1. Понятие функции (подпрограммы) на языке С. Прототип функции: назначение и использование. Способы объявления параметров. Область видимости функции.
2. Формальные и фактические параметры в описании и вызове функций

(подпрограмм). Передача параметров. Возврат управления из функции.

1. Локальные и глобальные переменные при использовании подпрограмм. Способы передачи аргументов в подпрограмму.
2. Спецификаторы хранения. Локальные и глобальные статические переменные. Регистровые переменные.
3. Спецификаторы хранения. Спецификатор extern. Многофайловая компиляция программ.
4. Декларация и инициализация указателя. Операции над указателями.
5. Указатели и массивы. Индексация указателей.
6. Способы передачи аргументов в подпрограмму. Функции при работе с массивами. Передача одномерного массива в функцию.
7. Понятие рекурсии. Организация рекурсии с помощью подпрограмм. Примеры работы рекурсии.
8. Структура рекурсивного алгоритма. Хранение данных при рекурсии. Возможные ошибки программирования.
9. Основные возможности работы с динамической памятью.
10. Создание динамических массивов и правила работы с ними.
11. Файловая система изучаемого языка С. Понятие файла. Типы файлов. Потоки и файлы. Стандартные потоки ввода/вывода. Текстовые и бинарные потоки.
12. Основные этапы работы с файлами на С. Указатель файла. Функции открытия и закрытия файла. Режимы доступа к файлам.
13. Наиболее распространенные функции работы с файлами. Макросы для работы с файлами.
14. Основные функции записи в файл. Запись символов, строк, форматированный вывод во внешний файл. Запись многобайтных данных.
15. Основные функции чтения из файла. Чтение символов, строк, чтение форматированных данных. Чтение многобайтных данных.
16. Способы доступа к файлам. Последовательный доступ к файлам. Произвольный доступ к файлам при организации двоичного ввода/вывода.
17. Файлы с произвольным доступом. Функции, реализующие произвольный доступ к файлу.
18. Потоковый ввод/вывод данных на С++. Операции для работы с потоком. Манипулятор endl.
19. Флаги форматированного ввода/вывода на С++. Методы и манипуляторы форматирования.
20. Работа с текстовыми и двоичными файлами на С++.
21. Программная реализация алгоритмов работы с файлами. Основные алгоритмы работы с данными файлов. Примеры вычислительных алгоритмов.
22. Показатели эффективности алгоритмов. Анализ времени выполнения алгоритмов.
23. Асимптотический анализ сложности алгоритмов. Рассматриваемые случаи. О- нотация. Классы сложности алгоритмов.
24. Алгоритмы поиска. Методы последовательного поиска. Сложность. Описание: словесное, графическое, на языке программирования.
25. Алгоритм бинарного поиска. Сложность. Описание: словесное, графическое, на языке программирования.
26. Алгоритм индексно-последовательного поиска. Сложность. Описание: словесное, графическое, на языке программирования.
27. Алгоритмы сортировки. Классификация и основные виды. Оценка сложности алгоритмов сортировки.
28. Алгоритмы простых сортировок: пузырьком, выбором, вставками. Словесное и графическое описание, реализация на языке программирования.
29. Обменные сортировки. Методы улучшения сортировки обменом. Словесное описание и реализация на языке программирования.
30. Алгоритмы сортировок выбором. Методы улучшения. Словесное описание и реализация на языке программирования.
31. Алгоритмы сортировок вставками. Методы улучшения. Словесное описание и реализация на языке программирования.
32. Алгоритмы сортировок: подсчетом, расческой, гномья и другие гибридные сортировки.
33. Типы данных пользователя. Понятие структуры. Объявление структуры. Объявление переменной-структуры. Инициализация переменных-структур.
34. Доступ к членам структуры. Массив переменных структуры. Указатель на структуру. Доступ к полям структуры посредством указателя.
35. Понятие объединения. Объявление и использование объединений. Назначение, объявление и использование перечислений.
36. Понятие графа. Элементы графа. Неориентированный, ориентированный, взвешенный, невзвешенный граф. Правила обхода графа. Матрица смежности.
37. Динамические структуры данных. Стек, очередь, список. Работа со стеком: добавление, удаление, поиск, сортировка элементов стека.
38. Однонаправленные очереди. Добавление, удаление элементов. Удаление очереди.
39. Работа с двусвязными списками. Добавление, удаление элементов.
40. Сортировка слиянием двусвязного списка.
41. Работа с двусвязными циклическими списками.
42. Древовидные структуры данных. Использование древовидных структур. Двоичное дерево поиска.
43. Понятие хеширования. Схемы хеширования. Хеш-таблица с линейной адресацией.
44. Хеш-таблицы с квадратичной и произвольной адресацией, с двойным хешированием. Хеш-таблица на основе связанных списков.
45. **Понятие функции (подпрограммы) на языке C. Прототип функции: назначение и использование. Способы объявления параметров. Область видимости функции.**

Функция – именованная и определенным образом оформленная группа операторов, вызываемых по мере необходимости.

Применение функций:

* сокращают код при многократном выполнении операций в разных местах программы;
* большую программу разбивает на части для упрощения представления (удобочитаемости);
* сложные алгоритмы разбивает на несколько этапов для достижения простоты реализации;

Функции – это строительные блоки для всех программ на языках C, C++.

Функции разрабатываются автономно, а затем подключаются и вызываются из главной функции main().

При применении функции различают:

* прототип;
* вызов;
* описание.

Общий вид прототипа функции:

тип возвр.\_значения имя\_функции (список\_параметров);

* тип\_возвр.\_значения определяет тип данного, возвращаемого функцией. Функция может возвращать любой тип данных, за исключением массивов;
* имя\_функции – имя функции;
* список\_параметров – список формальных параметров или типов данных параметров.

Если функция не имеет параметров, её список параметров пуст. В списке параметров перечисляются типы аргументов и их имена, разделенные запятыми, каждый параметр функции должен объявляться отдельно. Общий вид объявления параметров: f (тип\_переменной имя). При желании имена переменных в прототипе могут быть опущены.

Использование прототипов:

* функции, имеющие прототипы могут быть вызваны из других модулей;
* использование прототипов позволяет размещать функции в произвольном порядке (а не до первого использования);
* размещение прототипов в одном месте делает программу более читабельной.

Правила, определяющие область видимости, устанавливают, видит ли одна часть программы другую часть кода или данных и может ли к ним обращаться.

Код функции является закрытым и недоступным для любых операторов, расположенных в других функциях кроме операторов вызова. Код, образующий тело функции, скрыт от остальной части программы, если он не использует глобальные переменные.

1. **Формальные и фактические параметры в описании и вызове функции (подпрограмм). Передача параметров. Возврат управления из функции.**

В работе с подпрограммами приходится иметь дело с двумя видами параметров: список фактических параметров и список формальных параметров.

При объявлении функции указываются формальные параметры, которые потом используются внутри самой функции. При вызове функции мы используем фактические параметры (аргумент функции).

Если функция имеет аргументы, в ее заголовке должны быть объявлены переменные, принимающие их значения. Эти переменные называются формальными параметрами функции. Как и локальные переменные они создаются при входе в функцию и уничтожаются при выходе из нее.

Объявления параметров размещаются после имени функции.

Значения передаются функции путем использования фактических параметров. Конкретные значения фактических параметров присваиваются соответственно формальным параметрам.

Список аргументов – константы, переменные, выражения (значения перед вызовом определяются компилятором). Аргументы списка вызова должны совпадать со списком параметров функции по количеству и порядку следования. Типы аргументов при передаче преобразуются к указанным в функции типам параметров.

Способы передачи аргументов в подпрограмму:

* передача параметров по значению;
* передача параметров по ссылке;
* передача параметров по указателю.

Передача параметров по значению:

Формальному параметру функции присваивается копия значения аргумента, и все изменения параметра не отражаются в аргументе.

Передача параметров по ссылке:

Параметру присваивается адрес аргумента. Внутри функции этот адрес открывает доступ к фактическому аргументу. Все изменения, которым подвергается параметр, отражаются на аргументе.

Для передачи параметров по ссылке вместо аргумента передают указатель на него. Поскольку функция получает адрес аргумента, ее код может изменять значение фактического аргумента вне функции.

При вызове функции, использующей передачу параметров по ссылке, из основной программы передаются адреса аргументов.

Передача параметров по указателю:

Параметрами функции являются адреса, передаются значения. Ссылочный параметр – псевдоним соответствующего аргумента.

Параметры со значениями по умолчанию:

При объявлении функции для некоторых аргументов можно задавать значение по умолчанию, которое передается в функцию в случае, если при вызове соответствующий параметр не задан.

Компилятор присваивает имеющиеся значения последовательно слева направо, поэтому аргументы, имеющие значение по умолчанию, должны располагаться после аргументов, не имеющих такого значения.

В языке C/C++ весь массив нельзя передать в качестве аргумента функции. Вместо этого можно передать указатель на массив, т.е. имя массива без индексов. Если аргументом функции является массив, ей передается его адрес. Функция, получившая массив, получает доступ ко всем его элементам и может его модифицировать.

Передача переменного числа параметров:

Формат объявления функции с переменным числом параметров: тип\_возвращаемого\_значения имя\_функции (список\_параметров, …)

Список параметров содержит хотя бы один обязательный параметр. Многоточие указывает на возможность добавления любого числа параметров.

Функция может завершать выполнение и осуществлять возврат в вызывающую программу двумя способами. Первый способ используется тогда, когда после выполнения последнего оператора в функции встречается закрывающая фигурная скобка (}). Но на практике не так уж много функций используют именно такой способ завершения выполнения. В большинстве функций для завершения выполнения используется оператор return — или потому, что необхо-димо вернуть значение, или чтобы сделать код функции проще и эффективнее.

В функции может быть несколько операторов return.

1. **Локальные и глобальные переменные при использовании подпрограмм. Способы передачи аргументов в подпрограмму.**

Глобальные переменные доступны из любой части программы и могут быть использованы где угодно. Они сохраняют свои значения на всем протяжении выполнения программы. Объявления глобальных переменных должны размещаться вне всех функций.

Применение большого количества глобальных переменных повышает вероятность ошибок вследствие непредсказуемых побочных эффектов. Большинство проблем, возникающих при разработке больших программ, является следствием непредвиденного изменения значений переменных, которые используются в любом месте программы.

Локальные переменные – переменные, определенные внутри функции. Создаются при входе в функцию и уничтожаются при выходе из нее. Иначе, локальные переменные не сохраняют значения между вызовами функции. Единственное исключение – статические локальные переменные.

Список аргументов – константы, переменные, выражения (значения перед вызовом определяются компилятором). Аргументы списка вызова должны совпадать со списком параметров функции по количеству и порядку следования. Типы аргументов при передаче преобразуются к указанным в функции типам параметров.

