, 8, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    shiftRight(arr, n);

    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

Эти алгоритмы позволяют эффективно выполнять основные операции с одномерными массивами.

17. Особенности описания многомерных динамических массивов в языке C++. Динамическое распределение памяти для многомерных массивов с использованием операторов new и delete

Особенности описания многомерных динамических массивов в C++ В языке C++ многомерные динамические массивы могут быть описаны и выделены с использованием указателей и операторов new и delete. Основной особенностью является необходимость создания массива указателей для каждой измерения массива.

Динамическое распределение памяти для многомерных массивов Для создания двумерного динамического массива сначала создается массив указателей на строки, а затем для каждой строки выделяется память под элементы.

Пример создания и удаления двумерного динамического массива:

```
#include <iostream>
using namespace std;

int main() {
    int rows = 3;
    int cols = 4;

    // Создание массива указателей на строки
    int** array = new int*[rows];

    // Выделение памяти для каждого ряда
    for (int i = 0; i < rows; ++i) {
        array[i] = new int[cols];
    }

    // Инициализация и вывод массива
    for (int i = 0; i < rows; ++i) {
```

```
        for (int j = 0; j < cols; ++j) {
            array[i][j] = i * cols + j;
            cout << array[i][j] << " ";
        }
        cout << endl;
    }

    // Удаление массива
    for (int i = 0; i < rows; ++i) {
        delete[] array[i];
    }
    delete[] array;

    return 0;
}
```

В этом примере сначала выделяется память для массива указателей на строки, затем выделяется память для каждого ряда. После использования массива память освобождается в обратном порядке: сначала освобождаются строки, затем массив указателей.

18. Особенности описания многомерных динамических массивов в языке С. Динамическое распределение памяти для многомерных массивов с использованием функций malloc, calloc, realloc, free

Особенности описания многомерных динамических массивов в С В языке С многомерные динамические массивы описываются и создаются с использованием функций выделения памяти malloc, calloc, realloc и освобождения памяти free.

Динамическое распределение памяти для многомерных массивов Процесс создания двумерного динамического массива аналогичен процессу в C++, но с использованием функций выделения памяти из стандартной библиотеки С.

Пример создания и удаления двумерного динамического массива:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int rows = 3;
    int cols = 4;

    // Создание массива указателей на строки
    int** array = (int**)malloc(rows * sizeof(int*));
```

```
    // Выделение памяти для каждого ряда
    for (int i = 0; i < rows; ++i) {
        array[i] = (int*)malloc(cols * sizeof(int));
    }

    // Инициализация и вывод массива
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            array[i][j] = i * cols + j;
            printf("%d ", array[i][j]);
        }
        printf("\n");
    }

    // Удаление массива
    for (int i = 0; i < rows; ++i) {
        free(array[i]);
    }
    free(array);

    return 0;
}
```

В этом примере память выделяется с помощью функции malloc. После использования массива память освобождается с помощью функции free.

19. Динамические массивы. Обращение к i-й ячейке одномерного массива с использованием указателя и смещения. Обращение к ячейке двумерного массива на пересечении i-й строки и j-го столбца с использованием указателя и смещения

Динамические массивы Динамические массивы позволяют выделять память во время выполнения программы, что делает их более гибкими по сравнению со статическими массивами.

Обращение к i-й ячейке одномерного массива Обращение к элементам одномерного массива с использованием указателей и смещения:

```
#include <iostream>
using namespace std;

int main() {
    int n = 5;
    int* array = new int[n];

    // Инициализация массива
    for (int i = 0; i < n; ++i) {
```

```
        array[i] = i;
    }

    // Обращение к элементам с использованием указателя и смещения
    for (int i = 0; i < n; ++i) {
        cout << *(array + i) << " ";
    }
    cout << endl;

    delete[] array;

    return 0;
}
```

Обращение к ячейке двумерного массива Обращение к элементам двумерного массива с использованием указателей и смещения:

```
#include <iostream>
using namespace std;

int main() {
    int rows = 3;
    int cols = 4;
    int** array = new int*[rows];

    for (int i = 0; i < rows; ++i) {
        array[i] = new int[cols];
    }

    // Инициализация массива
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            array[i][j] = i * cols + j;
        }
    }

    // Обращение к элементам с использованием указателя и смещения
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            cout << *(*(array + i) + j) << " ";
        }
        cout << endl;
    }

    for (int i = 0; i < rows; ++i) {
        delete[] array[i];
```

```
    }
    delete[] array;

    return 0;
}
```

Этот подход позволяет более гибко и эффективно работать с массивами в C++ и C.

20. Алгоритмы обработки диагоналей и треугольников квадратной матрицы, сортировка элементов матрицы по строке (по столбцу)

Обработка диагоналей и треугольников квадратной матрицы В квадратной матрице можно выделить две главные диагонали:

- Главная диагональ: элементы с индексами (i, i).
- Побочная диагональ: элементы с индексами (i, n-1-i), где n — размер матрицы.

Пример обработки главной диагонали (нахождение суммы)

```
#include <iostream>
using namespace std;

int main() {
    const int n = 3;
    int matrix[n][n] = {{1, 2, 3},
                        {4, 5, 6},
                        {7, 8, 9}};
    int sum = 0;

    for (int i = 0; i < n; ++i) {
        sum += matrix[i][i];
    }

    cout << "Sum of main diagonal: " << sum << endl;

    return 0;
}
```

Пример обработки побочной диагонали (нахождение суммы)

