1. Указатели. Инициализация указателей. Операции с указателями.

2. Указатели в C++. Операция разыменования. Константные указатели и указатели на константы. Ссылки в C++.

3. Массивы. Динамические массивы. Динамические многомерные массивы.

4. Строки. Объявление и инициализация.

5. Функции для работы со строками и символами.

6. Структуры. Инициализация. Доступ к полям.

7. Структуры. Массивы структур.

8. Передача структур в функции

9. Указатели на структуры

10-11. Битовые поля. Объединения. Инициализация. Доступ к полям

12. Перечисления. Инициализация. Доступ к полям.

13. Классы памяти переменных

14. Область действия переменных

15. Модульное программирование.

17. Определение и вызов функций. Фактические и формальные параметры. Передача параметров в функции по значению, по ссылке, по указателю

18. Функции. Возвращаемые значения. Глобальные и локальные данные.

16. Подпрограммы. Преимущества и недостатки использования. Виды.

19. Функции. Параметры со значением по умолчанию. Способы передачи параметров

20. Функции. С переменным числом параметров.

21. Функции. Указатели на функции. Массив указателей на функции.

22. Параметры функции main()

23. Функции. Рекурсивные функции. Распределение памяти при рекурсивном вызове. Типы рекурсии. Рекомендации по использованию рекурсивного вызова.

24. Динамические структуры данных. Рекомендации для использования. Основные особенности.

25. Стек. Создание стека. Общие принципы работы

26. Стек. Удаление элемента. Добавление элемента. Общие принципы работы

27. Очередь. Создание очереди.

28. Очередь. Удаление элемента. Добавление элемента. Общие принципы работы

29. Линейные списки. Создание элемента списка.

30. Линейные списки. Добавление узла в начало списка.

31. Линейные списки. Поиск элемента.

32. Линейные списки. Удаление узла.

33. Двусвязный список. Создание. Удаление узла.

34. Двусвязный список. Создание. Добавление узла.

35. Деревья. Общие понятия.

36. Бинарные деревья

37. Обход бинарного дерева.

38. Бинарные деревья. Поиск в дереве.

39. Сбалансированные деревья

40. Файлы. Типы файлов.

41. Доступ к файлам.

42. Файлы. Текстовые файлы. Открытие \ закрытие.

43. Текстовые файлы. Ввод \ вывод в текстовые файлы.

44. Файлы. Бинарные файлы.

44. Бинарные файлы. Открытие \ закрытие файла. Диагностика ошибок.

45. Бинарные файлы. Функции чтения \ записи .

46. Бинарные файлы. Позиционирование в файле.

47. Потоки. Виды потоков. Классы и объекты потоков ввода-вывода.

48. Файловые потоки.

49. Средства работы с потоками ввода/вывода.

50. Форматирование ввода-вывода. Использование манипуляторов.

59. Поиск. Линейный.

60. Поиск. Двоичный.

53. Пузырьковая сортировка.

54. Сортировка методом простого включения.

55. Сортировка выбором.

56. Сортировка Хоара

57. Сортировка Шейкерная.

**1. Указатели. Инициализация указателей. Операции с указателями.**

**Указатель** — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект. Указатели широко используются в программировании на языке Си. Указатели часто используются при работе с массивами.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками. Каждая переменная в памяти имеет свой адрес - номер ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной.   
Признаком указателя для компилятора является наличие в описании переменной двух компонентов:  
1) типа объекта данных, для доступа к которому используется указатель (или, как часто говорят, на который ссылается указатель);  
2) символа \* перед именем переменной.

Указатель, как и любая переменная, должен быть объявлен.  
Общая форма объявления указателя:  
**тип \*имя объекта;**

Тип указателя— это тип переменной, адрес которой он содержит.

Для работы с указателями в Си определены две операции:

* операция \* (звездочка) — позволяет получить значение объекта по его адресу - определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;
* операция & (амперсанд) — позволяет определить адрес переменной.
* Над указателями можно выполнять унарные операции: инкремент и декремент. При выполнении операций ++ и — значение указателя увеличивается или уменьшается на длину типа, на который ссылается используемый указатель.
* В бинарных операциях сложения и вычитания могут участвовать указатель и величина типа int. При этом результатом операции будет указатель на исходный тип, а его значение будет на указанное число элементов больше или меньше исходного.
* В операции вычитания могут участвовать два указателя на один и тот же тип. Результат такой операции имеет тип int и равен числу элементов исходного типа между уменьшаемым и вычитаемым, причем если первый адрес младше, то результат имеет отрицательное значение.

Значения двух указателей на одинаковые типы можно сравнивать в операциях ==, !=, <, <=",">, >= при этом значения указателей рассматриваются просто как целые числа, а результат сравнения равен 0 (ложь) или 1 (истина).

**2. Указатели в C++. Операция разыменования. Константные указатели и указатели на константы. Ссылки в C++.**

Указатели в С++. Указатель – переменная, значением которой является адрес ячейки памяти. Указатель может ссылаться на переменную или функцию. Для этого нужно знать адрес переменной или функции. Указатели используются для передачи по ссылке данных, что намного ускоряет процесс обработки этих данных, так как их не надо копировать, как при передаче по значению, то есть, используя имя переменной. В основном указатели используются для организации динамического распределения памяти. Указатели могут ссылаться на другие указатели. При этом в ячейках памяти, на которые будут ссылаться первые указатели, будут содержаться не значения, а адреса вторых указателей. Чтобы получить доступ к значению, на которое ссылается указатель его необходимо разыменовывать соответствующее количество раз.

Указатели могут ссылаться на функции. Имя функции, как и имя массива само по себе является указателем, то есть содержит адрес входа.

Тип данных определяем такой, который будет возвращать функция, на которую будет ссылаться указатель.

int \*ptrvar;        // объявление указателя

     ptrvar = &var; // инициализация указателя

\*тип данных\* (\* \*имя указателя\*)(\*список аргументов функции\*); //объявление указателя на функцию

**Операция разыменования.**

Основной операцией при работе с указателями является получение доступа к значению, адрес которого хранится в указателе. Разыменование - обращения к значению в памяти, на которое указывает указатель. Разыменование может быть явным и неявным; в большинстве современных языков программирования разыменование происходит только при явном указании

Константные указатели и указатели на константы. Указатели могут указывать как на переменные, так и на константы. Чтобы определить указатель на константу, он тоже должен объявляться с ключевым словом const. Они не могут изменять адрес, который в них хранится, но могут изменять значение по этому адресу. Константный указатель на константу

И объединение обоих предыдущих случаев - константный указатель на константу, который не позволяет менять ни хранимый в нем адрес, ни значение по этому адресу

**Ссылки в C++.**Ссылка — это тип переменной в C++, который работает как псевдоним другого объекта или значения. C++ поддерживает три типа ссылок:

* Ссылки на неконстантные значения (обычно их называют просто «ссылки» или «неконстантные ссылки»), которые мы обсудим в этом уроке.
* Ссылки на константные значения (обычно их называют «константные ссылки»), которые мы обсудим в следующем уроке.
* Ссылка (на неконстантное значение) объявляется с использованием амперсанда (&) между типом данных и именем ссылки.

**Ссылки в качестве псевдонимов**. Ссылки обычно ведут себя идентично значениям, на которые они ссылаются. В этом смысле ссылка работает как псевдоним объекта, на который она ссылается

**Инициализация ссылок**. Cсылки нулевыми быть не могут, в отличии от указателей. После инициализации изменить объект, на который указывает ссылка — нельзя.

Ссылки чаще всего используются в качестве параметров в функциях. В этом контексте ссылка-параметр работает как псевдоним аргумента, и сам аргумент не копируется при передаче в параметр. Это, в свою очередь, улучшает производительность, если аргумент слишком большой или затратный для копирования.

**3. Массивы. Динамические массивы. Динамические многомерные массивы.**

**Массив** - это непрерывный участок памяти, содержащий последовательность объектов одинакового типа, обозначаемый одним именем.

Элемент массива (значение элемента массива) – значение, хранящееся в определенной ячейке памяти, расположенной в пределах массива, а также адрес этой ячейки памяти.

Каждый элемент массива в свою очередь характеризуется тремя величинами:  
адресом элемента - адресом начальной ячейки памяти, в которой расположен этот элемент;2индексом элемента (порядковым номером элемента в массиве);3значением элемента.

Адрес массива – адрес начального элемента массива.Имя массива – идентификатор, используемый для обращения к элементам массива.Размер массива – количество элементов массиваРазмер элемента – количество байт, занимаемых одним элементом массива.Длина массива – количество байт, отводимое в памяти для хранения всех элементов массива.

*Длина Массива = Размер Элемента \* Количество Элементов*

Признаком массива при описании является наличие парных скобок []. Константа или константное выражение в квадратных скобках задает число элементов массива: charbuffer[81]; intkey [4];  
Существуют две формы явной инициализации элементов массива:

* явное указание числа элементов массива и список начальных значений, возможно, с меньшим числом элементов. Например,chararray[10] ={'A','B','C','D'};Если список начальных значений содержит больше элементов, чем число в квадратных скобках, генерируется сообщение об ошибке;
* только со списком начальных значений. Компилятор определяет число элементов массива по списку инициализации.Например,chararray[] = {'A',B','C','D'}; создает массив ровно из 4 вышеуказанных элементов.

**(+см14)Динамическим** называется [массив](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29), размер которого может меняться во время исполнения программы. Для изменения размера динамического массива [язык программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), поддерживающий такие массивы, должен предоставлять встроенную функцию или оператор. Динамические массивы дают возможность более гибкой работы с данными, так как позволяют не прогнозировать хранимые объёмы данных, а регулировать размер массива в соответствии с реально необходимыми объёмами. В отличие от динамических массивов существуют статические массивы и массивы переменной длины. Размер статического массива определяется на момент компиляции программы. Размер массива переменной длины определяется во время выполнения программы. Отличием динамического массива от массива переменной длины является автоматическое изменение размеров, что не трудно реализуется в случаях его отсутствия, поэтому часто не различают массивы переменной длины с динамическими массивами.

**Для С++**

ВС++ операции new и delete предназначены для динамического распределения памяти компьютера.  Операция new  выделяет память из области свободной памяти, а операция delete высвобождает выделенную память.  Выделяемая память, после её использования должна высвобождаться, поэтому операции new и delete используются парами.

int\*ptrvalue = newint;//где ptrvalue – указатель на выделенный участок памяти типа int//new – операция выделения свободной памяти под создаваемый объект.

deleteptrvalue;// где ptrvalue – указатель на выделенный участок памяти типа int // delete – операция высвобождения памяти

По определению, многомерные массивы в С++ не существуют, массив всегда считается одномерным. Однако в С++ разрешается объявлять одномерные массивы массивов(т.е. многомерные). В памяти многомерный массив располагается в последовательных ячейках по слоям(строках).

*Динамические массивы передаются в функции,* **передается указатель на начало массива указателей,** а длина строки и количество строк вообще нигде не фигурируют. Контроль за границами массивов лежит полностью на программисте, поэтому, вероятно, стоит передавать в функцию отдельными параметрами размеры массива — количество строк и столбцов.

Создание такой сложной структуры как двумерный динамический массив требует множества системных вызовов по выделению памяти:

int \*\*A = (int \*\*)malloc(N\*sizeof(int \*));  
for(int i = 0; i < N; i++) {    A[i] = (int \*)malloc(M\*sizeof(int));}

При таком выделении памяти нельзя просто взять, и освободить память по адресу A, т.к. будет возникать утечка памяти.

Правильноеочищениетаково:

for(int i = 0; i < N; i++) {    free(A[i]);} free(A);

**Ещё пример**:

p=new(n);

for(int i=0;i<n;i++)p[i]=new(p[i]);

…

for(int i=0;i<n;i++)p[i]=delete(p[i]);

delete(p);

Функция malloc выделяет блок памяти, размером sizemem байт, и возвращает указатель на начало блока.

Функция calloc выделяет блок памяти для массива размером — num элементов, каждый из которых занимает   size байт, и инициализирует все свои биты в нулями.

**13. Классы памяти переменных**

Классы памяти определяют область видимости и время жизни переменных и/или функций в программе, написанной на языке С. Они предшествуют типу, который модифицируют. В языке С различают четыре класса памяти:

* auto (автоматический);
* register (регистровый);
* static (статический);
* extern (внешний).

Все локальные переменные по умолчанию принадлежат классу памяти auto, то есть являются автоматическими.

Класс памяти register используется для определения локальных переменных, которые должны храниться в регистре процессора, а не в оперативной памяти.

Класс памяти staticинструктирует компилятор поддерживать локальную переменную в течение всего времени жизни программы вместо того, чтобы создавать и уничтожать ее каждый раз, когда она входит и выходит из области видимости. Таким образом, статичность локальных переменных позволяет им сохранять свои значения между вызовами функций.

Класс памяти **extern** используется для того, чтобы дать ссылку на глобальную переменную, которая видна ВСЕМ файлам программы. Использование extern не может инициализировать переменную, однако указывает на имя переменной в месте памяти, определенное ранее.