Способы передачи аргументов в подпрограмму:

* передача параметров по значению;
* передача параметров по ссылке;
* передача параметров по указателю.

Передача параметров по значению:

Формальному параметру функции присваивается копия значения аргумента, и все изменения параметра не отражаются в аргументе.

Передача параметров по ссылке:

Параметру присваивается адрес аргумента. Внутри функции этот адрес открывает доступ к фактическому аргументу. Все изменения, которым подвергается параметр, отражаются на аргументе.

Для передачи параметров по ссылке вместо аргумента передают указатель на него. Поскольку функция получает адрес аргумента, ее код может изменять значение фактического аргумента вне функции.

При вызове функции, использующей передачу параметров по ссылке, из основной программы передаются адреса аргументов.

Передача параметров по указателю:

Параметрами функции являются адреса, передаются значения. Ссылочный параметр – псевдоним соответствующего аргумента.

Параметры со значениями по умолчанию:

При объявлении функции для некоторых аргументов можно задавать значение по умолчанию, которое передается в функцию в случае, если при вызове соответствующий параметр не задан.

Компилятор присваивает имеющиеся значения последовательно слева направо, поэтому аргументы, имеющие значение по умолчанию, должны располагаться после аргументов, не имеющих такого значения.

В языке C/C++ весь массив нельзя передать в качестве аргумента функции. Вместо этого можно передать указатель на массив, т.е. имя массива без индексов. Если аргументом функции является массив, ей передается его адрес. Функция, получившая массив, получает доступ ко всем его элементам и может его модифицировать.

Передача переменного числа параметров:

Формат объявления функции с переменным числом параметров: тип\_возвращаемого\_значения имя\_функции (список\_параметров, …)

1. **Спецификаторы хранения. Локальные и глобальные статические переменные. Регистровые переменные**

*\*Область видимости – определяет, в каких частях программы возможно использование данной переменной.*

*\*Класс памяти – время, в течение которого переменная существует в памяти компьютера.*

*Классы памяти:*

* *автоматический, локальный;*
* *статический, локальный;*
* *внешний, глобальный;*
* *регистровый, локальный.*

*\*Время жизни переменной – период времени между созданием и уничтожение переменной*

Спецификаторы хранения:

* auto;
* extern;
* static;
* register.

Они сообщают компилятору, как хранить объявленную переменную. Общий вид объявления: спецификатор\_хранения тип имя\_переменной.

Переменные, объявленные со спецификатором ***static***, хранятся постоянно внутри своей функции или файла. В отличие от глобальных переменных они невидимы за пределами своей функции или файла, но они сохраняют свое значение между вызовами. Эта особенность делает их полезными в общих и библиотечных функциях, которые будут использоваться другими программистами. Спецификатор ***static*** воздействует на локальные и глобальные переменные по-разному.

Локальные статические переменные

Для локальной переменной, описанной со спецификатором ***static***, компилятор выделяет в постоянное пользование участок памяти, точно так же, как и для глобальных переменных. Коренное отличие статических локальных от глобальных переменных заключается в том, что статические локальные переменные видны только внутри блока, в котором они объявлены. Говоря коротко, статические локальные переменные — это локальные переменные, сохраняющие свое значение между вызовами функции.

Статические локальные переменные очень важны при создании функций, работающих отдельно, так как многие процедуры требуют сохранения некоторых значений между вызовами. Если бы не было статических переменных, вместо них пришлось бы использовать глобальные, подвергая их риску непреднамеренного изменения другими участками программы.

Статическую локальную переменную можно инициализировать. Это значение присваивается ей только один раз — в начале работы всей программы, но не при каждом входе в блок программы, как обычной локальной переменной.

Глобальные статические переменные

Спецификатор ***static*** в объявлении глобальной переменной заставляет компилятор создать глобальную переменную, видимую только в том файле, в котором она объявлена. Статическая глобальная переменная, таким образом, подвергается внутреннему связыванию. Это значит, что хоть эта переменная и глобальная, тем не менее процедуры в других файлах не увидят ее и не смогут случайно изменить ее значение. Этим снижается риск нежелательных побочных эффектов. А в тех относительно редких случаях, когда для выполнения задачи статическая локальная переменная не подойдет, можно создать небольшой отдельный файл, который содержит только функции, в которых используется эта статическая глобальная переменная. Затем этот файл необходимо откомпилировать отдельно; тогда можно быть уверенным, что побочных эффектов не будет.

Спецификатор ***register*** используется, чтобы компилятор сохранял значение переменной в регистре центрального процессора, а не в памяти, как обычно.

В первых версиях компиляторов С спецификатор ***register*** сообщал компилятору, что переменная должна храниться в регистре процессора, а не в оперативной памяти, как все остальные переменные. Это приводит к тому, что операции с переменной ***register*** осуществляются намного быстрее, чем с обычными переменными, потому такая переменная уже находится в процессоре и не нужно тратить время на выборку ее значения из оперативной памяти (и на запись в память).

В настоящее время определение спецификатора ***register*** существенно расширено. Стандарты декларируют "доступ к объекту так быстро, как только возможно". Практически при этом символьные и целые переменные по-прежнему размещаются в регистрах процессора. Конечно, большие объекты (например, массивы) не могут поместиться в регистры процессора, однако компилятор получает указание "позаботиться" о быстродействии операций с ними. В зависимости от конкретной реализации компилятора и операционной системы переменные ***register*** обрабатываются по-разному. Иногда спецификатор ***register*** попросту игнорируется, а переменная обрабатывается как обычная, однако на практике это бывает редко.

Спецификатор ***register*** можно применить только к локальным переменным и формальным параметрам функций. В объявлении глобальных переменных применение спецификатора ***register*** не допускается.

1. **Спецификаторы хранения. Спецификатор extern. Многофайловая компиляция программ**

Спецификаторы хранения:

* auto;
* extern;
* static;
* register.

Они сообщают компилятору, как хранить объявленную переменную. Общий вид объявления: спецификатор\_хранения тип имя\_переменной.

Спецификатор extern указывает на то, что к объекту применяется внешнее связывание, именно поэтому они будут доступны во всей программе. *Описание* выделяет для объекта участок памяти, где он будет находиться. Один и тот же объект может быть объявлен неоднократно в разных местах, но описан он может быть только один раз.

В большинстве случаев объявление переменной является в то же время и ее описанием. Однако, если перед именем переменной стоит спецификатор ***extern***, то объявление переменной может и не быть ее описанием. Таким образом, если нужно сослаться на переменную, определенную в другой части программы, необходимо объявить ее как внешнюю (***extern***).

Спецификатор ***extern*** позволяет объявить переменную, не описывая ее. Но если в объявлении со спецификатором ***extern*** инициализировать переменную, то это объявление становится также и описанием. При этом программист обязательно должен учитывать, что объект может иметь много объявлений, но лишь одно описание.

Спецификатор ***extern*** играет большую роль в программах, состоящих из многих файлов. В языке С программа может быть записана в нескольких файлах, которые компилируются раздельно, а затем компонуются в одно целое. В этом случае необходимо как-то сообщить всем файлам о глобальных переменных программы. Самый лучший (и наиболее переносимый) способ сделать это – определить (описать) все глобальные переменные в одном файле и объявить их со спецификатором ***extern*** в остальных файлах.

На практике программисты обычно включают объявления ***extern*** в заголовочные файлы, которые просто подключаются к каждому файлу исходного текста программы. Это более легкий путь, который к тому же приводит к меньшему количеству ошибок, чем повторение этих объявлений вручную в каждом файле.

Короткая программа на языке С может состоять всего лишь из одного файла исходного текста. Однако при увеличении длины программы увеличивается также и время компиляции. Программа на С может состоять из двух или более файлов, компилируемых отдельно. Скомпилированные файлы программы компонуются с процедурами из библиотеки, формируя таким образом объектный код программы. Преимущество раздельной компиляции состоит в том, что при изменении одного файла нет необходимости перекомпилировать заново всю программу. При работе со сложными проектами это экономит много времени. Раздельная компиляция позволяет также нескольким программистам работать над одним проектом, так как она служит средством организации исходного текста программы для большого проекта.

1. **Декларация и инициализация указателя. Операции над указателями**

Указатель – это переменная, в которой хранится адрес другого объекта (как правило другой переменной).

Если одна переменная содержит адрес другой, говорят, что первая переменная ссылается на вторую. Переменная, хранящая адрес ячейки, должна быть объявлена как указатель.

Объявление указателя состоит из имени базового типа, символа \* и имени переменной.

Формат: тип\_указателя \* имя\_указателя;

Операторы для работы с указателями:

& – взятие адреса;

\* – разыменование.

Например:

int count, count2, \*m;

m = &count // указателю присвоили адрес переменной count

count2 = \*m; // переменной count2 присвоили значение count

Особенности указателей.

Указатель должен ссылаться на переменную правильного типа. Например, если в программе объявлен указатель на целочисленную переменную, компилятор полагает, что адрес, который в нем содержится, относится к переменной типа int, независимо от того, так ли это на самом деле.

Поскольку указателю можно присвоить любой адрес, то в случае несоответствия типов данных он будет обрезан или дополнен.

Адресная арифметика.

Операции с указателями:

* сложение с целыми числами;
* вычитание целых чисел.

Пусть int \*p ссылается на целочисленную переменную, размещенную по адресу 2000, и в системе отводится 4 байта для хранения целых чисел. И после операции p++, переменная будет равна не 2001, а 2004.

* При увеличении указатель ссылается на ячейку, в которой хранится следующий элемент базового типа.
* При уменьшении он ссылается на предыдущий элемент.
* Для указателей на символы сохраняются правила «обычной» арифметики, поскольку размер char = 1байт.
* Все остальные указатели увеличиваются или уменьшаются на длину соответствующей переменной, на которую они ссылаются.

1. **Указатели и массивы. Индексация указателей.**

Имя массива является указателем на его первый элемент. Если массив объявлен с помощью оператора int mass[10], то указателем на его первый элемент является mass.

Пример:

int \*p, mass[10];

p = mass; p = &mass[0]; // идентичны

Обращаться к элементам массива можно двумя способами:

int a[10];

a[4]=10;

или с помощью адресной арифметики:

\*(a+4)=10;

Индексация указателей:

* в языке C/C++ указатели и массивы тесно связаны друг с другом. Имя массива без индекса – это указатель на первый элемент.

1. **Способы передачи аргументов в подпрограмму. Функции при работе с массивами. Передача одномерного массива в функцию**

Список аргументов – константы, переменные, выражения (значения перед вызовом определяются компилятором). Аргументы списка вызова должны совпадать со списком параметров функции по количеству и порядку следования. Типы аргументов при передаче преобразуются к указанным в функции типам параметров.