```
#include <iostream>
using namespace std;

int main() {
    const int n = 3;
    int matrix[n][n] = {{1, 2, 3},
```

```
                    {4, 5, 6},
                    {7, 8, 9}};
    int sum = 0;

    for (int i = 0; i < n; ++i) {
        sum += matrix[i][n - 1 - i];
    }

    cout << "Sum of secondary diagonal: " << sum << endl;

    return 0;
}
```

Обработка треугольников матрицы Треугольники в квадратной матрице можно разделить на верхний и нижний треугольники относительно главной диагонали.

Пример обработки верхнего треугольника (нахождение суммы)

```
#include <iostream>
using namespace std;

int main() {
    const int n = 3;
    int matrix[n][n] = {{1, 2, 3},
                        {4, 5, 6},
                        {7, 8, 9}};
    int sum = 0;

    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            sum += matrix[i][j];
        }
    }

    cout << "Sum of upper triangle: " << sum << endl;

    return 0;
}
```

Пример обработки нижнего треугольника (нахождение суммы)

```
#include <iostream>
using namespace std;

int main() {
    const int n = 3;
```

```
    int matrix[n][n] = {{1, 2, 3},
                        {4, 5, 6},
                        {7, 8, 9}};
    int sum = 0;

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            sum += matrix[i][j];
        }
    }

    cout << "Sum of lower triangle: " << sum << endl;

    return 0;
}
```

Сортировка элементов матрицы по строке (по столбцу) Пример сортировки по строкам

```
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    const int n = 3;
    int matrix[n][n] = {{3, 2, 1},
                        {9, 8, 7},
                        {6, 5, 4}};

    for (int i = 0; i < n; ++i) {
        sort(matrix[i], matrix[i] + n);
    }

    // Вывод отсортированной матрицы
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Пример сортировки по столбцам

```
#include <iostream>
```

```cpp
#include <algorithm>
using namespace std;

int main() {
    const int n = 3;
    int matrix[n][n] = {{3, 2, 1},
                        {9, 8, 7},
                        {6, 5, 4}};

    for (int j = 0; j < n; ++j) {
        int temp[n];
        for (int i = 0; i < n; ++i) {
            temp[i] = matrix[i][j];
        }
        sort(temp, temp + n);
        for (int i = 0; i < n; ++i) {
            matrix[i][j] = temp[i];
        }
    }

    // Вывод отсортированной матрицы
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

21. Порядок расположения адресов переменных программы. Понятие указателя, допустимые операции над указателями. Непрямой доступ к значению переменной. Указатель на указатель, указатель на void, указатель и модификатор const

Порядок расположения адресов переменных программы В программах на C++ переменные располагаются в различных сегментах памяти:

- Стек (stack): для локальных переменных и параметров функций.
- Куча (heap): для динамически выделяемой памяти (например, с помощью new или malloc).
- Сегмент данных (data segment): для глобальных и статических переменных.

Понятие указателя Указатель — это переменная, которая хранит адрес другой переменной. Указатели позволяют эффективно работать с памятью, передавать большие объекты по ссылке, а также создавать динамические структуры данных.

Допустимые операции над указателями

- Присваивание адреса переменной указателю: int *p = &x;
- Разыменование указателя (доступ к значению по адресу): *p = 10;
- Арифметика указателей: p + 1, p - 1 (перемещение по массиву).

Непрямой доступ к значению переменной Непрямой доступ к значению переменной осуществляется через разыменование указателя:

```
int x = 10;
int *p = &x;
*p = 20; // изменяет значение x на 20
```

Указатель на указатель Указатель на указатель хранит адрес другого указателя:

```
int x = 10;
int *p = &x;
int **pp = &p;

cout << **pp << endl; // выводит 10
```

Указатель на void Указатель типа void* может хранить адрес любой переменной, но не может быть разыменован напрямую:

```
int x = 10;
void *p = &x;
// int y = *p; // ошибка
int y = *(int*)p; // необходимо приведение типа
```

Указатель и модификатор const Модификатор const может применяться к указателям различными способами:

- const int *p: указатель на константное значение (значение нельзя изменять через указатель).
- int *const p: константный указатель (указатель не может указывать на другой адрес).
- const int *const p: константный указатель на константное значение.

22. Функциональное назначение и синтаксис определения функций пользователя. Формальные и фактические параметры. Оператор return и вызов функции. Механизм передачи параметров в функцию по значению

Функциональное назначение и синтаксис определения функций Функции позволяют разбивать программу на логические части, которые можно многократно использовать. Функция определяет блок кода, который выполняет конкретную задачу.

```
#include <iostream>
using namespace std;

// Объявление функции
int sum(int a, int b);

// Определение функции
int sum(int a, int b) {
    return a + b;
}

int main() {
    int result = sum(3, 4); // Вызов функции
    cout << "Sum: " << result << endl;
    return 0;
}
```

Формальные и фактические параметры

- Формальные параметры: переменные, указанные в определении функции (int a, int b).
- Фактические параметры: значения, передаваемые при вызове функции (sum(3, 4)).

Оператор return Оператор return завершает выполнение функции и возвращает значение:

```
int sum(int a, int b) {
    return a + b; // Возвращает сумму a и b
}
```

Механизм передачи параметров в функцию по значению

Передача параметров по значению означает, что в функцию передается копия аргумента. Изменения в параметрах функции не влияют на оригинальные переменные:

```
void increment(int x) {
    x = x + 1;
```

```
}

int main() {
    int a = 5;
    increment(a);
    cout << "a: " << a << endl; // a останется 5
    return 0;
}
```