**14. Область действия переменных**

Область действия переменной – это правила, которые устанавливают, какие данные доступны из текущего места программы. Имеются три типа переменных: глобальные, локальные и формальные. Область действия локальных переменных – это те блоки, где локальные переменные объявлены. При выходе из блока локальная переменная и ее значение теряются.

Формальные переменные – это параметры в заголовке функции пользователя. Формальные параметры используются в функции так же, как локальные переменные. Область действия формальных параметров – блок, являющийся кодом функции.

Глобальными переменными называются те переменные, которые были созданы вне тела какого-то блока. Их можно всегда использовать во всей вашей программе, вплоть до ее окончания работы.

**4.Строки. Объявление и инициализация.**

Строка представляет собой массив символов, заканчивающийся нуль-символом. Нуль-символ — это символ с кодом, равным О, что записывается в виде управляющей последовательности. По положению нуль-символа определяется фактическая длина строки. Длина Строки определяется с помощью функции strlen, которая вычисляет длину, выполняя поиск нуль-символа.

Объявление и инициализация.

Строки реализуются посредством массивов символов.

char имя[длина];

Объявление строки в си имеет тот же синтаксис, что и объявление одномерного символьного массива. Длина строки указывается с учетом одного символа на хранение завершающего нуля, поэтому максимальное количество значащих символов в строке на единицу меньше ее длины. Объявление массивов строк в языке СИ 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.

**6.Функции для работы со строками и символами.**

[**strlen**](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstring-string-h/funkciya-strlen/)**(**имя\_строки**)**определяет длину указанной строки, без учёта нуль-символа

**Копирование строк**

[**strcpy**](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstring-string-h/funkciya-strcpy/)**(**s1,s2**)**выполняет побайтное копирование символов из строки  s2 в строку s1

[**strncpy**](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstring-string-h/funkciya-strncpy/)**(**s1,s2, n**)**выполняет побайтное копирование n символов из строки  s2 в строку s1. возвращает значения s1

**Конкатенация строк**

[**strcat(**s1,s2**)**](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstring-string-h/funkciya-strcat/)объединяет строку s2 со строкой s1. Результат сохраняется в s1

[**strncat(**s1,s2,n**)**](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstring-string-h/funkciya-strncat/)объединяет n символов строки s2 со строкой s1. Результат сохраняется в s1

**Сравнение строк**

[**strcmp(**s1,s2**)**](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstring-string-h/funkciya-strcmp/)сравнивает строку s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2,  <0  — если s1>s2 С учётом регистра

[**strncmp(**s1,s2**)**](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstring-string-h/funkciya-strncmp/)сравнивает n символов строки s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2,  <0  — если s1>s2 С учётом регистра

**stricmp(**s1,s2**)**сравнивает строку s1 со строкой s2 и возвращает результат типа **int**: 0 –если строки эквивалентны, >0 – если s1<s2,  <0  — если s1>s2 Без учёта регистра

**strnicmp(**s1,s2**)**сравнивает n символов строки s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2,  <0 — если s1>s2 Без учёта регистра

**Функции преобразования**

[**atof(**s1**)**](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstdlib-stdlib-h/funkciya-atof/)преобразует строку s1 в тип double

[**atoi(**s1**)**](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstdlib-stdlib-h/funkciya-atoi/)преобразует строку s1 в тип int

[**atol(**s1**)**](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstdlib-stdlib-h/funkciya-atol/)преобразуетстроку s1 втип long int

**Функции стандартной библиотеки ввода/вывода <stdio>**

[**getchar(**с**)**](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstdio-stdio-h/funkciya-getchar/)считывает символ ссо стандартного потока ввода, возвращает символ в формате int

[**gets(**s**)**](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstdio-stdio-h/funkciya-gets/)считывает поток символов со стандартного устройства ввода в строку sдо тех пор, пока не будет нажата клавиша ENTER

**12.Перечисления. Инициализация. Доступ к полям.**

**Перечисления** – средство создания типа данных посредством задания ограниченного множества значений.

Определение перечисляемого типа данных имеет вид

enum ID\_перечисляемого\_типа

{ список\_значений};

Значения данных перечисляемого типа указываются идентификаторами, например:

enum marks {

zero, one, two, three, four, five

};

Компилятор последовательно присваивает идентификаторам списка значений целочисленные величины 0, 1, 2,... . При необходимости можно явно задать значение идентификатора, тогда очередные элементы списка будут получать последующие возрастающие значения. Например:

enumlevel {

low=100, medium=500, high=1000, limit };

Константа limit по умолчанию получит значение, равное 1001.

Примеры объявления переменных перечисляемого типа:

enum marks Est;

enum level state;

Переменная типа marks может принимать только значения из множества {zero, one, two, three, four, five}.

Основные операции с данными перечисляемого типа:

– присваивание переменных и констант одного типа;

– сравнение для выявления равенства либо неравенства.

Практическое назначение перечисления – определение множества различающихся символических констант целого типа.

См 10 билет

**6.Структуры. Инициализация. Доступ к полям.**

В отличие от массива, все элементы которого однотипны, структура может содержать элементы разных типов.

struct [ имя\_типа ]

{ тип\_1 элемент\_1;

тип\_2 элемент\_2;

тип^п элемент^п; }

[ список^описателей ];

Элементы структуры называются полями структуры и могут иметь любой тип, кроме типа этой же структуры, но могут быть указателями на него. Если отсутствует имя типа, должен быть указан список описателей переменных, указателей или массивов.

Для инициализации структуры значения ее элементов перечисляют в фигурных скобках в порядке их описания:

struct{

char fio[30];

int date, code;

doublesalary;

}worker = { 1,2,3,4,5};

При инициализации массивов структур следует заключать в фигурные скобки каждый элемент массива (учитывая, что многомерный массив — это массив массивов).

Для переменных одного и того же структурного типа определена операция присваивания, при этом происходит поэлементное копирование. Структуру можно передавать в функцию и возвращать в качестве значения функции. Другие операции со структурами могут быть определены пользователем. Доступ к полям структуры выполняется с помощью операций выбора . (точка) при обращении к полю через имя структуры и -> при обращении через указатель, например.

Доступ к полям структуры производится по имени поля (а не по индексу, как у массивов):   
**имя**\_**структурной**\_**переменной***.***имя**\_**поля**  
**указатель**\_**на**\_**структуру***->***имя**\_**поля**

См 10 билет

**7.Структуры. Массивы структур.**

Если массивы по определению могут содержать внутри себя множество однотипных элементов, то внутри структур можно описать разное количество разнотипных элементов. А сами по себе **структуры являются пользовательским типом данных**.

Структуры часто образуют массивы. Чтобы объявить массив структур, вначале необходимо определить структуру (то есть определить агрегатный тип данных), а затем объявить переменную массива этого же типа. Чтобы получить доступ к определенной структуре, указывайте имя массива с индексом. Как и в других массивах переменных, в массивах структур индексирование начинается с 0.

Чтобы указать определенную структуру, находящуюся в массиве структур, необходимо указать имя этого массива с определенным индексом. А если нужно указать индекс определенного элемента в структуре, то необходимо указать индекс этого элемента.

8. Передача структур в функции

Когда структура используется как параметр функции, то она передается в функцию по значению, как и принято в языке программирования С. Например:

struct emp

{int empno;

char name[20];

double salary;};

void print(struct emp s) {printf("%d %s %f\n", s.empno, s.name, s.salary);}

Если необходимо, чтобы функция изменяла значение членов структуры, то в функцию передается указатель на структуру. В этом случае для доступа к членам структуры используется оператор ->, который имеет следующий вид: указатель->имя\_поля

struct emp{int empno;

char name[20];

double salary;};

void init\_emp(struct emp \*ps, int en, char \*nm, double sal) {ps->empno = en; strcpy(ps->name, nm); ps->salary = sal;}

9. Указатели на структуры

Как и другие указатели, указатель на структуру объявляется с помощью звездочки \*, которую помещают перед именем переменной структуры. Например, для ранее определенной структуры addr следующее выражение объявляет addr\_pointer указателем на данные этого типа (то есть на данные типа addr): struct addr \*addr\_pointer;

Указатели на структуры используются главным образом в двух случаях: когда структура передается функции с помощью вызова по ссылке, и когда создаются связанные друг с другом списки и другие структуры с динамическими данными, работающие на основе динамического размещения.

У такого способа, как передача любых (кроме самых простых) структур функциям, имеется один большой недостаток: при выполнении вызова функции, чтобы поместить структуру в стек, необходимы существенные ресурсы. Впрочем, для простых структур с несколькими членами эти ресурсы являются не такими уж большими. Но если в структуре имеется большое количество членов или некоторые члены сами являются массивами, то при передаче структур функциям производительность может упасть до недопустимо низкого уровня.

Когда функции передается указатель на структуру, то в стек попадает только адрес структуры. В результате вызовы функции выполняются очень быстро.

Чтобы с помощью указателя на структуру получить доступ к ее членам, необходимо использовать оператор стрелка ->. Оператор ->, который обычно называют оператором стрелки, состоит из знака "минус", за которым следует знак "больше". Стрелка применяется вместо оператора точки тогда, когда для доступа к члену структуры используется указатель на структуру. Впрочем, для простых структур с несколькими членами эти ресурсы являются не такими уж большими. Но если в структуре имеется большое количество членов или некоторые члены сами являются массивами, то при передаче структур функциям производительность может упасть до недопустимо низкого уровня. Как же решить эту

**10-11.Битовые поля. Объединения. Инициализация. Доступ к полям.**

**Битовые поля** – это особый вид полей структуры. Они используются для плотной упаковки данных, например, флажков типа «да/нет». Минимальная адресуемая ячейка памяти – 1 байт, а для хранения флажка достаточно одного бита. При описании битового поля после имени через двоеточие указывается длина поля в битах (целая положительная константа), не превышающая разрядности поля типа int:

struct fields {unsigned int flag: 1;unsigned int mask: 10;unsignedintcode: 5;};

Битовые поля могут быть любого целого типа. Имя поля может отсутствовать, такие поля служат для выравнивания на аппаратную границу. Доступ к полю осуществляется обычным способом – по имени. Единственное отличие этих полей от обычных объектов – запрет операции определения адреса (&). Следует учитывать, что использование битовых полей снижает быстродействие программы по сравнению с представлением данных в полных полях из-за необходимости выделения битового поля.

**Объединение** – поименованная совокупность данных разных типов, размещаемых с учетом выравнивания в одной и той же области памяти, размер которой достаточен для хранения наибольшего элемента.

Объединенный тип данных декларируется подобно структурному типу:

union ID\_объединения { описание полей };

Пример описания объединенного типа:

union word {int nom;charstr[20];};

Объединения применяют для экономии памяти в случае, когда объединяемые элементы логически существуют в разные моменты времени либо требуется разнотипная интерпретация поля данных.

**Обращение к полям структур**

Обращение к полям структур производится путем создания составных имен, которые образуются двумя способами:

1) при помощи операции принадлежности (.) общий вид которой

2) при помощи операции косвенной адресации (–>) в виде

Если в программе созданы объекты объявленного ранее шаблона:

Stud\_Type s1, \*s2; то к полям объекта s1 можно обратиться следующим образом:s1. Number, s1.Fio, s1. S\_b; или(&s1) –> Number, (&s1) –> Fio, (&s1) –> S\_b;

а к полям объекта, адрес которого s2:s2 –>Number, s2 –>Fio, s2 –>S\_b; или (\*s2) . Number, (\*s2) . Fio, (\*s2) . S\_b;

Структуры могут быть вложенными, т.е. поле структуры может быть связующим полем с внутренней структурой, описание которой должно предшествовать по отношению к основному шаблону.