Способы передачи аргументов в подпрограмму:

* передача параметров по значению;
* передача параметров по ссылке;
* передача параметров по указателю.

Передача параметров по значению:

Формальному параметру функции присваивается копия значения аргумента, и все изменения параметра не отражаются в аргументе.

Передача параметров по ссылке:

Параметру присваивается адрес аргумента. Внутри функции этот адрес открывает доступ к фактическому аргументу. Все изменения, которым подвергается параметр, отражаются на аргументе.

Для передачи параметров по ссылке вместо аргумента передают указатель на него. Поскольку функция получает адрес аргумента, ее код может изменять значение фактического аргумента вне функции.

При вызове функции, использующей передачу параметров по ссылке, из основной программы передаются адреса аргументов.

Передача параметров по указателю:

Параметрами функции являются адреса, передаются значения. Ссылочный параметр – псевдоним соответствующего аргумента.

Параметры со значениями по умолчанию:

При объявлении функции для некоторых аргументов можно задавать значение по умолчанию, которое передается в функцию в случае, если при вызове соответствующий параметр не задан.

Компилятор присваивает имеющиеся значения последовательно слева направо, поэтому аргументы, имеющие значение по умолчанию, должны располагаться после аргументов, не имеющих такого значения.

В языке C/C++ весь массив нельзя передать в качестве аргумента функции. Вместо этого можно передать указатель на массив, т.е. имя массива без индексов. Если аргументом функции является массив, ей передается его адрес. Функция, получившая массив, получает доступ ко всем его элементам и может его модифицировать.

Если аргументом функции является одномерный массив, ее формальный параметр можно объявить как указатель:

void func1(int \*x) { }

Если аргументом функции является одномерный массив, можно передать указатель на массив фиксированного размера:

void func1(int x[10]) {}

Если аргументом функции является одномерный массив, можно использовать массив неопределенного размера:

void func1(int x[ ]) {}

Передача переменного числа параметров:

Формат объявления функции с переменным числом параметров: тип\_возвращаемого\_значения имя\_функции (список\_параметров, …)

1. **Понятие рекурсии. Организация рекурсии с помощью подпрограмм. Примеры работы рекурсии**

Рекурсия – процесс определения какого-либо понятия через него же. Иногда его называют круговым определением.

Функции, вызывающие сами себя называются рекурсивными. Пример рекурсивной функции – функция fact(), вычисляющая факториал целого числа.

В языке C/C++ функция может вызывать саму себя.

Примеры работы рекурсии.

Вычисление факториала:

int fact(int n)

{

if (n<=0) return 1;

else return n\*fact(n-1);

}

Вычисление чисел Фибоначчи, которые определяются следующим рекурсивным соотношением: b0=0; b1=1, bn= bn-1+bn-2:

int fibr (int n)

{

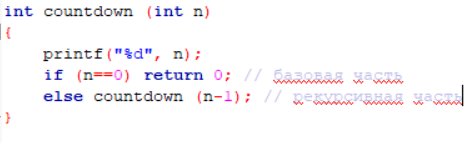
if (n<=1) return n;

else return fibr(n-1) + fibr(n-2);

}

1. **Структура рекурсивного алгоритма. Хранение данных при рекурсии. Возможные ошибки программирования.**

Рекурсивная функция состоит из двух частей: базовой и рекурсивной.



Когда функция вызывает сама себя, новый набор локальных переменных и параметров размещается в памяти в стеке, а код функции выполняется с самого своего начала, причем используются именно эти новые переменные. При рекурсивном вызове функции новая копия ее кода не создается. Новыми являются только значения, которые использует данная функция. При каждом возвращении из рекурсивного вызова старые локальные переменные и параметры извлекаются из стека, и сразу за рекурсивным вызовом возобновляется работа функции. При использовании рекурсивных функций стек работает подобно "телескопической" трубе, выдвигающейся вперед и складывающейся обратно.

Хотя и кажется, что рекурсия предлагает более высокую эффективность, но на самом деле такое бывает достаточно редко. Использование рекурсии в программах зачастую не очень сильно уменьшают их размер кода и обычно только незначительно увеличивает эффективность использования памяти. Кроме того, рекурсивные версии большинства программ могут выполняться несколько медленнее, чем их итеративные варианты, потому что при рекурсивных вызовах функций расходуются дополнительные ресурсы. Кроме того, большое количество рекурсивных вызовов функции может вызвать переполнение стека. Из-за того, что память для параметров функции и локальных переменных находится в стеке и при каждом новом вызове создается еще один набор этих переменных, то для переменных места в стеке может рано или поздно не хватить. Переполнение стека — вот обычная причина аварийного завершения программы, когда функция утрачивает контроль над рекурсивными обращениями.

Главным преимуществом рекурсивных функций является то, что с их помощью упрощается реализация некоторых алгоритмов, а программа становится понятнее. Например, алгоритм быстрой сортировки трудно реализовать итеративным способом. Кроме того, для некоторых проблем, особенно связанных с искусственным интеллектом, больше подходят рекурсивные решения. И наконец, некоторым людям легче думать рекурсивными категориями, чем итеративными.

В тексте рекурсивной функции обязательно должен быть выполнен условный оператор, например if, который при определенных условиях вызовет завершение функции, т.е. возврат, а не выполнит очередной рекурсивный вызов. Если такого оператора нет, то после вызова функция никогда не сможет завершить работы. Распространенной ошибкой при написании рекурсивных функций как раз и является отсутствие в них условного оператора.

1. **Основные возможности работы с динамической памятью**

Указатели используются для динамического выделения памяти компьютера для хранения данных.

Динамическое распределение означает, что программа выделяет память для данных во время своего выполнения.

Основу системы динамического распределения в C составляют функции malloc() и free().

Эти функции работают совместно:

* malloc() выделяет память;
* free() освобождает её.

Прототип функции malloc():

void \*malloc(size\_t size);

Функцию malloc() можно присвоить указателю любого типа.

Функция malloc() возвращает указатель на первый байт области памяти размером *size*, которая была выделена из динамически распределяемой области памяти. Если для удовлетворения запроса в динамически распределяемой области памяти нет достаточного объема памяти, возвращается нулевой указатель. Перед попыткой использовать выделенную память следует проверять, что возвращаемое значение не является нулевым указателем. Попытка использовать нулевой указатель обычно приводит к полному отказу системы.

char \*p;

p = malloc(1000); // выделение 1000 байтов

После присвоения указатель p ссылается на первый из 1000 байтов выделенного участка памяти.

Функция free() противоположна функции malloc() в том смысле, что она возвращает системе участок памяти, выделенный ранее с помощью функции malloc().

Прототип: void free(void *\*p*);

Функция free() возвращает в динамически распределяемую область памяти блок памяти, адресуемый указателем p, после чего эта память становится доступной для выделения в будущем.

Необходимо следить, чтобы free() вызывалась только с указателем, который был ранее получен в результате вызова одной из системных функций динамического распределения. Использование недопустимого указателя при вызове, скорее всего, приведет к разрушению механизма управления памятью и, возможно, вызовет крах системы. При передаче нулевого указателя функция free() не выполняет никакого действия.

1. **Создание динамических массивов и правила работы с ними**

Динамическим называется [массив](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2_(%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)), размер которого может изменяться во время исполнения программы. Возможность изменения размера отличает динамический массив от статического, размер которого задаётся на момент компиляции 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 самом языке С нет динамических массивов, но функции стандартной библиотеки malloc, free и realloc позволяют реализовать массив переменного размера:

int \*mass = (int\*)malloc(**sizeof**(int) \* n); *// Создание массива из n элементов типа int*

...

mass = (int\*)realloc(mas, **sizeof**(int) \* m); *// Изменение размера массива с n на m с сохранением содержимого*

...

free(mas); *// Освобождение памяти после использования массива*

Неудобство данного подхода состоит в необходимости вычислять размеры выделяемой памяти, применять явное преобразование типа и тщательно отслеживать время жизни массива (как и всегда при работе с динамически выделенной памятью в Си).

Начиная с версии стандарта [C99](https://ru.wikipedia.org/wiki/C99) в язык введены массивы переменной длины. Стандарт С99 позволяет в объявлении размера массива использовать любые выражения, в том числе такие, значение которых становится известным только во время выполнения. Объявленный таким образом массив называется *массивом переменной длины*. Однако переменную длину могут иметь только локальные массивы (т.е. видимые в блоке или в прототипе). Приведем пример массива переменной длины:

void f(int dim)

{

char str[dim]; /\* символьный массив переменной длины \*/

/\* ... \*/

}

Здесь размер массива str определяется значением переменной dim, которая передается в функцию f() как параметр. Таким образом, при каждом вызове f() создается массив str разной длины.

Массивы переменной длины добавлены в С99 главным образом для поддержки численных методов обработки данных. В программировании это средство распространено достаточно широко. Однако следует помнить, что стандарт С89 (и некоторые компиляторы C++) не поддерживает массивы переменной длины.

1. **Файловая система изучаемого языка C. Понятие файла. Типы файлов. Потоки и файлы. Стандартные потоки ввода/вывода. Текстовые и бинарные потоки.**

Файловая система языка C предназначена для широкого спектра средств ввода/вывода, включая терминалы, дисководы и принтеры. Файловая система преобразует каждое устройство в некоторую логическую абстракцию, называемую потоком. Все потоки в файловой системе языка C одинаковы, а файлы могут различаться.

Язык C содержит методы для работы с файлами, позволяющие:

* открыть файл;
* использовать специальные функции ввода/вывода для осуществления операций чтения/записи для данного файла.

Файл – это именованный раздел, обычно на диске, для сохранения информации.

На языке С файл – последовательность байтов, каждый из которых считывается в индивидуальном порядке.

Файлом считается все – от файла на диске до дисплея или принтера. Не все файлы имеют одинаковые возможности.

Файлы подразделяют на входные (с исходными данными) и выходными (с результатами работы программы).

Выполнив операцию открытия, поток можно связать с конкретным файлом и использовать для обмена данными с программой. По аналогии с потоками, С обеспечивает два способа представления файлов:

* бинарное (двоичное) – каждый байт файла доступен программе;
* текстовое – то, что видит программа может отличаться от того, что расположено в нем.

Поток – это логическое устройство, которое осуществляет передачу данных от источника к приемнику.

В начале выполнения любой программы, написанной на языке С, автоматически открываются 3 потока:

* stdin (стандартный поток ввода) – для ввода с клавиатуры,
* stdout (стандартный поток вывода) – для вывода на экран,
* stderr (стандартный поток ошибок) – для вывода на экран.

Обычно эти потоки связаны с консолью, но ОС может перенаправлять их на другие устройства.

Стандартные потоки можно применять как файловые указатели любой функции, использующей указатель типа FILE \*fp;

Если файл может поддерживать запрос позиции, при его открытии курсор файла устанавливается в начало. При чтении/записи очередного символа курсор перемещается на одну позицию вперед. При закрытии файла его связь с потоком разрывается. Если файл был открыт для записи, то его содержимое записывается на внешнее устройство. Этот процесс называют очисткой потока.

Существуют два вида потоков: текстовый и бинарный.

Текстовый поток – последовательность символов. Стандарт языка позволяет (но не требует) организовывать потоки в виде строк, заканчивающихся символом перехода. В последней строке символ перехода указывается обязательно.

Бинарный поток – последовательность байтов, однозначно соответствующих последовательности байтов, записанной на внешнем устройстве. Количество записанных/считанных байтов совпадает с количеством байтов на внешнем устройстве. Бинарный поток может содержать дополнительные нулевые байты, количество которых зависит от конкретной реализации.

1. **Основные этапы работы с файлами на С. Указатель файла. Функции открытия и закрытия файла. Режимы доступа к файлам**

Этапы работы с файлами:

1. Создать файловый объект (переменную, поток);
2. Выполнить чтение / запись файла;
3. Закрыть файл (очистить поток).

Указатель файла — это то, что соединяет в единое целое всю систему ввода/вывода языка С. Указатель файла — это указатель на структуру типа FILE. Он указывает на структуру, содержащую различные сведения о файле, например, его имя, статус и указатель текущей позиции в начало файла. В сущности, указатель файла определяет конкретный файл и используется соответствующим потоком при выполнении функций ввода/вывода. Чтобы выполнять в файлах операции чтения и записи, программы должны использовать указатели соответствующих файлов. Чтобы объявить переменную-указатель файла, используйте такого рода оператор: FILE \*fp;

Функция fopen() открывает поток и связывает с этим потоком определенный файл. Затем она возвращает указатель этого файла. Чаще всего (а также в оставшейся части этой главы) под файлом подразумевается дисковый файл. Прототип функции fopen() такой:

FILE \*fopen(const char \**имя\_файла*, const char \**режим*);

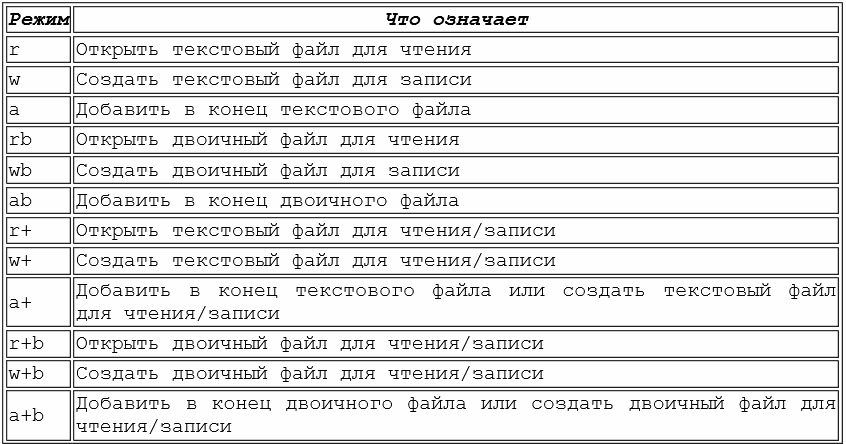
где имя\_файла — это указатель на строку символов, представляющую собой допустимое имя файла, в которое также может входить спецификация пути к этому файлу. Строка, на которую указывает *режим*, определяет, каким образом файл будет открыт.

Функция fclose() закрывает поток, который был открыт с помощью вызова fopen().Функция fclose() записывает в файл все данные, которые еще оставались в дисковом буфере, и проводит, так сказать, официальное закрытие файла на уровне операционной системы. Отказ при закрытии потока влечет всевозможные неприятности, включая потерю данных, испорченные файлы и возможные периодические ошибки в программе. Функция fclose() также освобождает блок управления файлом, связанный с этим потоком, давая возможность использовать этот блок снова. Так как количество одновременно открытых файлов ограничено, то, возможно, придется закрывать один файл, прежде чем открывать другой. Прототип функции fclose() такой:

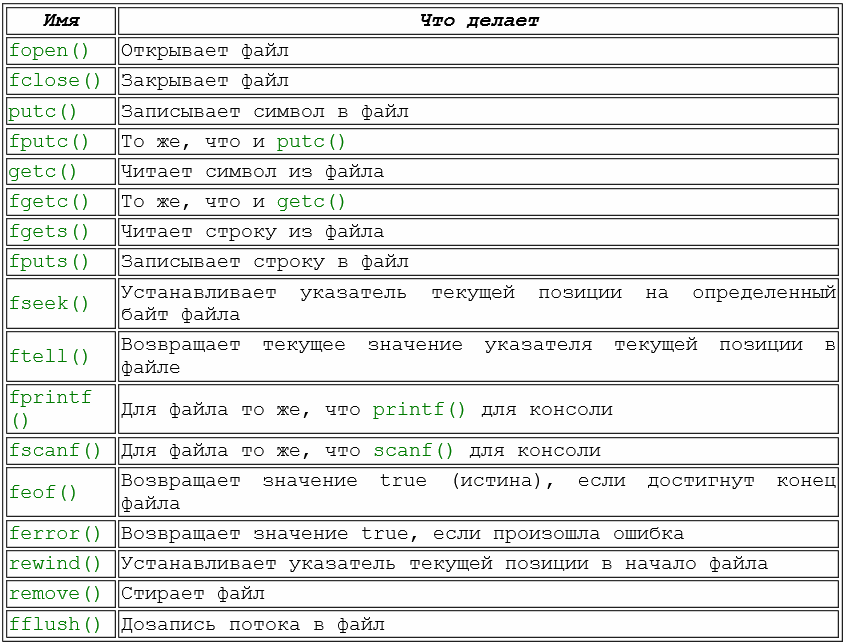
int fclose(FILE \*уф);

где уф — указатель файла, возвращенный в результате вызова fopen(). Возвращение нуля означает успешную операцию закрытия. В случае же ошибки возвращается EOF. Чтобы точно узнать, в чем причина этой ошибки, можно использовать стандартную функцию ferror() (о которой вскоре пойдет речь). Обычно отказ при выполнении fclose() происходит только тогда, когда диск был преждевременно удален (стерт) с дисковода или на диске не осталось свободного места.

Режимы открытия файла:



1. **Наиболее распространенные функции работы с файлами. Макросы для работы с файлами**



В файле stdio.h определены макросы:

* NULL определяет нулевой указатель;
* EOF обычно определяет константу, равную -1. Он задает значение, возвращаемое функцией ввода при попытке прочесть несуществующую запись после конца файла;
* FOPEN\_MAX определяет количество файлов, которые можно открыть одновременно.

Остальные макросы (SEEK\_SET, SEEK\_CUR, SEEK\_END) используются функцией fseek(), осуществляющей прямой доступ к записям файла.

1. **Основные функции записи в файл. Запись символов, строк, форматированный вывод во внешний файл. Запись много байтовых файлов.**

Запись символа

В системе ввода/вывода языка С определяются две эквивалентные функции, предназначенные для вывода символов: putc() и fputc(). Две идентичные функции имеются просто потому, чтобы сохранять совместимость со старыми версиями С.

Функция putc() записывает символы в файл, который с помощью fopen() уже открыт в режиме записи. Прототип этой функции следующий:

int putc(int *ch*, FILE \**уф*);

где уф — это указатель файла, возвращенный функцией fopen(), a ch — выводимый символ. Указатель файла сообщает putc(), в какой именно файл следует записывать символ. Хотя *ch* и определяется как int, однако записывается только младший байт.

Если функция putc() выполнилась успешно, то возвращается записанный символ. В противном же случае возвращается EOF.

Ввод строк: fputs()

Кроме putc(), в языке С также поддерживаются родственная функция fputs(). Она записывает строки в файл, тоже находящийся на диске. Эта функция работает почти как putc(), но записывает не один символ, а целую строку. Прототипы функций fgets() и fputs() следующие:

int fputs(const char \**cmp*, FILE \**уф*);

Функция fputs() пишет в определенный поток строку, на которую указывает *cmp*. В случае ошибки эта функция возвращает EOF.

Функция fwrite()

Для записи данных, тип которых может занимать более 1 байта, в файловой системе языка С имеется функция fwrite(). Эта функция позволяет записать блоки данных любого типа. Прототип следующий:

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

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*. И, наконец, *уф* — это указатель файла, то есть на уже открытый поток.

Функция fwrite() возвращает количество записанных элементов. Если ошибка не произошла, то возвращаемый результат будет равен значению счетчик.

Функция fprinf()

Кроме основных функций вывода, о которых шла речь, в системе вывода языка С также имеется функция fprintf(). Эта функция, за исключением того, что предназначены для работы с файлами, ведет себя точно так же, как и printf(). Прототип функции fprintf() следующий:

int fprintf(FILE \**уф*, const char \**управляющая\_строка*, ...);

где *уф* — указатель файла, возвращаемый в результате вызова fopen(). Операции вывода функция fprintf() и выполняет с тем файлом, на который указывает *уф*.

1. **Основные функции чтения из файла. Чтение символов, строк, чтение форматированных данных. Чтение многобайтовых данных**

Чтение символа

Для ввода символа также имеются две эквивалентные функции: getc() и fgetc(). Обе определяются для сохранения совместимости со старыми версиями С.

Функция getc() записывает символы в файл, который с помощью fopen() уже открыт в режиме для чтения. Прототип этой функции следующий:

int getc(FILE \**уф*);

где *уф*— это указатель файла, имеющий тип FILE и возвращенный функцией fopen(). Функция getc() возвращает целое значение, но символ находится в младшем байте. Если не произошла ошибка, то старший байт (байты) будет обнулен.

Если достигнут конец файла, то функция getc() возвращает EOF. Поэтому, чтобы прочитать символы до конца текстового файла, можно использовать следующий код;

do {

ch = getc(fp);

} while(ch!=EOF);

Однако getc() возвращает EOF и в случае ошибки. Для определения того, что же на самом деле произошло, можно использовать ferror().

Ввод / вывод строк: fputs() и fgets()

Кроме getc(), в языке С также поддерживаются родственная им функции fgets().Она читает строки символов из файла на диске. Эта функция работает почти как getc(), но читают и записывают не один символ, а целую строку. Прототип функци fgets() следующий:

char \*fgets(char \**cmp*, int *длина*, FILE \**уф*);

Функция fgets() читает из определенного потока строку, и делает это до тех пор, пока не будет прочитан символ новой строки или количество прочитанных символов не станет равным *длина*-1. Если был прочитан разделитель строк, он записывается в строку, чем функция fgets() отличается от функции gets(). Полученная в результате строка будет оканчиваться символом конца строки ('0'). При успешном завершении работы функция возвращает *cmp*, а в случае ошибки — пустой указатель (null).