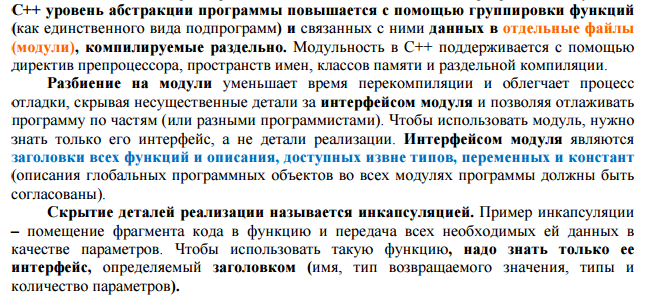
Объявляем переменную и указатель на переменные такой структуры:

struct Person a, \*p; Инициализируем указатель p адресом переменной а: p = &a;

Тогда обращение к полям структурной переменной a будет выглядеть следующим образом:a . fioa. f1 . daya . f1 . montha . f1 . yearилиp–>fiop–>f1.dayp–>f1.monthp–>f1.year

**15.Модульное программирование.**

в

Концепцию модульного программирования можно сформулировать в виде нескольких понятий и положений:

* Функциональная декомпозиция задачи - разбиение большой задачи на ряд более мелких, функционально самостоятельных подзадач - модулей. Модули связаны между собой только по входным и выходным данным.
* Модуль - основа концепции модульного программирования. Каждый модуль в функциональной декомпозиции представляет собой "[черный ящик](http://wiki.mvtom.ru/index.php?title=%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%D1%87%D0%B5%D1%80%D0%BD%D0%BE%D0%B3%D0%BE_%D1%8F%D1%89%D0%B8%D0%BA%D0%B0&action=edit&redlink=1)" с одним входом и одним выходом. Модульный подход позволяет безболезненно производить модернизацию программы в процессе ее эксплуатации и облегчает ее сопровождение. Дополнительно модульный подход позволяет разрабатывать части программ одного проекта на разных языках программирования, после чего с помощью [компоновочных средств](http://wiki.mvtom.ru/index.php/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%BE%D0%B2%D1%89%D0%B8%D0%BA) объединять их в единый загрузочный модуль.
* Реализуемые решения должны быть простыми и ясными. Если назначение модуля непонятно, то это говорит о том, что декомпозиция начальной или промежуточной задачи была проведена недостаточно качественно. В этом случае необходимо еще раз проанализировать задачу и, возможно, провести дополнительное разбиение на подзадачи. При наличии сложных мест в проекте их нужно подробнее документировать с помощью продуманной системы комментариев. Этот процесс нужно продолжать до тех пор, пока действительно не удастся добиться ясного понимания назначения всех модулей задачи и их оптимального сочетания.
* Назначение всех переменных модуля должно быть описано с помощью комментариев по мере их определения.

**17. Определение и вызов функций. Фактические и формальные параметры. Передача параметров в функции по значению, по ссылке, по указателю**

Функция декларация сообщает компилятор о имени функции и как вызвать функцию. Фактическое тело функции можно определить отдельно.

Объявление функции имеет следующие части -

return\_type function\_name( parameter list );

При создании функции C ++ вы даете определение того, что должна делать функция. Чтобы использовать функцию, вам придется её вызвать. Когда программа вызывает функцию, управление программой передается вызываемой функции. Вызываемая функция выполняет определенную задачу, и когда выполняется оператор return или когда достигается завершающая ее конечная скобка, она возвращает управление программой обратно в основную программу.

function\_name( parameter list );

Фактическими параметрами называются параметры, которые задаются при вызове процедуры (функции).

Формальные параметры ведут себя как другие локальные переменные внутри функции и создаются при входе в функцию и уничтожаются при выходе.

При вызове функции существует два способа передачи аргументов функции -

|  |  |
| --- | --- |
| Call by Value (Вызов по значению) | Этот метод копирует фактическое значение аргумента в формальный параметр функции. В этом случае изменения, внесенные в параметр внутри функции, не влияют на аргумент. |
| Call by Pointer (Вызов по указателю) | Этот метод копирует адрес аргумента в формальный параметр. Внутри функции адрес используется для доступа к фактическому аргументу, используемому в вызове. Это означает, что изменения, внесенные в параметр, влияют на аргумент. |
| Call by Reference (Вызов по ссылке) | Этот метод копирует ссылку аргумента в формальный параметр. Внутри функции ссылка используется для доступа к фактическому аргументу, используемому в вызове. Это означает, что изменения, внесенные в параметр, влияют на аргумент. |

**18.Функции. Возвращаемые значения. Глобальные и локальные данные.**

Функция - это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение. Разделение программы на функции позволяет также избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно из разных точек программы многократно.Любая программа состоит из функций, одна из которых должна иметь имя main (с нее начинается выполнение программы).

Функция начинает выполняться в момент вызова.

Возвращаемые значения.

Все функции, кроме функций типа void, возвращают значения. Данное значение определяется в операторе return. Если функция не определена как void и если не указано возвращаемое значение, то возвращается мусор. Если функция не объявлена как void, она может использоваться в качестве операнда в любом корректном выражении.

Если функция объявляется как void, она не может использоваться в выражениях. Например, предположим, что f() объявлена как void.

Хотя все функции не типа void имеют значения возврата, при написании программ обычно используется три типа функций. Первый тип - это вычислительные функции. Он предназначен для выполнения операций с аргументами и возвращает значение, основываясь на этих операциях. Примером таких функций являются sqr() и sin() - стандартные библиотечные функции.

Следующий тип функций обрабатывает информацию и возвращает значение, показывающее, была ли работа успешной или привела к ошибке. Примером является fwrite() - функция, используемая для записи информации в файл. Если запись проведена успешно, fwrite() возвращает число удачно записанных элементов. Если возникла ошибка, возвращаемое число не равно числу элементов, которые требовалось записать.

Последний тип функций не имеет определенного возвращаемого значения. Функция является обычной процедурой и не выдает значения. Примером служит srand(), используемая для инициализации генератора случайных чисел функции rand(). Иногда функции, не выдающие осмысленного результата, что-то все-таки выдают. Например, printf() возвращает число напечатанных символов. Очень трудно найти программу, которая проверяет это. Следовательно, хотя все функции, кроме функций, объявленных как void, возвращают значения, нет необходимости использовать все эти значения.

В противоположность локальным переменным глобальные переменные видны всей программе и могут использоваться любым участком кода. Они хранят свои значения на протяжении всей работы программы. Глобальные переменные создаются путем объявления вне функции. К ним можно получить доступ в любом выражении, независимо от того, в какой функции находится данное выражение.

Глобальные переменные хранятся в фиксированной области памяти, устанавливаемой компилятором. Глобальные переменные чрезвычайно полезны, когда одни и те же данные используются в нескольких функциях программы. Следует избегать ненужного использования глобальных переменных по трем причинам:

Они используют память в течение всего времени работы программы, а не тогда, когда они необходимы.

Использование глобальных переменных вместо локальных приводит к тому, что функции становятся более частными, поскольку они зависят от переменных, определяемых снаружи.

Использование большого числа глобальных переменных может вызвать ошибки в программе из-за неизвестных и нежелательных эффектов.

**16. Подпрограммы. Преимущества и недостатки использования. Виды.**

Подпрограмма-поименованная или иным образом идентифицированная часть [компьютерной программы](https://dic.academic.ru/dic.nsf/ruwiki/6823), содержащая описание определённого набора действий. Подпрограмма может быть многократно вызвана из разных частей программы.

Преимущества использования подпрограмм:

* [декомпозиция](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BE%D0%BC%D0%BF%D0%BE%D0%B7%D0%B8%D1%86%D0%B8%D1%8F) сложной задачи на несколько более простых подзадач: это один из двух главных инструментов структурного программирование (второй - структуры данных);
* уменьшение [дублирования кода](https://ru.wikipedia.org/wiki/%D0%94%D1%83%D0%B1%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%B4%D0%B0) и возможность [повторного использования кода](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B2%D1%82%D0%BE%D1%80%D0%BD%D0%BE%D0%B5_%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%B4%D0%B0) в нескольких программах
* распределение большой задачи между несколькими разработчиками или стадиями проекта;
* [сокрытие](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BA%D1%80%D1%8B%D1%82%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) деталей реализации от пользователей подпрограммы;
* улучшение отслеживания выполнения кода

Недостатком использования подпрограмм можно считать накладные расходы на вызов подпрограммы, однако современные трансляторы стремятся оптимизировать данный процесс.

В языках программирования высокого уровня используется два типа подпрограмм: процедуры и функции.

Функция — это подпрограмма специального вида, которая, кроме получения параметров, выполнения действий и передачи результатов работы через параметры имеет ещё одну возможность — она может возвращать результат. Вызов функции является, с точки зрения языка программирования, выражением, он может использоваться в других выражениях или в качестве правой части присваивания. Подробнее см. в статье Функция (программирование).

Процедура — это независимая именованная часть программы, которую после однократного описания можно многократно вызвать по имени из последующих частей программы для выполнения определенных действий.

**19. Функции. Параметры со значением по умолчанию. Способы передачи параметров**

При обращении к функции, можно опускать некоторые её аргументы, но для этого необходимо при объявлении прототипа данной функции проинициализировать её параметры какими-то значениями, эти значения и будут использоваться в функции по умолчанию. Аргументы по умолчанию должны быть заданы в прототипе функции. Если в функции несколько параметров, то параметры, которые опускаются должны находиться правее остальных. Таким образом, если опускается самый первый параметр функции, то все остальные параметры тоже должны быть опущены. Если опускается какой-то другой параметр, то все параметры, расположенные перед ним могут не опускаться, но после него они должны быть опущены. Разработаем программу, в которой объявим функцию с аргументами по умолчанию.

double heron\_space(const double a = 5, const double b = 6.5, const double c = 10.7); //параметрыфункцииинициализированыпоумолчанию

Иногда при многократном использовании одной функции необходимо менять не все её параметры, в таком случае использование параметров по умолчанию очень удобно.

Способы:

В языке С допустимы только два способа: по значению и по адресу.

Передача данных по значению

Этот способ передачи данных в подпрограмму является основным и действует поумолчанию. Фактический параметр вычисляется в вызывающей функции и его значение передаётся на место формального параметра в вызываемой функции. На этом связь между фактическим и формальным параметрами прекращается.

В качестве фактического параметра можно использовать константу, переменную или более сложное выражение. Передача данных по значению пригодна только для простых данных, которые являются входными параметрами. Если параметр является выходным данным или массивом, то передача его в функцию по значению не приемлема.

double fsum(double x, double eps);

Передача данных по адресу

По адресу в функцию всегда передаются массивы (рассмотрим это в следующих темах). Для массива это вообще единственный способ передачи данных в языках С/С++. Так же по адресу можно передать те простые объекты, которые являются выходными данными (или входными и выходными одновременно).

В случае передачи данных по адресу фактический параметр может быть только переменной (константа или выражение не имеют адреса!).

void Obmen(double \*x, double \*y);

**20 .Функции. С переменным числом параметров.**

Для того, чтобы получить доступ ко всем параметрам, принимаемых функцией, нужно знать имя и тип хотя бы одного параметра.

MyFunc(100,200,300,400,0);//В функцию передается 4 параметра и пятый будет использован как идентификатор того, что параметры кончились

MyFunc(4,100,200,300,400);//В функцию передается 5 параметров и первыйбудет использован как идентификатор того, сколько всего параметров

Проход при помощи указателя

void MyFunc(int a,...) //Так объявляется функция с заранее неизвестным числом параметров

{

  int \*P=&a; //Взятие адреса у первого параметра

  while (\*P) //Пока встречаются параметры и параметр не равен 0

   {

     cout<<\*P<<" ";//Вытаскиваем значение с адреса по которому живет параметр

     P++; //Адресная арифметика. Смена текущего адреса на следующий

   }

}

Проход при помощи макросов

#include<stdio.h>

#include <stdarg.h>

void sum(char \*msg, ...) {

//считает сумму любого числа параметров и выводит ее с сообщением msg

//число параметров это функции переменно, концом списк служит значение 0

int total = 0;

va\_list ap; //тип для хранения информации, передаваемой в va\_start, va\_arg, va\_end

int arg; //очередной целочисленный аргумент

va\_start(ap, msg); //инициализирует список параметров

while ((arg = va\_arg(ap,int)) != 0) { //извлекает следующий параметр типа int

total += arg; //накопление суммы

}

printf(msg, total);

va\_end(ap); //очищаем и закрываем список параметров

}

void main(void) {

sum("Сумма 1=%d\n",1,2,3,4,0);

sum("Сумма 2=%d\n",10,20,0);

}

**21. Функции. Указатели на функции. Массив указателей на функции.**

Хотя функция - это не переменная, она все равно имеет физическое местоположение в памяти, которое может быть присвоено указателю. Адрес функции является входной точкой функции. Поэтому указатель на функцию может использоваться для вызова функции.

Для того, чтобы понять, как можно использовать массив указателей на функции вышеприведенным способом, представим, что необходимо создать простейшую систему инвентаризации, позволяющую вводить, удалять и просматривать данные, а также выходить в операционную систему. Если функции, выполняющие эти действия, называются соответственно enter(), del(), review() и quit(), то следующий фрагмент кода корректно инициализирует массив указателей на функции:

void enter(void), del(void), review(void), quit(void); int menu(void);

void (\*options[])(void) = {

enter,

del,

review,

quit};

Использование массивов указателей на функции типично не только для компиляторов и интерпретаторов, но и для программ, работающих с базами данных, поскольку такие программы часто предоставляют много вариантов и в них важна эффективность.

В программе на С адресом функции служит ее имя без скобок и аргументов (это похоже на адрес массива, который равен имени массива без индексов).

int (\*p)(const char \*, const char \*);

Это объявление сообщает компилятору, что p — это указатель на функцию, имеющую два параметра типа const char \* и возвращающую значение типа int. Скобки вокруг p необходимы для правильной интерпретации объявления компилятором. Подобная форма объявления используется также для указателей на любые другие функции, нужно лишь внести изменения в зависимости от возвращаемого типа и параметров функции.

Теперь рассмотрим функцию check(). В ней объявлены три параметра: два указателя на символьный тип (a и b) и указатель на функцию cmp. Обратите внимание на то, что указатель функции cmp объявлен в том же формате, что и p. Поэтому в cmp можно хранить значение указателя на функцию, имеющую два параметра типа const char \* и возвращающую значение int. Как и в объявлении p, круглые скобки вокруг \*cmp необходимы для правильной интерпретации этого объявления компилятором.