Функция fread()

Для чтения и записи данных, тип которых может занимать более 1 байта, в файловой системе языка С имеется функция: fread(). Эта функция позволяет читать блоки данных любого типа. Прототип следующий:

size\_t fread(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. 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.

Функция fscanf()

Кроме основных функций ввода, о которых шла речь, в системе ввода языка С также имеется функция fscanf(). Эта функция, за исключением того, что предназначены для работы с файлами, ведет себя точно так же, как и scanf(). Прототип функции fscanf() следующий:

int fscanf(FILE \**уф*, const char \**управляющая\_строка*, ...);

где *уф* — указатель файла, возвращаемый в результате вызова fopen(). Операции ввода функции fscanf() выполняет с тем файлом, на который указывает *уф*.

1. **Способы доступа к файлам. Последовательный доступ к файлам. Произвольный доступ к файлам при организации двоичного ввода/вывода**

Существует два способа доступа к файлам последовательный и прямой(произвольный). Последовательный доступ к файлам реализуется последовательным считыванием или записью информации. Прямой доступ к файлам позволяет обратиться к произвольному месту в файле, если ее положение можно вычислить.

При последовательном доступе к файлам, при выполнении операции чтения и записи, указатель на положение в файле смещается на 1. Таким образом происходит перемещение по всему файлу.

При прямом доступе можно выполнять операции ввода/вывода, используя систему ввода/вывода языка С и функцию fseek(), которая устанавливает указатель текущей позиции в файле.

Обычно прямой доступ может потребоваться лишь для двоичных файлов. Причина тут простая – так как в текстовых файлах могут выполняться преобразования символов, то может и не быть прямого соответствия между тем, что находится в файле и тем байтом, к которому нужен доступ. Единственный случай, когда надо использовать fseek() для текстового файла – это доступ к той позиции, которая была уже найдена с помощью ftell(); такой доступ выполняется с помощью макроса SEEK\_SET, используемого в качестве начала отсчета.

Хорошо помните следующее: даже если в файле находится один только текст, все равно этот файл при необходимости можно открыть и в двоичном режиме. Никакие ограничения, связанные с тем, что файлы содержат текст, к операциям прямого доступа не относятся. Эти ограничения относятся только к файлам, открытым в текстовом режиме.

1. **Файлы с произвольным доступом. Функции, реализующие произвольный доступ к файлу**

При прямом доступе можно выполнять операции ввода/вывода, используя систему ввода/вывода языка С и функцию fseek(), которая устанавливает указатель текущей позиции в файле. Вот прототип этой функции:

int fseek(FILE \**уф*, long int *колич\_байт*, int *начало\_отсчета*);

Здесь *уф* — это указатель файла, возвращаемый в результате вызова функции fopen(), *колич\_байт* — количество байтов, считая от *начало\_отсчета*, оно определяет новое значение указателя текущей позиции, а *начало отсчёта* — это один из следующих макросов:

Начало отсчета Макрос

Начало файла SEEK\_SET

Текущая позиция SEEK\_CUR

Конец файла SEEK\_END

Поэтому, чтобы получить в файле доступ на расстоянии *колич\_байт* байтов от начала файла, *начало\_отсчета* должно равняться SEEK\_SET. Чтобы при доступе расстояние отсчитывалось от текущей позиции, используйте макрос SEEK\_CUR, а чтобы при доступе расстояние отсчитывалось от конца файла, нужно указывать макрос SEEK\_END. При успешном завершении своей работы функция fseek() возвращает нуль, а в случае ошибки — ненулевое значение.

Функцию fseek() можно использовать для доступа внутри многих значений одного типа, просто умножая размер данных на номер элемента, который вам нужен.

Текущее значение указателя текущей позиции в файле можно определить с помощью функции ftell(). Вот ее прототип:

long int ftell(FILE \**уф*);

Функция возвращает текущее значение указателя текущей позиции в файле, связанном с указателем файла *уф*. При неудачном исходе она возвращает -1.

Обычно прямой доступ может потребоваться лишь для двоичных файлов. Причина тут простая — так как в текстовых файлах могут выполняться преобразования символов, то может и не быть прямого соответствия между тем, что находится в файле и тем байтом, к которому нужен доступ. Единственный случай, когда надо использовать fseek() для текстового файла — это доступ к той позиции, которая была уже найдена с помощью ftell(); такой доступ выполняется с помощью макроса SEEK\_SET, используемого в качестве начала отсчета.

Хорошо помните следующее: даже если в файле находится один только текст, все равно этот файл при необходимости можно открыть и в двоичном режиме. Никакие ограничения, связанные с тем, что файлы содержат текст, к операциям прямого доступа не относятся. Эти ограничения относятся только к файлам, открытым *в текстовом режиме*.

1. **Потоковый ввод/вывод данных на С++. Операции для работы с потоком. Манипулятор endl.**

В **С++** существует стандартная библиотека классов, ориентированная на организацию потокового ввода-вывода.

**Поток** – это логическое устройство, которое осуществляет передачу

данных от источника к приемнику.

Классы ввода-вывода образуют иерархию по принципу наследования (на каждом следующем уровне иерархии в класс добавляются новые свойства и методы). Базовым в этой иерархии является класс **ios** (исключение составляют лишь классы буферизированных потоков). В классе **ios** объединены базовые данные и методы для ввода-вывода. Прямыми потомками класса **ios** являются классы **istream** и **ostream**.

**istream** – это класс входных потоков;

**ostream** – класс выходных потоков;

**iostream** – класс двунаправленных потоков ввода-вывода.

Потоковые классы, их свойства и методы (данные и функции) становятся видимыми и доступными в программе при подключении библиотеки **iostream.**

Библиотека iostream не только подключает к программе описания классов **ios**, **istream**, **ostream**, **iostream**, но и содержит определения стандартных потоков ввода-вывода:

**cin** – объект класса istream, связанный со стандартным буферизированным входным потоком, направлен от внешнего устройства (по умолчанию – с клавиатуры) в оперативную память;

**cout** – объект класса ostream, связанный со стандартным буферизированным выходным потоком, направлен из оперативной памяти на внешнее устройство (по умолчанию – на экран компьютера);

**cerr** – объект класса ostream, связанный со стандартным небуферизированным выходным потоком (обычно дисплей консоли), в который направляются сообщения об ошибках;

**clog** – объект класса ostream, связанный со стандартным буферизированным выходным потоком (обычно дисплей консоли), в который с буферизацией направляются сообщения об ошибках.

Для работы с потоками применяются следующие операции:

**Операция ввода**класса istream называется извлечением (чтением) данных из потока. Она обозначается с помощью символа операции сдвига вправо **>>**.

**Операция вывода**класса ostream называется вставкой или включением (или записью) данных в поток. Она обозначается с помощью символа операции сдвига влево **<<**.

Конструкции **>>** и **<<** играют роль операции извлечения и вставки по умолчанию только в том случае, если слева от них находятся объекты, соответственно, классов **istream** и **ostream**:

cin **>>** имя\_объекта\_базового\_типа

cout **<<** выражение\_базового\_типа

cerr **<<** выражение\_базового\_типа

clog **<<** выражение\_базового\_типа

Впротивном случае они как обычно обозначают операции сдвига.

В языке C для перехода на новую строку использовали управляющий символ ‘**\n**’

Манипулятор **endl** выполняет:

* перенос строки;
* производит сброс буферов потока вывода,

что повышает надежность программы, но снижает скорость её выполнения.

1. **Флаги форматированного ввода/вывода на С++. Методы и манипуляторы форматирования**

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.

Флаги устанавливают параметры ввода/вывода, которые будут действовать на все последующие операторы до тех пор, пока не будут отменены.

Установка флага вывода:

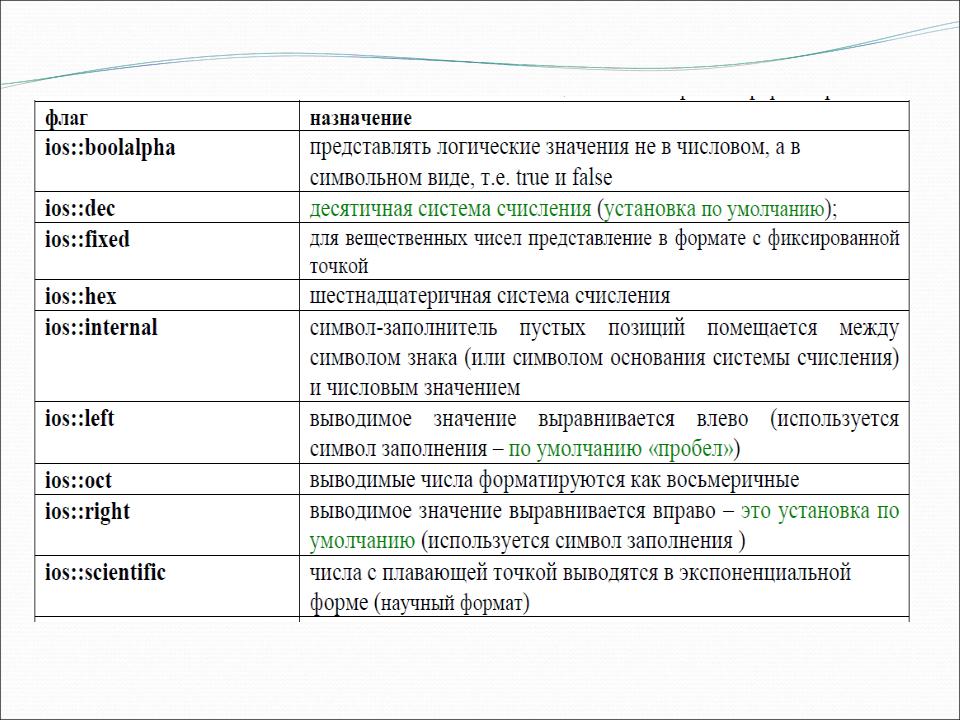
Cout.setf(ios::флаг);

Снятие флага вывода:

Cout.unsetf(ios::флаг);

Установка нескольких флагов:

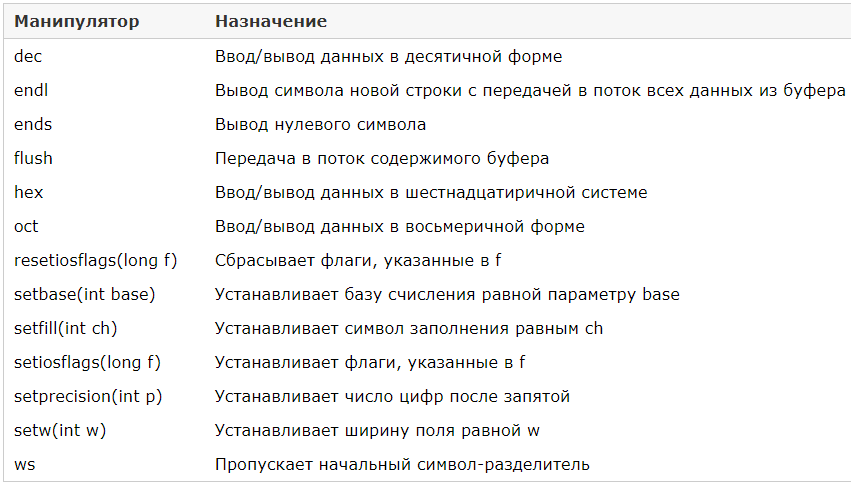
Cout.setf(ios:: флаг1 | ios:: флаг2 | ios:: флаг3);



Флаги удобно использовать в тех случаях, когда следует изменить параметры всех последующих операторов ввода-вывода. Использование большего количества флагов для управления одним оператором ввода-вывода не совсем удобно.