Вначале в программе указателю p присваивается адрес стандартной библиотечной функции strcmp(), которая сравнивает строки. Потом программа просит пользователя ввести две строки и передает указатели на них функции check(), которая их сравнивает. Внутри check() выражение (\*cmp)(a, b) вызывает функцию strcmp(), на которую указывает cmp, с аргументами a и b. Скобки вокруг \*cmp обязательны. Существует и другой, более простой, способ вызова функции с помощью указателя:

Массив указателей на функции можно использовать в разных целях. Один из вариантов такого использования – это если, к примеру, у вас в коде есть одна куча разных функций, но схожих по смыслу и другая куча разных функций, не похожих на первые функции, но похожих друг на друга. Вот можно похожие функции собрать в массив и потом уже вызывать из массива. Не возьмусь утверждать, но скорее всего так будет проще не запутаться. Я не буду приводить пример с десятками функций, а приведу один из простейших.

using namespace std;

void a() {cout<<"A";} //функцияа

void b() {cout<<"B";} //функция b

void c() {cout<<"C";} //функция с

int main(){

void (\*P[3])()={a,b,c}; //Массив из 3-х указателей на функции

(\*P[2])(); //Обращение к элементу массива вызывает функцию

  cin.get();

 return 0;

**22. Параметры функции main()**

Иногда при запуске программы бывает полезно передать ей какую-либо информацию. Обычно такая информация передается функции main() с помощью аргументов командной строки. Аргумент командной строки — это информация, которая вводится в командной строке операционной системы вслед за именем программы. Например, чтобы запустить компиляцию программы, необходимо в командной строке после подсказки набрать примерно следующее:

cc имя\_программы

имя\_программы представляет собой аргумент командной строки, он указывает имя той программы, которую вы собираетесь компилировать.

Чтобы принять аргументы командной строки, используются два специальных встроенных аргумента: argc и argv. Параметр argc содержит количество аргументов в командной строке и является целым числом, причем он всегда не меньше 1, потому что первым аргументом считается имя программы. А параметр argv является указателем на массив указателей на строки. В этом массиве каждый элемент указывает на какой-либо аргумент командной строки. Все аргументы командной строки являются строковыми, поэтому преобразование каких бы то ни было чисел в нужный двоичный формат должно быть предусмотрено в программе при ее разработке. Очень важно правильно объявлять argv. Вот как это делают чаще всего: char \*argv[]; Пустые квадратные скобки указывают на то, что у массива неопределенная длина. Теперь получить доступ к отдельным аргументам можно с помощью индексации массива argv. Например, argv[0] указывает на первую символьную строку, которой всегда является имя программы; argv[1] указывает на первый аргумент и так далее.

Имена argc и argv являются традиционными, но не обязательными. Эти два параметра в функции main() вы можете назвать как угодно. Кроме того, в некоторых компиляторах для main() могут поддерживаться-дополнительные аргументы,

**23. Функции. Рекурсивные функции. Распределение памяти при рекурсивном вызове. Типы рекурсии. Рекомендации по использованию рекурсивного вызова.**

ВС функции могут вызывать сами себя. Функция является рекурсивной, если оператор в теле функции вызывает функцию, содержащую данный оператор. Иногда называемая круговым определением, рекурсия является процессом определения чего-либо с использованием самой себя.

Когда функция вызывает сама себя, в стеке выделяется место для новых локальных переменных и параметров. Код функции работает с данными переменными. Рекурсивный вызов не создает новую копию функции. Новыми являются только аргументы. Поскольку каждая рекурсивно вызванная функция завершает работу, то старые локальные переменные и параметры удаляются из стека и выполнение продолжается с точки, в которой было обращение внутри этой же функции. Рекурсивные функции вкладываются одна в другую как элементы подзорной трубы.

Рекурсивные версии большинства подпрограмм могут выполняться немного медленнее, чем их итеративные эквиваленты, поскольку к необходимым действиям добавляются вызовы функций. Но в большинстве случаев это не имеет значения. Много рекурсивных вызовов в функции может привести к переполнению стека. Поскольку местом для хранения параметров и локальных переменных функции является стек и каждый новый вызов создает новую копию переменных, пространство стека может исчерпаться. Если это произойдет, то возникнет ошибка - переполнение стека.

Основным преимуществом применения рекурсивных функций является использование их для более простого создания версии некоторых алгоритмов по сравнению с итеративными эквивалентами. Например, сортирующий алгоритм Quicksort достаточно трудно реализовать итеративным способом. Некоторые проблемы, особенно связанные с искусственным интеллектом, также используют рекурсивные алгоритмы. Наконец, некоторым людям кажется, что думать рекурсивно гораздо легче, чем итеративно.

При написании рекурсивных функций следует иметь оператор if, чтобы заставить функцию вернуться без рекурсивного вызова. Если это не сделать, то, однажды вызвав функцию, выйти из нее будет невозможно. Это наиболее типичная ошибка, связанная с написанием рекурсивных функций. Надо использовать при разработке функции printf() и getchar(), чтобы можно было узнать, что происходит, и прекратить выполнение в случае обнаружения ошибки.

(другой источкник )Рекурсию можно назвать простой, если в функции присутствует лишь один рекурсивный вызов. Такую рекурсию можно назвать еще рекурсией первого порядка. Но рекурсивный вызов может появляться в функции более, чем один раз. В таких случаях можно выделить следующие виды рекурсии:

1) параллельная рекурсия - тело определения функции function\_1 содержит вызов некоторой функции function\_2, несколько аргументов которой являются рекурсивными вызовами функции function\_1.

(defun function\_1 … (function\_2 … (function\_1 …) … (function\_1 …) … ) … )

2) взаимная рекурсия - в теле определения функции function\_1 вызывается некоторая функции function\_2, которая, в свою очередь, содержит вызов функции function\_1.

(defun function\_1 … (function\_2 … ) … ) (defun function\_2 … (function\_1 … ) … )

3) рекурсия более высокого порядка - в теле определения функции аргументом рекурсивного вызова является рекурсивный вызов.

(defun function\_1 … (function\_1 … (function\_1 …) … ) … )

Типы рекурсии: линейная,повторительная,каскадная(древовидная), косвенная(взаимная), удаленная.

Линейная рекурсия- Рекурсия, при которой рекурсивные вызовы на любом рекурсивном срезе, инициируют не более одного последующего рекурсивного вызова, называется линейной. Это наиболее простой и часто встречающийся тип рекурсии. Большая часть ранее разобранных примеров относилась именно к этому типу рекурсии

Повторительная рекурсия -Частный случай линейной рекурсии с отсутствующими предварительными или отложенными вычислениями называется повторительной рекурсией. Например для целого неотрицательного числа *n* подсчитывает и возвращает количество цифр “1” в двоичном представлении *n*.

Древовидная рекурсия-Этот тип рекурсии определяется как дополнительный к линейному. Иными словами, рекурсию называют каскадной, если она не является линейной.При каскадной рекурсии, рекурсивные обращения, как правило, приводят к необходимости многократно решать одни и те же подзадачи. И поэтому, по возможности, от неё или следует избавляться или предпринимать шаги, освобождающие от необходимости производить повторные решения подзадач. Пример- нахождение числа фибоначи

Косвенная рекурсия-Циклическая последовательность вызовов нескольких функций *F*1, *F*2, …, *Fk* (процедур, алгоритмов) друг друга: *F*1 вызывает *F*2, *F*2 вызывает *F*3, …, *Fk* вызывает *F*1 (*k*>1), называют косвенной или взаимной рекурсией. Пример-Генерация разбиения множества

Удалённая рекурсия-Рекурсия называется удаленной, если в теле функции при рекурсивных вызовах, в выражениях, являющихся фактическими параметрами, снова встречаются рекурсивные вызовы этой функции. следует избегать рекурсий там, где есть очевидное итерационное решение.

Использовать рекурсию удобно, например, в тех случаях, когда отказ от нее заставил бы программиста самого организовывать стек для хранения промежуточных результатов. Рекурсия весьма эффективна при работе с графами. Ее применение часто позволяет давать более прозрачные и сжатые описания алгоритмов, чем это было бы возможно без рекурсии.

В качестве *примера* удачного применения рекурсии рассмотрим процедуру *обхода двоичного дерева*.

**24. Динамические структуры данных. Рекомендации для использования. Основные особенности.**

Если до начала работы с данными невозможно определить, сколько памяти потребуется для их хранения, *память* следует распределять во *время выполнения* программы *по* мере необходимости отдельными блоками. Блоки связываются друг с другом с помощью указателей. Такой способ организации данных называется динамической структурой данных, поскольку она размещается в динамической памяти и ее размер изменяется во *время выполнения* программы.

Из динамических структур в программах чаще всего используются линейные списки, стеки, очереди и бинарные деревья. Они различаются способами связи отдельных элементов и допустимыми операциями. *Динамическая* структура, в отличие от массива или записи, может занимать несмежные участки оперативной памяти.

Элемент динамической структуры состоит из двух частей: информационной, ради хранения которой и создается структура, иуказателей, обеспечивающих *связь* элементов друг с другом.

В качестве информационной части может быть данное любого типа: целого, вещественного, символьного, строкового, структуры и т.п.  
Элемент списка на С++ может быть описан так:

struct node{

int a; // информационное поле

struct node\* next;};// указатель на следующий элемент

Линейный список — это динамическая структура данных, каждый элемент которой посредством указателя связывается со следующим элементом.Список можно сформироватьспособами: 1- по принципу стека: первый созданный элемент будет последним в списке, 2- по принципу очереди: первый созданный элемент будет первым в списке.

Стек – это такой линейный список, в котором и добавление новых и удаление существующих элементов возможно только с головного элемента. Функции для работы:Push - Добавить (положить) в конец(т.е.голову) стека новый элемент

Pop - Извлечь из стека последний(т.е. вершину) элемент

по принципу стека:  
1. Пусть указатель на ранее созданный элемент pred=NULL (т.е. до формирования списка нет такого элемента). 2. Выделить память под текущий элемент списка p.

3. Занести значение в информационное поле. 4. Занести значение в поле next равное pred. 5. Сохранить в pred значение p текущего элемента. 6. Повторять п. 2 – 5, пока не будет сформировано нужное количество элементов списка. 7. В p будет указатель на первый элемент списка.

Очередь — это динамическая структура данных, добавление элементов в которую выполняется в один конец, а удаление — из другого конца, который называется концом очереди.

Формирование списка по принципу очередь

1. Задать начальное значение указателя на первый элемент списка равный NULL (т.е. до формирования списка нет такого элемента).  
2. Если указатель на первый элемент списка равен NULL, выделить память под элемент списка p и сохранить адрес в head. 3. Занести значение в информационное поле. 4. Занести значение в поле next равное NULL.  
5. Сохранить в pred значение p текущего элемента.  
6. Повторять п. 2 – 5, пока не будет сформировано нужное количество элементов списка.  
7. В head будет указатель на первый элемент списка.

**25. Стек. Создание стека. Общие принципы работы**

Стек – это упорядоченный набор элементов, в котором добавление новых и удаление существующих элементов допустимо только с одного конца, который называется вершиной стека.  
Стек может быть реализован на односвязном списке. В этом случае каждый элемент стека хранит значение и указатель на следующий элемент. На вершину стека будет ссылаться указатель Node \*top:   
  
struct Node {   
int val;   
Node [\*next](https://vk.com/club6386);   
Node(int v = 0, Node \*n = NULL) {   
val = v;   
next = n;   
}   
} \*top;  
Принцип работы стека подразумевает абстрактное понятие вершины стека — места, в которое происходит добавление элементов и из которого происходит удаление элементов.

**26. Стек. Удаление элемента. Добавление элемента. Общие принципы работы**

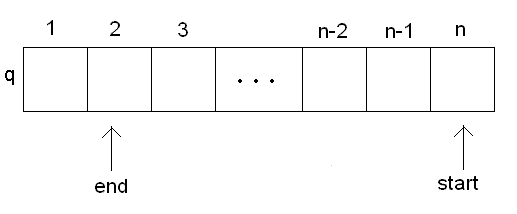
См 27 билет  
**Вставка элемента**   
Для вставки элемента в стек требуется выполнить следующие действия:   
Фактически элемент добавляется в список, поэтому прежде всего нужно выделить память под новый экземпляр Node и задать в нём нужное значение val;   
Вставка элемента происходит в вершину стека, поэтому следующим элементом для вставляемого будет тот, на который в данный момент указывает top;   
Наконец, вершина стека теперь должна указывать на добавленный элемент.   
Благодаря наличию конструктора Node(v, n) все вышеописанные операции можно выполнить одной строкой:   
void push(int value) {   
top = new Node(value, top);   
}   
**Извлечение элемента**    
Заметим, что извлечение элемента возможно только в том случае, когда стек не является пустым. Попытку извлечь элемент из пустого стека нужно должным образом обрабатывать (возбуждать исключение, возвращать специальное значение и т. д.).   
Удаление элемента из стека подразумевает следующую последовательность операций:   
Так как необходимо вернуть извлечённое значение, но также и освободить память, отведённую под соответствующий элемент списка, требуется сохранить значение top->val во временной переменой для возможности последующего возврата;   
Указатель на вершину стека нужно сместить на следующий элемент, но перед этим нужно сохранить во временной переменной предыдущий адрес вершины, чтобы можно было освободить память;   
Далее можно освободить память и вернуть сохранённое ранее значение.   
int pop() {   
if (top == NULL)   
/\* обработка ошибки - нет элементов для извлечения \*/   
int v = top->val;   
Node \*tmp = top;   
top = top->next;   
delete tmp;   
return v;   
}

**27. Очередь. Создание очереди.**

Очередь -*упорядоченный набор элементов*, в котором извлечение элементов происходит с одного его конца, а добавление новых элементов - с другого.Добавление элемента (принято обозначать словом enqueue — поставить в очередь) возможно лишь в конец очереди, выборка — только из начала очереди (что принято называть словом dequeue — убрать из очереди), при этом выбранный элемент из очереди удаляется.(это структуры FIFO (First - In - First- Out - "первым пришел - первым исключается"))

**Способы реализации очереди**

Существует несколько способов реализации очереди в языках программирования.

Первый способ представляет очередь в виде [массива](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) и двух целочисленных [переменных](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) start и end.  
[](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Queue1.gif)  
Обычно start указывает на голову очереди, end — на элемент, который заполнится, когда в очередь войдёт новый элемент. При добавлении элемента в очередь в q[end] записывается новый элемент очереди, а end уменьшается на единицу. Если значение end становится меньше 1, то мы как бы циклически обходим массив, и значение переменной становится равным n. Извлечение элемента из очереди производится аналогично: после извлечения элемента q[start] из очереди переменная start уменьшается на 1. С такими алгоритмами одна ячейка из n всегда будет незанятой (так как очередь с n элементами невозможно отличить отпустой), что компенсируется простотой алгоритмов.

Преимущества данного метода: возможна незначительная экономия памяти по сравнению со вторым способом; проще в разработке. Недостатки: максимальное количество элементов в очереди ограничено размером массива.

Связный список

Второй способ основан на работе с динамической памятью. Очередь представляется в качестве [линейного списка](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA), в котором добавление/удаление элементов идет строго с соответствующих его концов.

Преимущества данного метода: размер очереди ограничен лишь объёмом памяти.

Недостатки: сложнее в разработке; требуется больше памяти; при работе с такой очередью память сильнее фрагментируется; работа с очередью несколько медленнее.

**Пример**

typedef struct

{int qh, qt;

int val[MAXN];

} queue;

void enq(queue \*q, int x)

{q->val[(q->qt++)%MAXN] = x;}   *//добавить эелемент в хвост*

int deq(queue \*q)

{return q->val[(q->qh++)%MAXN];}   *//извлечь из головы*

**28. Очередь. Удаление элемента. Добавление элемента. Общие принципы работы**

Очередь–линейный абстрактный тип данных, доступ к элементам которого организован по принципу FIFO(от англ. first-in-first-out).

В соответствие с принципом FIFO для очереди должны быть определены две точки входа: началоочереди, откуда происходит удаление элементов, и ко-нецочереди, куда происходитдобавление элементов. В частном случае очередь можно рассматривать как список, вставка элементов в который происходит на одном конце, а удаление элементов–на другом.

#define QMAX 100  
struct queue {  
  int qu[QMAX];  
  int rear, frnt;  
};

void insert(struct queue \*q, int x) {  
  if(q->rear < QMAX-1) {  
    q->rear++;  
    q->qu[q->rear]=x;  
  }  
  else  
    printf("Очередь полна!\n");  
  return;  
}

int remove(struct queue \*q) {  
  int x;  
  if(isempty(q)==1) {  
    printf("Очередь пуста!\n");  
    return(0);  
  }  
  x = q->qu[q->frnt];  
  q->frnt++;  
  return x;  
}

**29. Линейные списки. Создание элемента списка.**

Линейный список — это динамическая структура данных, каждый элемент которой посредством указателя связывается со следующим элементом.

Из определения следует, что каждый элемент списка содержит поле данных (Data) (оно может иметь сложную структуру) и поле ссылки на следующий элемент (next). Поле ссылки последнего элемента должно содержать пустой указатель (NULL).

Так как ссылка всего одна (только на следующий элемент), то такой список является односвязным.

Когда говорят о линейном списке, то, как правило, подразумевают именно односвязный список.

structList

{

   Intd;

   List \*next;

};

ListCreateList(int x){

   List \*Head = new List;

Head->next=NULL;

Head->d=x;

ReturnHead;

}

**30. Линейные списки. Добавление узла в начало списка.**

См 31 билет

struct List

{

   Int d;

   List \*next;

};

Void newelem(List\*Head,int x){

   List \*p = new List;

   p->d=x;

   p->next = Head;

   Head=p;}

**31. Линейные списки. Поиск элемента.**

Линейный список — это динамическая структура данных, каждый элемент которой посредством указателя связывается со следующим элементом. Из определения следует, что каждый элемент списка содержит поле данных (Data) (оно может иметь сложную структуру) и поле ссылки на следующий элемент (next).

struct List

{

   Int d;

   List \*next;

};

bool findelem(List\*Head,int x){

   List \*p = new List;

p=head;

While(p!=Null){

If(p->d == x){cout<<”Есть элемент”;return true;}

p=p->next;

}

Cout<<”Элемента нет”;return false;

}

**32. Линейные списки. Удаление узла.**

См 31 билет

struct List

{

   Int d;

   List \*next;

};

Void delelem(List\*Head,intx){//удалениеузласозначениемx

   List \*p = new List;

P=Head;

List\* Pred,\*Next;

While(p->d!=x){

Pred=p;

P=p->Next;

}

List \*a=new List;

A=p;

Next=a->next;

Pred->next=Next;

Delete a;

}

**33. Двусвязный список. Создание. Удаление узла.**

См 34 билет

Чтение и удаление элемента с адресом sp

int ReadOchdD(tochd \*sp)

{

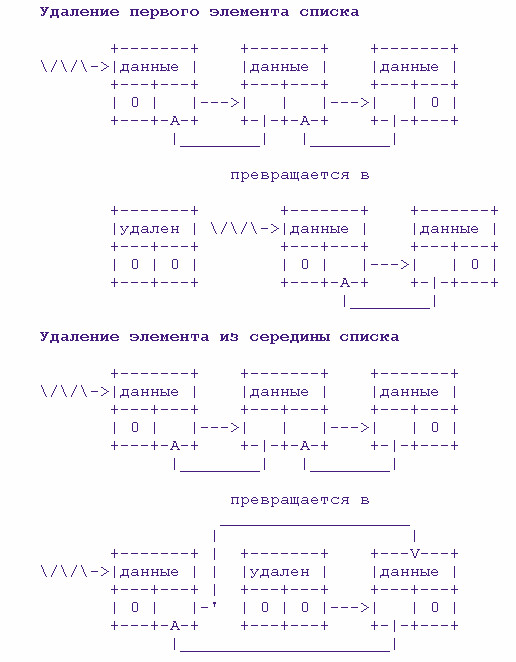
int inf= sp->inf;

sp->left->rigth = sp->rigth;

sp->rigth->left = sp->left;

delete sp;

return inf; }

****

**34. Двусвязный список. Создание. Добавление узла.**

Двусвязанный список состоит из элементов, содержащих ссылки как на предыдущий, так и на последующий элементы. Такая организация позволяет осуществлять перемещение по списку в любом направлении. Это упрощает работу со списком, в частности, вставку и удаление. Также повышается устойчивость работы, т. к. можно восстановить потерянную ссылку. Недостатком является введение дополнительного указателя, что несколько усложняет программу.

Объявление рекурсивного типа – двусвязанная очередь:

struct tochd

{

int inf;

tochd \*left;

tochd \*rigth;

} \*sp;

**Создание очереди**

void NewOchd(tochd \*\*sl, tochd \*\*sr)

{

\*sl=new tochd;

\*sr=new tochd;

(\*sl)->left = NULL;

(\*sl)->rigth = \*sr;

(\*sr)->left = \*sl;

(\*sr)->rigth = NULL;

return;

}

**Добавление элемента после заданного**

void AddOchdRigth(tochd \*sp, int inf)

{

tochd \*spt=new tochd;

spt->inf = inf;

spt->left = sp;

spt->rigth = sp->rigth;

sp->rigth = spt;

spt->rigth->left = spt;

return;

}

**Добавление элемента перед заданным**

void AddOchdLeft(tochd \*sp, int inf)

{

tochd \*spt=new tochd;

spt->inf = inf;

spt->left = sp->left;

spt->rigth = sp;

spt->left->rigth = spt;

sp->left = spt;

return;

}

**35. Деревья. Общие понятия.**

Дерево — одна из наиболее широко распространённых [структур данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) в [информатике](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0), эмулирующая [древовидную структуру](https://ru.wikipedia.org/wiki/%D0%94%D1%80%D0%B5%D0%B2%D0%BE%D0%B2%D0%B8%D0%B4%D0%BD%D0%B0%D1%8F_%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0) в виде набора связанных узлов.

*Корневой узел* — самый верхний узел дерева (узел 2 на примере).

*Корень* — одна из вершин, по желанию наблюдателя.

*лист*, *листовой* или *терминальныйузел* — узел, не имеющий дочерних элементов

*Внутренний узел* — любой *узел* дерева, имеющий *потомков*, и таким образом, не являющийся *листовым узлом*

*Поддерево* — часть древообразной структуры данных, которая может быть представлена в виде отдельного дерева. Любой узел дерева *T* вместе со всеми его узлами-потомками является поддеревом дерева *T*.

Дерево считается *ориентированным*, если в корень не заходит ни одно ребро.

Объявление бинарного дерева:

structttree

{

int inf;

ttree \*left;

ttree \*rigth;

} \*proot

**37. Бинарные деревья. Обход дерева в ширину.**

Обход в ширину производится с помощью очереди. Первоначально в очередь помещается корень, затем, пока очередь не пуста, выполняются следующие действия:

* Из очереди выталкивается очередной узел;
* Этот узел обрабатывается;
* В очередь добавляются оба сына этого узла.

Узлы дерева на рисунке перечисляются в порядке обхода в ширину следующим образом: A, B, C, D, E, F, G, H, I, J. Заметим, что перечисление узлов происходит в порядке удаления от корня, что делает поиск в ширину удобным, например, для поиска узла дерева со значением k, наиболее близкого к корню, и т.д.

void width(t \*root)  
{    if (!root)  
        return;  
    add(root);  
    while (!empty()) {  
        t \*curr = del();  
        printf("%d ", curr->data);  
        if (curr->left)  
            add(curr->left);  
        if (curr->right)  
            add(curr->right);}}

**36. Бинарные деревья**

Двои́чное де́рево — иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками. Двоичное дерево является упорядоченным ориентированным деревом.[1]

Для практических целей обычно используют два подвида двоичных деревьев — двоичное дерево поиска и двоичная куча.

Деревом называют структуру данных, которая имеет древовидный вид, то есть характеризуется наличием набора

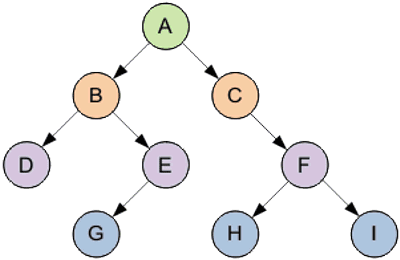
связанных узлов. Бинарное дерево — это конечное множество элементов, связанных с двумя разными бинарными деревьями — правым и левым поддеревьями.

Способ представления:

На что следует обратить внимание: — А — корень дерева; — B — корень левого поддерева и предок D; — С — корень правого поддерева; — D — потомок родительского узла B; если D располагается на уровне i, то B – на уровне i-1.

Необходимо выделить следующие термины: — узел (вершина) — это каждый элемент бинарного дерева; — ветви — связи между узлами; — глубина (высота) — наибольший уровень какого-нибудь элемента; — лист (терминальный узел) — термин применяется, если элемент не имеет потомков; — внутренние узлы — узлы ветвления; — степень внутреннего узла — число его потомков (наибольшая степень всех существующих узлов — это степень всего бинарного дерева); — длина пути к x — количество ветвей, которые необходимо пройти от корня до текущего узла x. Длина пути самого корня равна нулю, а узел на уровне i обладает длиной пути, которая равна i.

Использование



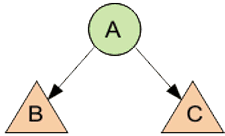
На практике бинарные деревья применяются, когда в каждой точке какого-нибудь вычислительного процесса нужно принимать одно из 2-х возможных решений. Существует множество задач, решаемых таким способом. Одна из них — выполнение операции, условно говоря, X с каждым элементом дерева. X рассматривается в качестве параметра обобщенной задачи посещения всех вершин либо задачи обхода дерева. Если рассмотреть такую задачу в качестве единого последовательного процесса, то можно сказать, что отдельные вершины посещаются в определенном порядке, то есть могут считаться линейно расположенными.