**Манипуляторы** помещаются в операторы ввода/вывода непосредственно перед форматируемым значением.

Использование в программе параметризованных манипуляторов потока возможно только после включения заголовочного файла **iomanip**; непараметризованные манипуляторы потоков могут использоваться и без **iomanip** (например, endl, dec, oct, hex).



Установка ширины поля вывода:

* cout.width(n) – устанавливает ширину поля вывода, равную n позиций.
* cout.precision(m) – определяет m цифр в дробной части числа.

Защита от некорректного ввода:

*поток***.good()** возвращает 1 в случае успешного ввода, 0 – в случае ошибки;

*поток***.clear()** сбрасывает флаг ошибки;

*поток***.ignore()**(кол, символ) удаляет из буфера не более кол символов до символ включительно.

Так как манипуляторы не являются функциями-членами, то они не должны вызываться объектом.

Так как манипуляторы представляют собой функции, то их можно использовать, например, так:

**hex(cout);**

Здесь вызов функции-манипулятора устанавливает шестнадцатеричный формат основания числа для потока *cout*.

Отчасти манипуляторы дополняют функционал, для форматирования ввода/вывода. Но большинство манипуляторов выполняют точно, то же самое, что и функции с флагами форматирования. Есть случаи, когда проще пользоваться флагами или функциями форматирования, а иногда удобнее использовать манипуляторы форматирования. Именно поэтому в С++ предусмотрено несколько средств форматирования ввода/вывода.

1. **Работа с текстовыми и двоичными файлами на С++**

В нашем, человеческом представлении, файлы в компьютере могут хранится в двух вариантах: понятно компьютеру и понятно нам. Понятные нам файлы — это т**екстовые файлы**, понятные компьютеру — это **бинарные файлы**.

В зависимости от задачи бывает удобно выбрать один из этих вариантов. Например, для настроек какой-нибудь игры или программы может быть удобно хранить файлы в текстовом виде, чтобы можно было взять открыть любым текстовым редактором и поменять какую-нибудь настройку быстро и просто. А вот сохранения игр удобнее хранить в бинарных файлах. Как минимум это лёгкая защита от применения читов. Ведь что за игра, если тебе система дала 100 монет, а ты открываешь файл, пишешь 1000.000 монет, покупаешь самое крутое и такой прям классный-преклассный игрок. Выбор двоичного или текстового файла может зависеть от ситуации.

Для работы с файлами необходимо подключить заголовочный файл **<fstream>**.

**Работа с текстовыми файлами**

Работа с текстовым файлом в языке С++ очень напоминает работу с обычным выводом текста на экран. Для вывода на экран мы используем встроенный в язык объект **cout** и операцию **<<**, а в случае работы с файлом мы будем использовать собственное имя объекта и применяемую к этому объекту операцию **<<**.

**Поток** – это логическое устройство, которое осуществляет передачу

данных от источника к приемнику.

Потоки для работы с файлами создаются как объекты следующих классов:

* **ofstream** – для вывода (записи) данных в файл;
* **ifstream** – для ввода (чтения) данных из файла;
* **fstream** – для чтения и для записи данных (двунаправленный обмен).

Создание объекта (файловой системы):

* для записи в файл

**ofstream of;**

* для чтения из файла

**ifstream if;**

* для чтения и записи

**fstream fp;**

Открытие файла:

Поток**.open**(“имя файла”);

fp**.open**(“file.txt”);

Открыть файла для записи можно несколькими способами (пример):

**ofstream out;**

**out.open("tdata.txt");**

**Или**

**ofstream out("tdata.txt");**

**out** — это названный нами объект, выполняющий роль файла. Имя можно давать любое, не противоречащее правилам именования переменных.

**ofstream** — это тип объекта.

* Если файл с указанным именем существует до запуска программы, то после запуска программы всё, что хранил существующий файл, будет стёрто. Будьте внимательны!
* Если вы создаёте файл вовнутрь каких-то папок, то все эти папки должны уже существовать, иначе файл создан не будет. Обратите на это внимание.

Проверка открытия файла:

поток**.is\_open()** – функция возвращает ненулевое значение, если поток связан с файлом, ноль – впротивном случае.

Закрытие файла:

Поток**.close**(“имя файла”);

fp**.close**(“file.txt”);

Пример записи данных в текстовый файл:

**{…**

**ofstream out("C:\\FromC\\myfile.txt");** // Открываем файл для записи

**out << "HelloWorld" <<'\n';** //Записываем первую строчку

**out << "UraRabotaet!!!" <<'\n';** //Записываем вторую строчку

**out.close();** //Закрываем файл

**…}**

**Работа с бинарными (двоичными) файлами**

Для открытия бинарного файла обязательно ставим режим "бинарный файл":

**ofstream out("bdata.txt", ios::binary);**

Закрытие бинарного файла не отличается от текстового:

**out.close();**

* **Запись в двоичный файл:**

*поток***.write**(адрес, кол)

*адрес (char\*)* – начальный адрес записываемой информации

*кол* – количество записываемых байт.

* **Чтение из двоичного файла:**

*поток***.read**(адрес, кол)

*адрес (char\*)* – начальный адрес блока памяти, в который будет помещаться считанная информация

*кол* – количество считываемых байт.

Для того, чтобы проще было работать с бинарными файлами, нужно запомнить две простые истины:

1. **Выводите строку байтов.**
2. **Следите за размерами.**

Когда мы говорим о бинарных данных, тип **char** выступает в роли типа **byte.**

В С++ нету типа **byte**, а вместо типа **byte** используют тип **char**. Согласно правилам языка С++, тип **char** оказывается наименьшей единицей информации.

В наших компьютерах используется такая модель памяти, где одна ячейка занимает 1 байт. А один символ **char** гарантировано, как утверждают в стандарте, должен залезать в эту ячейку. Поэтому, в зависимости от ситуации, тип **char** можно трактовать или как символьный тип, или как байтовый тип. Файлы, хранящие последовательность байтов, называют бинарными файлами. Это связано с тем, что байт делится на биты, а биты представляют собой двоичные цифры. Слово бинарный переводится как двоичный.

Чтобы мы могли записать какое-нибудь значение в бинарном представлении, нам нужно для начала вывести это бинарное представление, а чтобы записалось правильное количество байт, нужно явно указывать это количество. Это выглядит приблизительно следующим образом:

|  |  |
| --- | --- |
| **(char\*)&x** | Так мы делаем строку байтов для того, чтобы отдать потоку, открытому в двоичном режиме |
| **sizeof(x)** | Так мы ограничиваем число уходящих в поток байтов нужным числом |

Пример записи данных в бинарный файл:

**{…**

**const char\* FName = "C:\\FromC\\1.txt";** //Путь к файлу

**int x = 100;** //Переменные для записи

**double y = 5.988;**

/\*РАБОТА С БИНАРНЫМ ФАЙЛОМ\*/

**ofstream out(FName, ios::binary);** //Ставим режим "бинарный файл"

**out.write((char\*)&x, sizeof(x));** //Записываем в файл значение "x"

**out.write((char\*)&y, sizeof(y));** //Записываем в файл значение "y"

**out.close();** //Сохраняем файл

**…}**

Пример извлечения данных из бинарного файла:

**{…**

**const char\* FName = "C:\\FromC\\1.txt";** //Путь к файлу

**int x = 0;**//Переменные

**double y = 0;**

/\*НАЧАЛО РАБОТЫ С ФАЙЛОМ\*/

**ifstream in(FName,ios::binary);**

**in.read((char\*)&x, sizeof(x));** //перенос байтов из файла в "х"

**in.read((char\*)&y, sizeof(y));**//перенос байтов из файла в "y"

**in.close();**

/\*КОНЕЦ РАБОТЫ С ФАЙЛОМ\*/

**…}**

На самом деле для компьютеров нет различия между бинарными файлами и текстовыми файлами, это различие существует только из-за нашего восприятия мира. Вы легко можете прочитать самый простейший текстовый файл при бинарном подходе, при этом вам текст будет точно так же понятен, как он понятен и сам по себе, в блокноте. Текстовые файлы — это только-лишь частный случай бинарных файлов.

1. **Программная реализация алгоритмов работы с файлами. Основные алгоритмы работы с данными файлов. Примеры вычислительных алгоритмов**
2. **Показатели эффективности алгоритмов. Анализ времени выполнения алгоритмов**

Показатели эффективности алгоритмов:

* **Время выполнения** – временная эффективность (количество выполняемых операций; показывает насколько быстро работает алгоритм);
* **Объём потребляемой памяти** – пространственная эффективность (отражает максимальное количество памяти требуемой для выполнения алгоритма);
* **Объём потребляемой электроэнергии**.

Показатели эффективности позволяют сравнивать алгоритмы между собой и оценивать потребности алгоритма в вычислительных ресурсах: процессорном времени, памяти, пропускной способности сети.

**Анализ времени выполнения алгоритмов**

|  |  |
| --- | --- |
| Что влияет на время выполнения алгоритма (программы)?   1. Размер входных данных; 2. Алгоритм; 3. Качество реализации алгоритма на языке программирования; 4. Качество скомпилированного кода; 5. Производительность вычислительной машины.   Для большинства алгоритмов количество выполняемых ими операций напрямую зависит **не** **от** самих **значений** входных данных, а **от** их **размера**. |  |

У каждого алгоритма есть параметры, определяющие размер его входных данных

**Поиск наименьшего элемента в массиве:**

n – количество элементов в массиве

**Алгоритм умножения двух матриц:**

количества строк m и столбцов n в матрицах

**Сравнения двух строк:**

s1, s2 – длина первой и второй строк

**Поиск кратчайшего пути в графе между парой вершин:**

n, m – количество вершин и ребер в графе

*Время выполнения алгоритма* оценивают как **функцию от размера входных данных** – подсчёт количества “базовых операций”, выполняемых им.

**Базовая операция** (требуется один временной шаг – такт вычислительной машины

* +, -, \*, /, =, if, …
* обращение к памяти (чтение, запись)

Циклы и подпрограммы состоят из *последовательности простых операций*.

1. **Асимптотический анализ сложности алгоритмов. Рассматриваемые случаи. О-нотация. Классы сложности алгоритмов.**

Измерение сложности алгоритмов - анализ времени, которое потребуется для обработки данных. При этом точное время мало кого интересует: оно зависит от процессора, типа данных, языка программирования и множества других параметров. Важна лишь асимптотическая сложность, т. е. сложность при стремлении размера входных данных к бесконечности.

Основные оценки роста, встречающиеся в асимптотическом анализе:

*Ο* (*Ο*-большое) – верхняя асимптотическая оценка роста временной функции;

*Ω* (Омега) – нижняя асимптотическая оценка роста временной функции;

*Θ* (Тета) – нижняя и верхняя асимптотические оценки роста временной функции.

Пусть n – величина объема данных. Тогда рост функции алгоритма f(n) можно ограничить функций g(n) асимптотически:

|  |  |
| --- | --- |
| Обозначение | Описание |
| f(n) ∈ Ο(g(n)) | f ограничена сверху функцией g с точностью до постоянного множителя |
| f(n) ∈ Ω(g(n)) | f ограничена снизу функцией g с точностью до постоянного множителя |
| f(n) ∈ Θ(g(n)) | f ограничена снизу и сверху функцией g |

Для нас наибольший интерес представляет Ο-функция. Под фразой «сложность алгоритма есть Ο(f(n))» подразумевается, что с увеличением объема входных данных n, время работы алгоритма будет возрастать не быстрее, чем некоторая константа, умноженная на f(n).

Важные правила асимптотического анализа:

|  |  |
| --- | --- |
| *Ο*(k\*f) = *Ο*(f) | постоянный множитель k (константа) отбрасывается, поскольку с ростом объема данных, его смысл теряется, например: *Ο*(9,1n) = *Ο*(n) |
| *Ο*(f\*g) =*Ο*(f)\**Ο*(g) | оценка сложности произведения двух функций равна произведению их сложностей, например: *Ο*(5n\*n) = *Ο*(5n)\**Ο*(n) = *Ο*(n)\**Ο*(n) = *Ο*(n\*n) = *Ο*(n2) |
| *Ο*(f/g)=*Ο*(f)/*Ο*(g) | оценка сложности частного двух функций равна частному их сложностей, например:*Ο*(5n/n) = *Ο*(5n)/*Ο*(n) = *Ο*(n)/*Ο*(n) = *Ο*(n/n) = *Ο*(1) |
| *Ο*(f+g) | равна доминанте *Ο*(f) и *Ο*(g)оценка сложности суммы функций определяется как оценка сложности доминанты первого и второго слагаемых, например: *Ο*(n5+n10) = *Ο*(n10) |

Порядок роста

Порядок роста описывает то, как сложность алгоритма растет с увеличением размера входных данных. Чаще всего он представлен в виде Ο-нотации (от нем. «Ordnung» — порядок) : Ο(f(x)), где f(x) — формула, выражающая сложность алгоритма. В формуле может присутствовать переменная n, представляющая размер входных данных.

Алгоритм, решая различные задачи размерности n, будет выполнять в каком-то случае наибольшее количество операций, а в каком-то случае наименьшее количество операций. Введем следующие обозначения:

**Худший случай** – наибольшее количество операций, совершаемых алгоритмом А (просто название алгоритма) для решения конкретных проблем размерностью n;

**Лучший случай** – наименьшее количество операций, совершаемых алгоритмом А для решения конкретных проблем размерностью n;

**Средний случай** – среднее количество операций, совершаемых алгоритмом А для решения конкретных проблем размерностью n.



Самое важное здесь то, что O(n) означает, что алгоритм потребует не более n шагов.

1. **Алгоритмы поиска. Методы последовательного поиска. Сложность. Описание: словесное, графическое, на языке программирования.**

Выделяют следующие алгоритмы поиска:

* Последовательный (линейный);
* Последовательный с транспозицией;
* Последовательный с перемещением в начало;
* Бинарный;
* Индексно-последовательный.

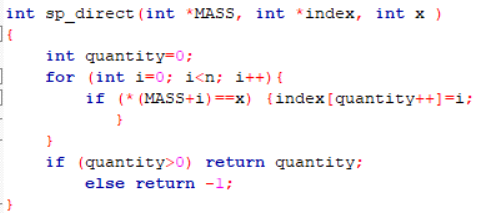
Последовательный (линейный) поиск – это простейший вид поиска заданного элемента на некотором множестве, осуществляемый путем последовательного сравнения очередного рассматриваемого значения с искомым до тех пор, пока эти значения не совпадут.

Идея этого метода заключается в следующем. Множество элементов просматривается последовательно в некотором порядке, гарантирующем, что будут просмотрены все элементы множества (например, слева направо). Если в ходе просмотра множества будет найден искомый элемент, просмотр прекращается с положительным результатом; если же будет просмотрено все множество, а элемент не будет найден, алгоритм должен выдать отрицательный результат.

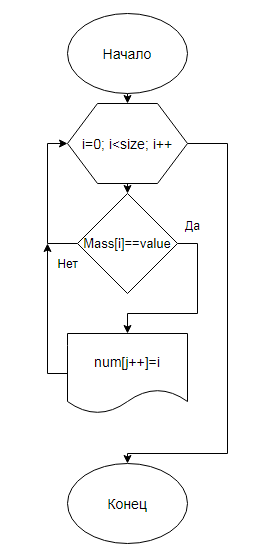
O(n) — линейная сложность

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

Реализация на языке программирования:

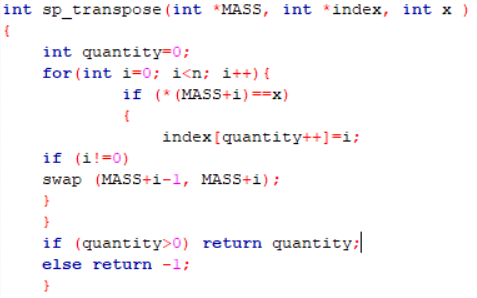


Блок-схема:



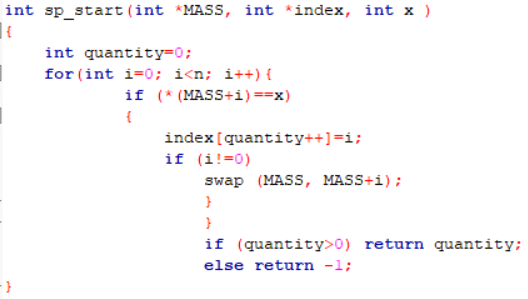
Некоторым улучшением прямого последовательного поиска является метод транспозиции: каждый запрос к записи сопровождается сменой мест этой и предшествующий записи; в итоге наиболее часто используемые записи постепенно перемещаются в начало таблицы; и при последующем обращении к ней запись будет находиться почти сразу.

Реализация на языке программирования:



На идее подобной методу транспозиции основан и метод перемещения в начало: каждый запрос к записи сопровождается её перемещением в начало таблицы; в итоге в начале таблицы оказываются записи, используемые в последнее время.

Реализация на языке программирования:

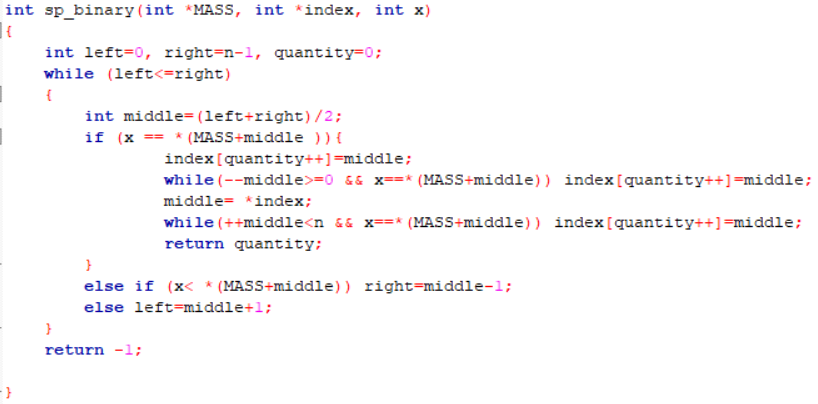


1. **Алгоритм бинарного поиска. Сложность. Описание: словесное, графическое, на языке программирования**

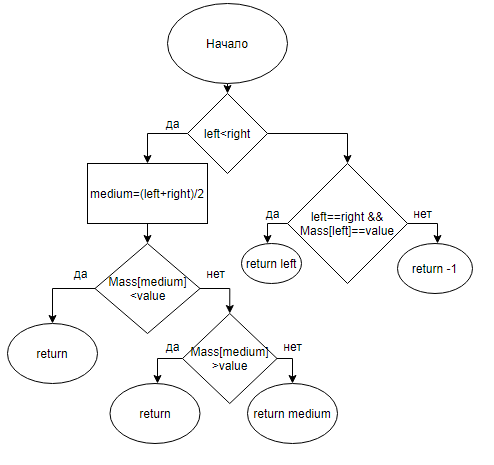
Бинарный поиск производится в упорядоченном массиве. При бинарном поиске искомый ключ сравнивается с ключом среднего элемента в массиве. Если они равны, то поиск успешен. В противном случае осуществляется аналогично в левой или правой частях массива. Алгоритм может быть определен в рекурсивной и нерекурсивной формах.

Логарифмическая сложность O(log n).