**37. Обход бинарного дерева.**

**Способы обхода**

Предположим, что имеется дерево (не пустое), в котором A является корнем, а B и C — левым и правым поддеревьями.

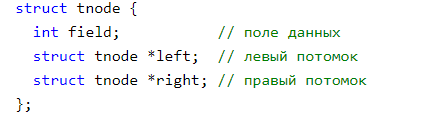
Есть 3 способа обхода: 1. Обход в прямом порядке — сверху вниз: A, B, C — префиксная форма. 2. Обход слева направо (симметричный порядок): B, A, C — инфиксная форма. 3. Обход снизу вверх (обратный порядок): B, C, A — постфиксная форма.

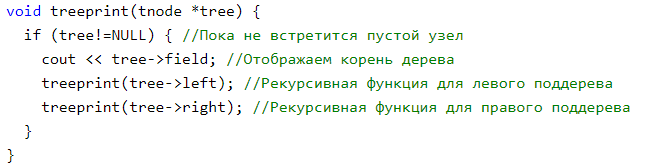


Реализация

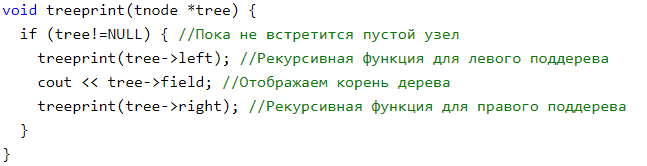
На практике вершину древа можно описать в качестве структуры:

Тогда обход в префиксной форме будет выглядеть следующим образом:

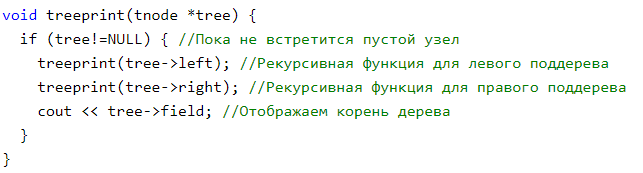




Теперь инфиксная форма:



И постфиксная:



**38. Бинарные деревья. Поиск в дереве.**

Бинарное дерево поиска обладает следующим свойством: если x— узел бинарного дерева с ключом k, то все узлы в левом поддереве должны иметь ключи, меньшие k, а в правом поддереве большие k.

Поиск с заданным ключом:

void poisktree(ttree \*p,int key, bool &b, int &inf)

{

if ((p != NULL) && (b != true))

{

if (p->inf !=key)

{

poisktree(p->left, key,b,inf);

poisktree(p->rigth,key,b,inf);

}

else

{

b=true;

inf=p->inf;

}

}

return;

}

Поиск элемента с максимальным ключом

int poiskmaxtree(ttree \*p)

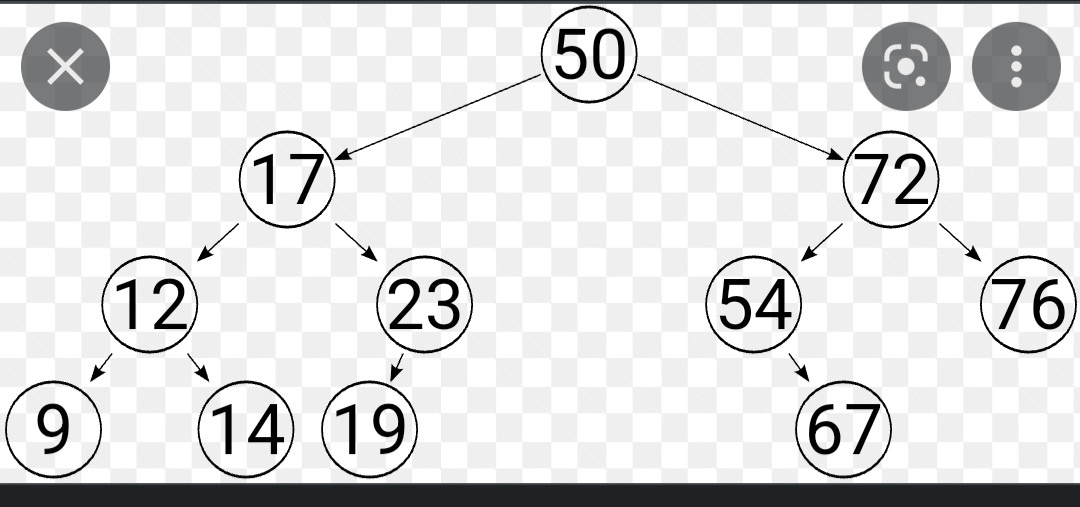
{

while (p->rigth != NULL) p = p->rigth;

return p->inf;

}

**39. Сбалансированные деревья**

Дерево называется сбалансированным тогда и только тогда, когда для каждого его узла высоты его левого и правого поддеревьев отличаются не более чем на единицу. Для каждого узла дерева можно определить показатель сбалансированности как разность между высотой правого и левого поддерева данного узла.

**40. Файлы. Типы файлов.**

Бинарный,текстовый

Те́кстовый файл — компьютерный [файл](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB), содержащий [текстовые данные](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5), как правило, организованные в виде [строк](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%BE%D0%B2%D1%8B%D0%B9_%D1%82%D0%B8%D0%BF).

Текстовым файлам противопоставляются [двоичные файлы](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB), в которых содержатся данные, не рассчитанные на интерпретацию в качестве [текста](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82) (например, файлы, хранящие закодированные звук или изображение).

Двоичный (бинарный) файл — в широком смысле: последовательность произвольных [байтов](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82). Название связано с тем, что байты состоят из [бит](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82), то есть двоичных цифр.

**41. Доступ к файлам.**

Текстовые    файлы не  являются   файлами  прямого  доступа. Это означает, что во  время   работы  с  текстовыми  файлами   допускается   только  один  вид  операции: либо   чтение, либо  запись.

Вотличие от текстовых файлов доступ к элементам бинарных файлов выполняется в произвольном порядке, а не последовательно.

Для получения доступа к бинарному файлу(потоку) нужно:

1. Создать поток соответствующего типа:

*ifstream*- для ввода из файла ;

*ofstream*- для вывода в файл;

*fstream*   - для обмена с файлом в двух направлениях.

1. Связать его с файлом данных и открыть (open) для работы в определенном режиме,

с обязательным указанием двоичного режима *ios::binary*(по умолчанию потоки открываются в текстовом режиме):

*void ifstream::open(const char \*имя\_файла, openmode режим=ios::in|ios::binary);*

*void ofstream::open(const char \*имя\_файла,openmode режим=ios::out|ios::trunс|ios::binary);*

*void fstream::open(const char \*имя\_файла,openmode режим=ios::in|ios::out|ios::binary);*

где имя\_файла – имя файла, в которое может входить спецификатор пути; режим – задает режим открытия файла

1. Обмен данными с файлом через поток: запись в поток; чтение из потока; управление состоянием потока. Для записи в поток в C++ используется метод write:*write( const char\_type \*\_Str, streamsize \_Count );*

Для чтения из потока используется метод read: *read( const char\_type \*\_Str, streamsize \_Count )*

язык Си не имеет прямой поддержки [произвольного доступа](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF) к файлам данных; чтобы считать записанную информацию в середине файла, программисту приходится создавать поток, [ищущий в середине файла](https://ru.wikipedia.org/w/index.php?title=Fseek&action=edit&redlink=1), а затем последовательно считывать байты из потока.

**42. Файлы. Текстовые файлы. Открытие \ закрытие.**

В зависимости от способа описания, выделяют текстовые и двоичные. Вид файла определяет способ хранения информации в файле. Текстовый файл представляет собой набор строк произвольной длины, каждая строка заканчивается управляющим символом ‘\n’. Текстовый файл является файлом последовательного доступа. Логически текстовый файл можно представить как именованную последовательность байтов, имеющую начало и конец. Чтение из файла или запись в файл ведется байт за байтом от начала к концу.

Для того чтобы чтобы открыть файл на запись или чтение надо использовать схему: open->work->close. Работа с потоком начинается с открытия, поток можно открыть для чтения и лоя записи, в двоичном или в текстовом режиме.

Для работы с файлом используется указатель. Это адрес блока данных в памяти в которой хранится вся информация файла.

Функция открытия потока:

FILE \*fp=fopen(“имя файла”,”режим открытия”);

Функция закрытия потока:

Fclose(fp);

**43. Текстовые файлы. Ввод \ вывод в текстовые файлы.**

Файлы позволяют пользователю считывать большие объемы данных непосредственно с диска, не вводя их с клавиатуры. Существуют два основных типа файлов: **текстовые и двоичные**.

*Текстовыми* называются файлы, состоящие из любых символов. Они организуются по строкам, каждая из которых заканчивается символом «*конца строки»*. Конец самого файла обозначается символом «*конца файла»*. При записи информации в текстовый файл, просмотреть который можно с помощью любого текстового редактора, все данные преобразуются к символьному типу и хранятся в символьном виде.

В *двоичных* файлах информация считывается и записывается в виде блоков определенного размера, в которых могут храниться данные любого вида и структуры.

Для работы с файлами используются специальные типы данных, называемые *потоками.*Поток **ifstream**служит для работы с файлами в режиме чтения, а **ofstream** в режиме записи. Для работы с файлами в режиме как записи, так и чтения служит поток **fstream**.

В программах на C++ при работе с текстовыми файлами необходимо подключать библиотеки **iostream** и**fstream**.

Для того чтобы записывать данные в текстовый файл, необходимо:

* описать переменную типа **ofstream**.
* открыть файл с помощью функции **open**.
* вывести информацию в файл.
* обязательно закрыть файл.

Для считывания данных из текстового файла, необходимо:

* описать переменную типа **ifstream**.
* открыть файл с помощью функции **open**.
* считать информацию из файла, при считывании каждой порции данных необходимо проверять, достигнут ли конец файла.
* закрыть файл.

**45. Бинарные файлы. Функции чтения \ записи .**

# Функции fread() и fwrite()

Для чтения и записи данных, тип которых может занимать более 1 байта, в файловой системе языка С имеется две функции**: fread() и fwrite().** Эти функции позволяют читать и записывать блоки данных любого типа. Их прототипы следующие:

**size\_t fread(void \**буфер*, size\_t *колич\_байт*, size\_t *счетчик*, FILE \**уф*);**

**size\_t fwrite(const void \**буфер*, size\_t *колич\_байт*, size\_t *счетчик*, FILE \**уф*);**

Для **fread()***буфер* — это указатель на область памяти, в которую будут прочитаны данные из файла. А для **fwrite()***буфер* — это указатель на данные, которые будут записаны в файл. Значение *счетчик* определяет, сколько считывается или записывается элементов данных, причем длина каждого элемента в байтах равна *колич\_байт*. (Вспомните, что тип size\_t определяется как одна из разновидностей целого типа без знака.) И, наконец, *уф* — это указатель файла, то есть на уже открытый поток.

Функция fread() возвращает количество прочитанных элементов. Если достигнут конец файла или произошла ошибка, то возвращаемое значение может быть меньше, чем счетчик. А функция fwrite() возвращает количество записанных элементов. Если ошибка не произошла, то возвращаемый результат будет равен значению счетчик.

Одним из самых полезных применений функций fread() и fwrite() является чтение и запись данных пользовательских типов, особенно структур. Например, еслиопределенаструктура

struct struct\_type {

float balance;

charname[80];

} cust;

то следующий оператор записывает содержимое cust в файл, на который указывает fp:

fwrite(&cust, sizeof(struct struct\_type), 1, fp);

**44. Бинарные файлы. Открытие \ закрытие файла. Диагностика ошибок.**

**Двоичный (бинарный) файл** — в широком смысле: последовательность произвольных [байтов](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82). Название связано с тем, что байты состоят из [бит](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82), то есть двоичных цифр.В узком смысле слова двоичные файлы противопоставляются [текстовым](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB) файлам. При этом с точки зрения технической реализации на уровне аппаратуры, текстовые файлы являются частным случаем двоичных файлов, и, таким образом, в широком значении слова под определение «двоичный файл» подходит любой файл.

### Прототип функции fopen:

|  |  |
| --- | --- |
|  | FILE\* fopen( constchar\* fname, constchar\* modeopen ); |

Функция fopen открывает файл, имя которого указано в параметре fname и связывает его с потоком, который может быть идентифицирован для выполнения различных операций с файлом. Операции с потоком, выполнение которых разрешено определяются параметром modeopen.

**Функция fclose()** используется для закрытия потока, ранее открытого с помощью fopen(). Она сохраняет в файл данные, находящиеся в дисковом буфере, и выполняет операцию системного уровня по закрытию файла. Вызов fclose() освобождает блок управления файлом, связанный с потоком, и делает его доступным для повторного использования.

Функция fclose() имеет прототип: int fclose(FILE \*fp);  
где fp - это указатель на файл, возвращенный fopen(). Если возвращен 0, то это означает, что операция закрытия выполнена успешно, а если EOF, то, значит, была ошибка.

**45. Бинарные файлы. Позиционирование в файле.**

Двоичный файл отличается от текстового тем, что данные в нем представлены во внутренней форме. А поскольку при внутреннем представлении используется двоичная система счисления, то «в честь ее» файлы и называются двоичными. По существу, двоичный файл является аналогом внутренней (оперативной, физической) памяти – неограниченным массивом байтов с возможностью непосредственного обращения (произвольного доступа) к любой его части.

int fread (void \*buf, int size, int nrec, FILE \*fd);

int fwrite (void \*buf, int size, int nrec, FILE \*fd);

Функци **fread**читает, а функция **fwrite** пишет в файл, начиная с текущей позиции, массив из **nrec** элементов размерностью **size** байтов каждый, возвращая количество успешно прочитанных (записанных) элементов.

С каждым открытым файлом связывается такой параметр как **текущая позиция**(текущий адрес) - номер байта, начиная с которого будет выполняться очередная операция чтения-записи.

Функция **long ftell(FILE \*fp)**возвращает текущую позицию в файле. Если по каким-то причинам текущая позиция не определена, функция возвращает **-1.**

Функция **int    fseek(FILE \*fp, long pos, int mode)** устанавливает текущую позицию в файле на байт с номером **pos.** Параметр **mode**определяет, относительно чего отсчитывается текущая позиция в файле.

Функция **fseek** возвращает значение **0** при успешном позиционировании и **-1 (EOF)**- при ошибке. Различные значения параметра **mode**определяют различные способы адресации данных вфайле:

**SEEK\_****SET**определяет **абсолютную**адресацию данных в файле от его начала. Заметим, что функция **ftell**возвращает текущую позицию в абсолютном значении;

**SEEK\_****END**за начало координат берет конец файла (**EOF**).  
long      fsize;

fseek(fl,0L,SEEK\_END); // Установить позицию на конец файла

fsize = ftell(fd);                           // Прочитать значение текущей позиции

**SEEK\_****CUR**дает способ **относительной**адресации от текущего положения указателя в файле. Таким образом, задается *расстояние*в байтах от текущей переменной до адресуемой. Если это*расстояние* само находится в файле, то оно обычно носит название **смещения.**

fseek(fd,100,SEEK\_SET);           // По адресу 100 находится смещение

fread(&P,sizeof(long),1,fd);          // Читается P=46, после чтения текущая позиция

fseek(fd,i,SEEK\_CUR);               // 100+sizeof(long)=104, позиционирование 104+46=150

**47. Потоки. Виды потоков. Классы и объекты потоков ввода-вывода.**

Абстракция потока особенно важна в языке программирования Си, где он представляет собой источник ввода и/или вывода данных, обычно байтов, связанный с файлом, устройством, либо другим процессом.

По сути, ввод/вывод в языке C++ реализован с помощью потоков. Абстрактно, поток — это последовательность символов, к которым можно получить доступ. Со временем поток может производить или потреблять потенциально неограниченные объемы данных.

Есть 2 типа потоков. Поток ввода (или «входной поток») используется для хранения данных, полученных от источника данных: клавиатуры, файла, сети и т.д. Например, пользователь может нажать клавишу на клавиатуре в то время, когда программа не ожидает ввода. Вместо игнорирования нажатия клавиши, данные помещаются во входной поток, где затем ожидают ответа от программы.

И наоборот, поток вывода (или «выходной поток») используется для хранения данных, предоставляемых конкретному потребителю данных: монитору, файлу, принтеру и т.д. При записи данных на устройство вывода, это устройство может быть не готовым принять данные немедленно. Например, принтер все еще может прогреваться, когда программа уже записывает данные в выходной поток. Таким образом, данные будут находиться в потоке вывода до тех пор, пока принтер не начнет их использовать.

Некоторые устройства, такие как файлы и сети, могут быть источниками как ввода, так и вывода данных.

Хорошая новость заключается в том, что программисту не нужно знать детали взаимодействия потоков с разными устройствами и источниками данных, ему нужно только научиться взаимодействовать с этими потоками для чтения и записи данных.

Ввод/вывод в C++

Хотя класс ios является дочерним классу ios\_base, очень часто именно этот класс будет наиболее родительским классом, с которым вы будете работать/взаимодействовать напрямую. Класс ios определяет кучу разных вещей, которые являются общими для потоков ввода/вывода.

Класс istream используется для работы с входными потоками. Оператор извлечения >> используется для извлечения значений из потока. Это имеет смысл: когда пользователь нажимает на клавишу клавиатуры, код этой клавиши помещается во входной поток. Затем программа извлекает это значение из потока и использует его.

Класс ostream используется для работы с выходными потоками. Оператор вставки << используется для помещения значений в поток. Это также имеет смысл: вы вставляете свои значения в поток, а затем потребитель данных (например, монитор) использует их.

Класс iostream может обрабатывать как ввод, так и вывод данных, что позволяет ему осуществлять двунаправленный ввод/вывод.

Наконец, остались 3 класса, оканчивающиеся на \_withassign. Эти потоковые классы являются дочерними классам istream, ostream и iostream (соответственно). В большинстве случаев вы не будете работать с ними напрямую.

Стандартный поток — это предварительно подключенный поток, который предоставляется программе её окружением. Язык C++ поставляется с 4-мя предварительно определенными стандартными объектами потоков, которые вы можете использовать (первые три вы уже встречали):

cin — класс istream\_withassign, связанный со стандартным вводом (обычно это клавиатура);

cout — класс ostream\_withassign, связанный со стандартным выводом (обычно это монитор);

cerr — класс ostream\_withassign, связанный со стандартной ошибкой (обычно это монитор), обеспечивающий небуферизованный вывод;

сlog — класс ostream\_withassign, связанный со стандартной ошибкой (обычно это монитор), обеспечивающий буферизованный вывод.

Небуферизованный вывод обычно обрабатывается сразу же, тогда как буферизованный вывод обычно сохраняется и выводится как блок. Поскольку clog используется редко, то его обычно игнорируют.

**48. Файловые потоки.**

Для работы с файлами в стандартной библиотеке определен заголовочный файл fstream, который определяет базовые потоки для чтения и записи файлов. В частности, это:

 ifstream: для чтения с файла

 ofstream: для записи в файл

 fstream: совмещает запись и чтение

Для работы с данными типа wchar\_t для этих потоков определены двойники:

 wifstream

 wofstream

 wfstream

Открытие файла. При операциях с файлом вначале необходимо открыть файл с помощью функции open(). Данная функция имеет две версии:

 open(путь)

 open(путь, режим)

Для открытия файла в функцию необходимо передать путь к файлу в виде строки. 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

Если при открытии режим не указан, то по умолчанию для объектов ofstream применяется режим ios::out, а для объектов ifstream - режим ios::in. Для объектов fstream совмещаются режимы ios::out и ios::in.

**49. Средства работы с потоками ввода/вывода.**

Посимвольный ввод-вывод

В функциях посимвольного ввода-вывода происходит прием одного символа из файла или передача одного символа в файл:

**int fgetc(FILE \*f)** - считывает и возвращает символ из файла f;

**int fputc(int ch, FILE \*f)** - записывает в файл f код ch символа.

Построчный ввод-вывод

В функциях построчного ввода-вывода происходит перенос из файла, или

в файл строк символов:

**int fgets (char \*S, int m, FILE \*f)** - чтение из файла f в строку S m байт;

**int fputs (char \*S, FILE \*f)** - запись в файл f строки S до тех пор, пока

не встретится '\0', который в файл не переносится и на символ '\n' не заменяется.

Блоковый ввод-вывод

В функциях блокового ввода-вывода работа происходит с целыми блоками информации:

**int fread (void \*p, int size,int n, FILE \*f)-** считывает n блоков по size байт каждый из файла

f в область памяти с указателем p (необходимо заранее отвести память под считываемый блок);

**int fwrite (void \*p, int size,int n, FILE \*f)-** записывает n блоков по size байт каждый из области памяти с указателем p в файл f.

Форматированный ввод-вывод производится функциями:

**int fscanf (FILE \*f, char \*формат,список адресов объектов)-** считывает из файла f информацию для объектов в соответствии с указанными форматами;

**int fprintf (FILE \*f, char \*формат,список объектов)-** записывает в файл f объекты, указанные в списке в соответствии с форматами.

Данные функции аналогичны функциям scanf() и printf(), рассмотренным

раньше, только добавлен параметр – указатель на файл.

Рассмотрим наиболее распространенные функции с помощью которых можно организовать работу с файлами:

**int fileno(FILE \*f)–**возвращает значение дескриптора файла f-fd(число, определяющее номер файла);

**long filelength(int fd)–**возвращает длину файла, имеющего номер (дескриптор) fd в байтах;

**int chsize(int fd, long pos)–**выполняет изменение размера файла, имеющего номер fd, признак конца файла устанавливается после байта с номером pos;

**intfseek(FILE \*f, long size, intkod)–**выполняет смещение указателя файла f на size байт в направлении признака kod:0 -от начала файла;1 -от текущей позиции указателя;2 -от конца файла;

**long ftell(FILE \*f)–**возвращает значение указателя на текущую позицию файла (-1 –ошибка);

**int feof(FILE \*f)–**возвращает ненулевое значение при правильной записи признака конца файла;

**intfgetpos(FILE \*f, long\*pos)–**определяет значение текущей позиции pos фай-ла f, возвращает 0 при успешном завершении

**50. Форматирование ввода-вывода. Использование манипуляторов.**

Возможность управлять вводом-выводом в С++, обеспечивают форматирующие функции-члены, флаги и манипуляторы. Флаги, функции и манипуляторы выполняют одну и туже задачу — задают определённый формат ввода/вывода информации в потоках. Ввод/вывод на экран/с экрана в С++ осуществляется с помощью операторов cin и cout соответственно, а значит манипуляторы форматирования используются совместно с данными операторами ввода/вывода. Различие между функциями флагами и манипуляторами форматирования состоит в способе их применения. Теперь рассмотрим способы применения объектов форматирования.

//Основные форматирующие функции-члены:

cout.fill('/\*symbol\*/'); // устанавливает символ заполнитель

// где symbol - символ заполнитель, символ передаётся в одинарных кавычках

cout.width(/\*width\_field\*/); // задает ширину поля

// где width\_field - количество позиций(одна позиция вмещает один символ)

cout.precision(/\*number\*/); // задает количество знаков после десятичной точки

// где number - количество знаков после десятичной точки

Доступ к функциям осуществляется через операцию точка, а в круглых скобочках передаётся аргумент. Аргумент функции fill() может передаваться в виде символа, обрамленного одинарными кавычками, или в виде числа(код символа). Одних функций не достаточно для форматирования потоков ввода/вывода, поэтому в С++ предусмотрен ещё один способ форматирования — флаги.

Флаги форматирования позволяют включить или выключить один из параметров ввода/вывода. Чтобы установить флаг ввода/вывода, необходимо вызвать функцию setf(), если необходимо отключить флаг вывода, то используется функция unsetf(). Далее показаны конструкции установки и снятия флагов вывода.

// установка флага вывода

cout.setf( ios::/\*name\_flag\*/ );

// где name\_flag - это имя флага

Доступ к функциям оператора вывода выполняется через операцию точка. Метод setf() принимает один аргумент — имя флага. Флаги вывода объявлены в классе ios, поэтому, перед тем, как обратиться к флагу, необходимо написать имя класса — ios, после которого, с помощью операции разрешения области действия, вызвать нужный флаг.

// снятие флага вывода

cout.unsetf( ios::/\*name\_flag\*/ );

// где name\_flag - это имя флага

Если при вводе/выводе необходимо установить(снять) несколько флагов, то можно воспользоваться поразрядной логической операцией ИЛИ |. В этом случае конструкция языка C++ будет такой:

// установка нескольких флагов

cout.setf( ios::/\*name\_flag1\*/ | ios::/\*name\_flag2\*/ | ios::/\*name\_flag\_n\*/ );

// снятие нескольких флагов

cout.unsetf( ios::/\*name\_flag1\*/ | ios::/\*name\_flag2\*/ | ios::/\*name\_flag\_n\*/ );

**51. Поиск. Линейный.**

Для нахождения некоторого элемента (ключа) в заданном неупорядоченном массиве используется алгоритм линейного (последовательного) поиска. Он работает как с неотсортированными массивами(более эффективно с неотсорт), так и отсортированными.

Алгоритм: Начиная с первого, все элементы массива последовательно просматриваются и сравниваются с искомым. Если на каком-то шаге текущий элемент окажется равным искомому, тогда элемент считается найденным, и в качестве результата возвращается номер этого элемента, либо другая информация о нем.. Иначе, следуют возвратить что-то, что может оповестить о его отсутствии в пройденной последовательности.

Код:

int LineSearch(int A[], int key)  
{  
for (i=0; i<N; i++)  
if (A[i]==key) return i;  
return -1;  
}

**52. Поиск. Двоичный.**

Двоичный(бинарный) поиск — алгоритм поиска элемента в отсортированном массиве. Двоичный поиск можно использовать только в том случае, если есть массив, все элементы которого упорядочены. Бинарный поиск не используется для поиска максимального или минимального элементов, так как в отсортированном массиве эти элементы содержатся в начале и в конце массива соответственно. Поэтому алгоритм бинарного поиска применим, если необходимо найти некоторый ключевой элемент в массиве. То есть организовать поиск по ключу, где ключ — это определённое значение в массиве.

Алгоритм: есть упорядоченный массив, берем число из середины массива, сравниваем с искомым. Если оно оказалось больше, значит искомое число в первой половине массива, если меньше — во второй. Продолжаем делить оставшуюся половину, когда находим нужное число возвращаем его индекс, если не находим возвращаем null.

int BinarySearch(char\* N, unsigned n, char key) // поиск

{

int L=0;

int R=n;

char S[255];

ifstream in1(N);

while (!in1.eof())

{

for(int i=0; i<225; i++)

in1.get(S[i]);

}

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

{

while (L<R)

{

int m = (L+R)/2;

if (S[m]<key)

L=m+1;

else

R=m;

}

}

if (S[R]==key)

{

int q=R+1;

cout<<" Номер элемента \n"<<q;

}

else

cout <<" Элемент не найден \n";

return -1;

}

**53. Пузырьковая сортировка.**

Сортировка пузырьком — один из самых известных алгоритмов сортировки. Здесь нужно последовательно сравнивать значения соседних элементов и менять числа местами, если предыдущее оказывается больше последующего. Таким образом элементы с большими значениями оказываются в конце списка, а с меньшими остаются в начале.

Этот алгоритм считается учебным и почти не применяется на практике из-за низкой эффективности: он медленно работает на тестах, в которых маленькие элементы (их называют «черепахами») стоят в конце массива. Однако на нём основаны многие другие методы, например, шейкерная сортировка и сортировка расчёской.

// Сортировка массива пузырьком

for (int i = 0; i < size - 1; i++) {

for (int j = 0; j < size - i - 1; j++)

{ if (arr[j] > arr[j + 1]) {

// меняем элементы местами

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp; } } }

**54. Сортировка методом простого включения.**

Хотя этот метод сортировки намного менее эффективен, чем сложные алгоритмы (такие как быстрая сортировка), у него есть ряд преимуществ:

 прост в реализации;

 эффективен на небольших наборах данных, на наборах данных до десятков элементов может оказаться

 лучшим;

 эффективен на наборах данных, которые уже частично отсортированы;

 это устойчивый алгоритм сортировки (не меняет порядок элементов, которые уже отсортированы);

 может сортировать массив по мере его получения;

 не требует временной памяти, даже под стек.

На каждом шаге алгоритма выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированной последовательности до тех пор, пока набор входных данных не будет исчерпан. Метод выбора очередного элемента из исходного массива произволен; может использоваться практически любой алгоритм выбора.

// Сортировка простого включение

static void Main(string[] args) {

int[] a = { 7, 0, -4, 3, 1, -2, 5 };

int i, j, min, temp;

for(i=0; i < a.Length - 1; i++) {

min = i; //устанавливаем начальное значение минимального индекса

//находим минимальный индекс элемента

for (j = i + 1; j < a.Length; j++)

{ if (a[j] < a[min]) min = j; } //меняем значения местами

temp = a[i]; a[i] = a[min]; a[min] = temp; }

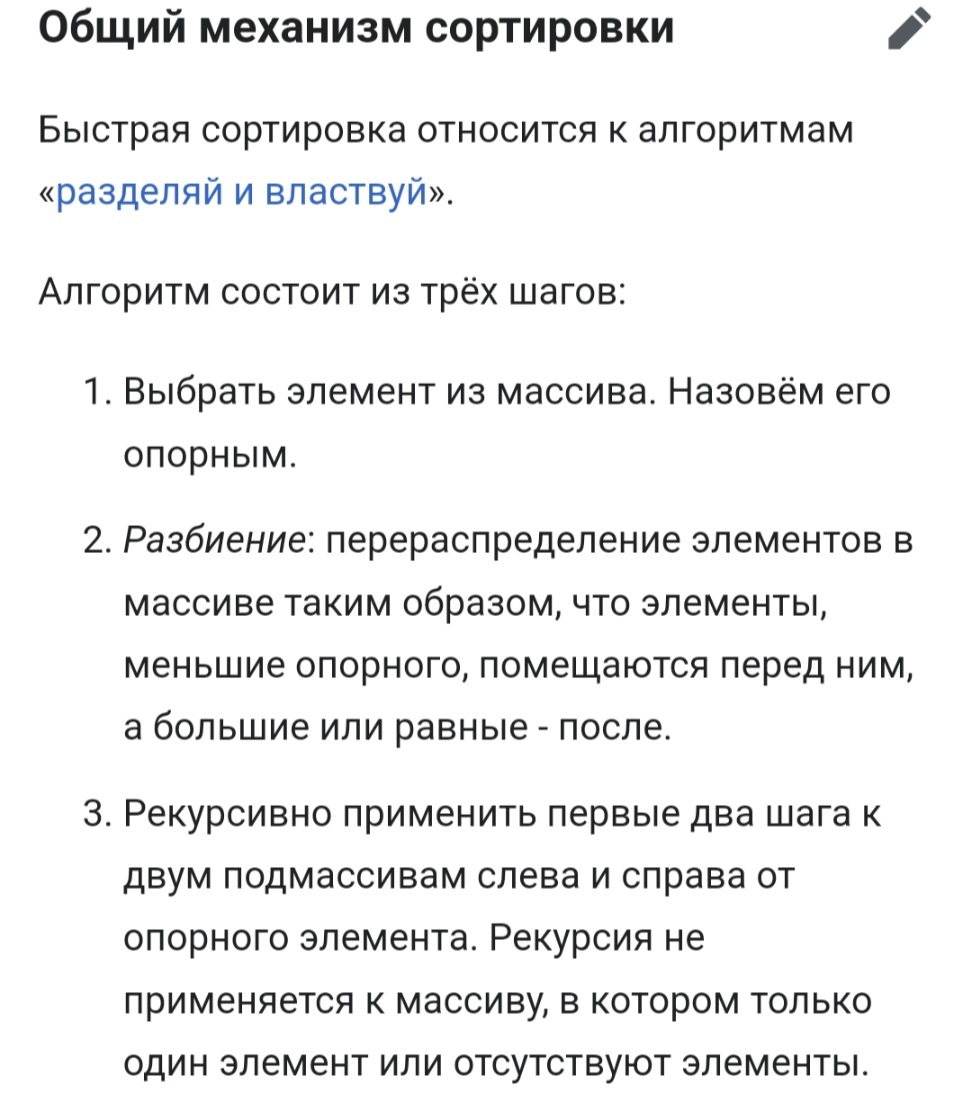
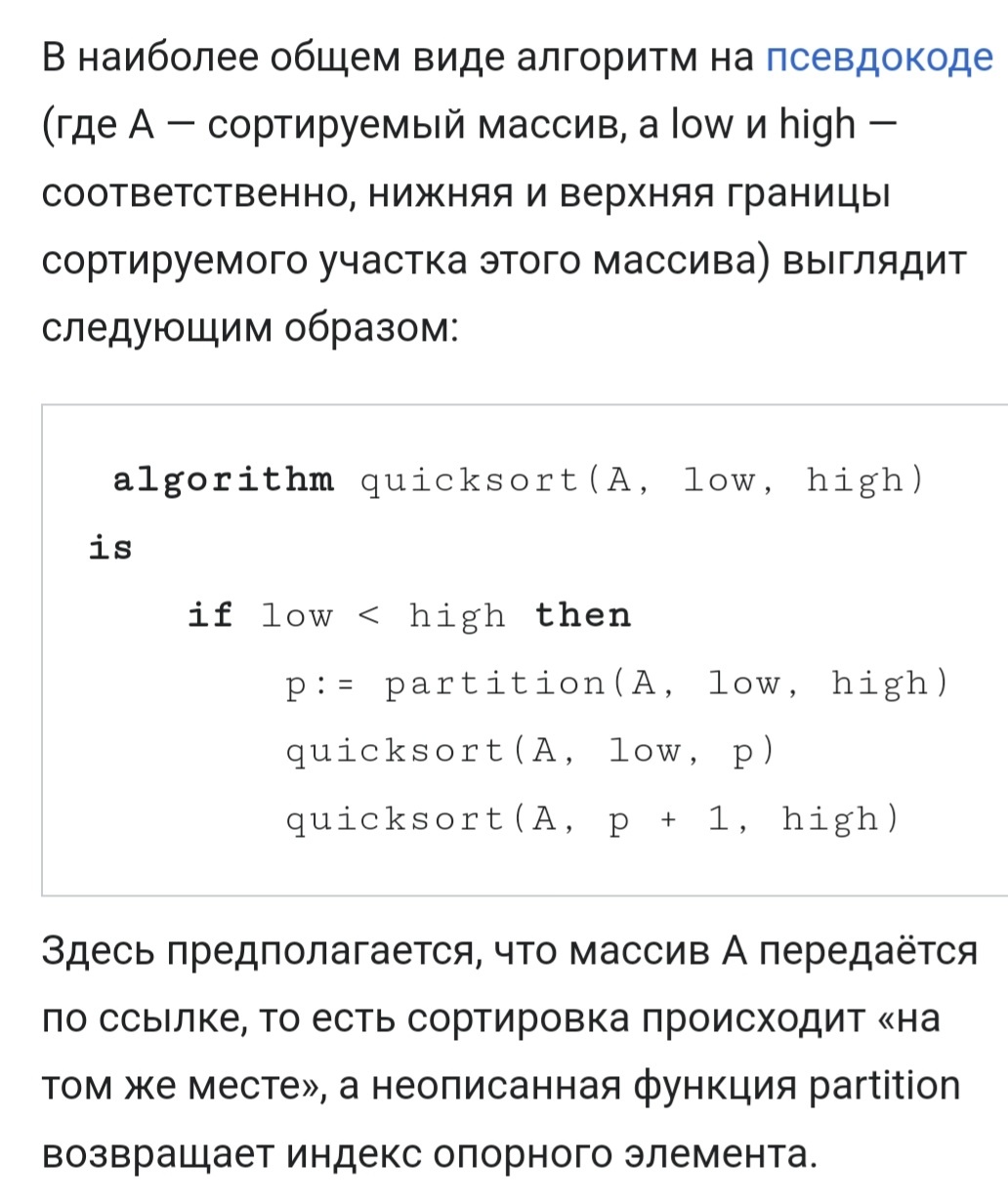
**55. Сортировка выбором.**

Находим (выбираем) в массиве элемент с минимальным значением на интервале от 1-го элемента до n-го (последнего) элемента и меняем его местами с первым элементом. На втором шаге находим элемент с минимальным значением на интервале от 2-го до n-го элемента и меняем его местами со вторым элементом. И так далее для всех элементов до n-1-го.Таким образом, суть алгоритма сортировки выбором сводится к многократному поиску минимального (максимального) элементов в неотсортированной части массива.

void selection\_sort (int \*a, int n){  
int i, j, m, t;  
for(i = 0; i < n; i++){  
for(j = i, m = i; j < n; j++){  
if(a[j]< a[m])  
 m = j;}  
 t = a[i];  
 a[i] = a[m];  
 a[m] = t;}}  
int main (){  
int a[] = {4, 65, 2, -31, 0, 99, 2, 83, 782, 1};  
int n = sizeof a / sizeof a[0];  
 selection\_sort(a, n);  
return0;}

**56. Сортировка Хоара**

Быстрая сортировка, сортировка Хоара (англ. quicksort), часто называемая qsort (по имени в стандартной библиотеке языка Си) — алгоритм сортировки, разработанный английским информатиком Тони Хоаром во время своей работы в МГУ в 1960 году.



**57. Сортировка Шейкерная.**

Сортировка перемешиванием ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Cocktailsort*) — разновидность [пузырьковой сортировки](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BF%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%BE%D0%BC). Границы рабочей части массива (т.е. части массива, где происходит движение) устанавливаются в месте последнего обмена на каждой итерации. Массив просматривается поочередно справа налево и слева направо.

#define try\_swap { if (a[i] < a[i - 1])\  
 { t = a[i]; a[i] = a[i - 1]; a[i - 1] = t; t = 0;} }  
void cocktailsort(int\*a,size\_t len){  
 size\_t i;  
 int t =0;  
 while(!t){  
 for(i =1, t =1; i < len; i++) try\_swap;  
 if(t)break;  
 for(i = len -1, t =1; i; i--) try\_swap;}}

int main(){  
 int x[]={5,-1,101,-4,0,1,8,6,2,3};  
 size\_t i, len =sizeof(x)/sizeof(x[0]);  
 cocktailsort(x, len);  
 for(i =0; i < len; i++)  
 [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("%d\n", x[i]);  
 return0;}