Реализация на языке программирования:

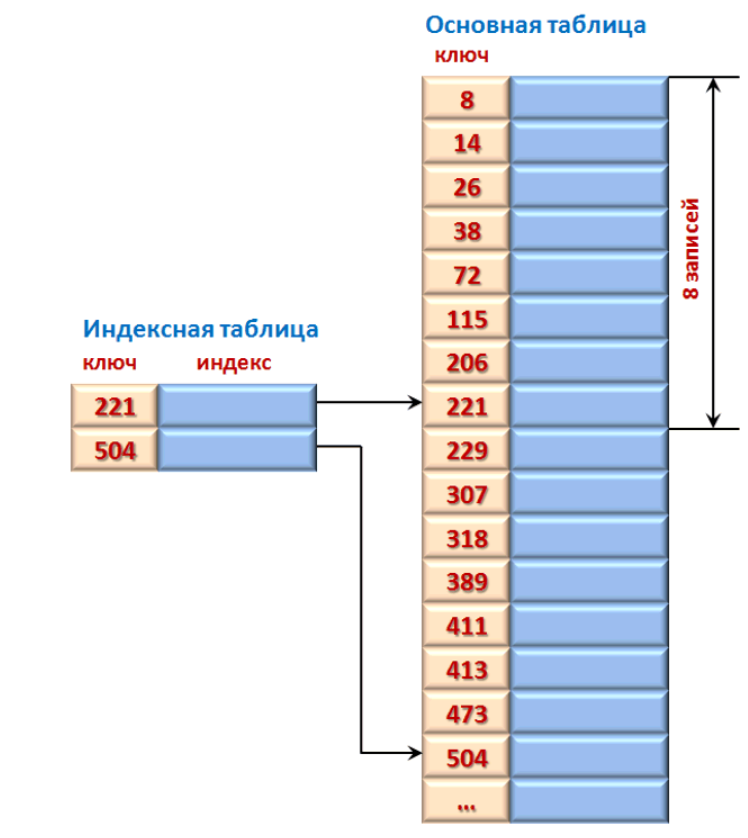


Блок-схема:



1. **Алгоритм индексно-последовательного поиска. Сложность. Описание: словесное и графическое описание, реализация на языке программирования**

В дополнение к отсортированной таблице заводится вспомогательная таблица, называемая индексной. Каждый элемент индексной таблицы состоит из ключа и указателя на запись в основной таблице, соответствующую этому ключу. Элементы в индексной таблице, так же как элементы в основной таблице, должны быть отсортированы по этому ключу. Если индекс имеет размер, составляющий одну восьмую от размера основной таблицы*,* то каждая восьмаязапись основной таблицы будет представлена в индексной таблице.



Достоинство алгоритма заключаются в том, что сокращаются время поиска, так как последовательный поиск первоначально ведется в индексной таблице, имеющей меньший размер, чем основная таблица; когда найден правильный индекс, второй последовательный поиск выполняется по небольшой части записей основной таблицы.

Логарифмическая сложность O(log n).

Реализация на языке программирования:

int index(int key, int \*Mass)

{int i, B[8]={0};

for(i=0; i<N; i+=8)

{

B[i]=Mass[i];

}

for(i=0; i<8; i+=8)

{

if(key==B[i])

{

printf("Index=%d", i); break;

}

else if(key>B[i] && key<B[i+7])

{

for(i=1; i<i+7;i++)

{

if(Mass[i]==key)

{

printf("Index=%d", i); break;

}

}

}

else

{

for(i=9; i<N; i++)

{

if(Mass[i]==key)

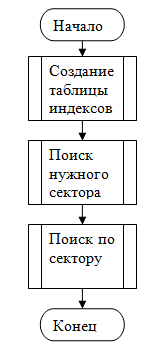
{

printf("Index=%d", i); break;

}

}}}}

Блок-схема:



1. **Алгоритмы сортировки. Классификация и основные виды. Оценка сложности алгоритмов сортировки.**

Алгоритмы сортировки:

* внутренние методы сортировки – элементы размещены в оперативной памяти;
* внешние сортировки – элементы размещены во внешней памяти;
* сортировки на месте – алгоритм, не использующий дополнительной памяти, кроме сортировки массива;
* устойчивая сортировка – алгоритм, не меняющий порядок следования равных элементов.

Виды алгоритмов сортировки:

* Обменные:

– Пузырьком;

– Шейкер-сортировка;

– Быстрая сортировка;

* Выбором:

– Выбором;

– Пирамидальная;

* Вставками:

– Вставками;

– Шилдта;

* Слиянием;
* Без сравнения (подсчетом), гибридные и др.

Алгоритмы устойчивой сортировки:

* Пузырьком;
* Шейкер
* Пирамидальная;
* Быстрая;
* Интроспективная (сочетание быстрой и пирамидальной).

Примером квадратичных и субквадратичных алгоритмов являются:

* сортировка выбором (SelectSort);
* сортировка пузырьком (BubbleSort) и ее улучшения;
* сортировка простыми вставками (InsertSort);
* сортировка Шелла (ShellSort).

Примером логарифмических и линейных алгоритмов являются:

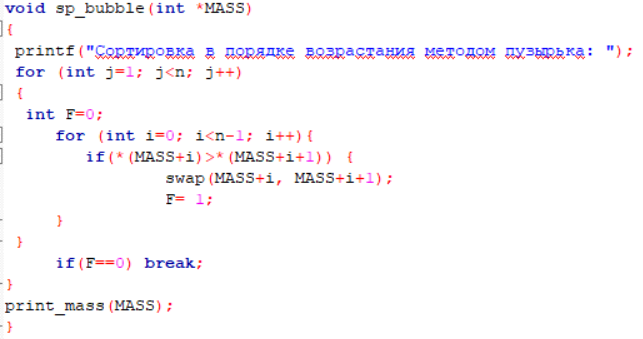
* пирамидальная сортировка (HeapSort);
* быстрая сортировка (QuickSort);
* поразрядная сортировка (RadixSort).

1. **Алгоритмы простых сортировок: пузырьком, выбором, вставками. Словесное и графическое описание, реализация на языке программирования**

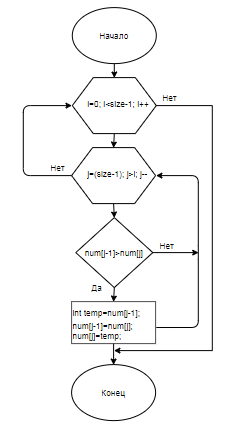
Сортировка пузырьком.

Выполняются повторяющиеся проходы по массиву. При этом элементы последовательно сравниваются попарно и, если порядок в паре неверный выполняется обмен элементов. Проходы по массиву повторяются n-1 раз или пока на очередном проходе не окажется, что обмены больше не нужны, т. е. массив отсортирован.

Реализация на языке программирования:



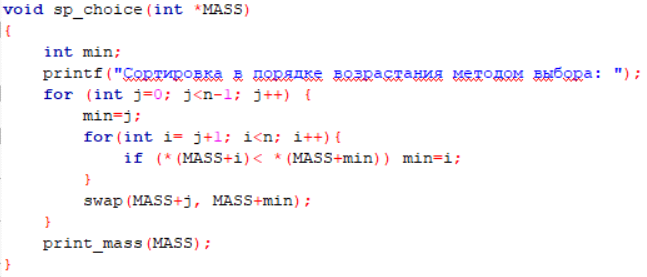
Блок-схема:



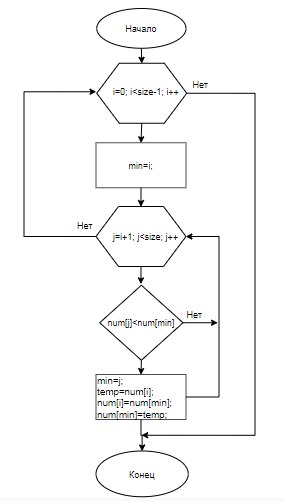
Сортировка выбором.

Находим минимальное значение (номер минимального значения). Выполняем обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции). Сортируем оставшиеся элементы, исключив из рассмотрения уже отсортированные элементы.

Реализация на языке программирования:



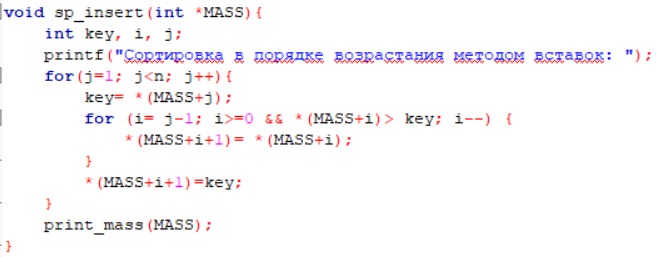
Блок-схема:



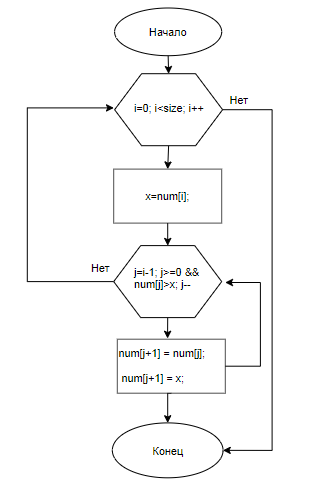
Сортировка вставками.

Сортировка вставками (*Insertion Sort*) — это простой алгоритм сортировки. Суть его заключается в том что, на каждом шаге алгоритма мы берем один из элементов массива, находим позицию для вставки и вставляем. Стоит отметить что массив из 1-го элемента считается отсортированным.

Реализация на языке программирования:



Блок-схема:



1. **Обменные сортировки. Методы улучшения. Словесное описание и реализация на языке программирования**

К обменным сортировкам относятся:

* Пузырьком;
* Шейкер;
* Быстрая

Сортировка пузырьком:

Алгоритм сортировки пузырьком основан на принципе сравнения и обмена пары соседних элементов до тех пор, пока не будут отсортированы все элементы. Как и в методе [прямого выбора](https://prog-cpp.ru/sort-select/), совершаются проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к началу массива.

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

{

for(j=(size-1); j>i; j--)

{

if(Mass[j]<Mass[j-1])

{

temp=Mass[j];

Mass[j]=Mass[j-1];

Mass[j-1]=temp;

}

}

}

Одним из вариантом улучшения данного алгоритма является установление флага, который показывает, что последовательность уже отсортирована. Флаг сокращает время выполнения алгоритма в случае, если массив отсортирован раньше чем через n-1 раз.

Качественно другое улучшение алгоритма можно получить из следующего наблюдения. Хотя легкий пузырек снизу поднимется наверх за один проход, тяжелые пузырьки опускаются со минимальной скоростью: один шаг за итерацию. Так что массив 2 3 4 5 6 1 будет отсортирован за 1 проход, а сортировка последовательности 6 1 2 3 4 5 потребует 5 проходов.

Чтобы избежать подобного эффекта, можно менять направление следующих один за другим проходов. Получившийся алгоритм иногда называют "*шейкер-сортировкой*".

Если в части массива перестановки не происходят, то эта часть уже отсортирована, и ее можно исключить из рассмотрения – границы рабочей части массива устанавливаются в месте последнего обмена. Т. к. минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигает только на одну позицию вправо, то массив просматривается в двух направлениях.

Реализация на языке программирования:

while(left<=right)

{

for(i=size-1; i>=left; i--)

{if(Mass[i-1]>Mass[i])

{

int temp=Mass[i];

Mass[i]=Mass[i-1];

Mass[i-1]=temp;

left++;}

}

for(i=1; i<=right; i++)

{

if(Mass[i-1]>Mass[i])

{

int temp=Mass[i];

Mass[i]=Mass[i-1];

Mass[i-1]=temp;

right--;}

}

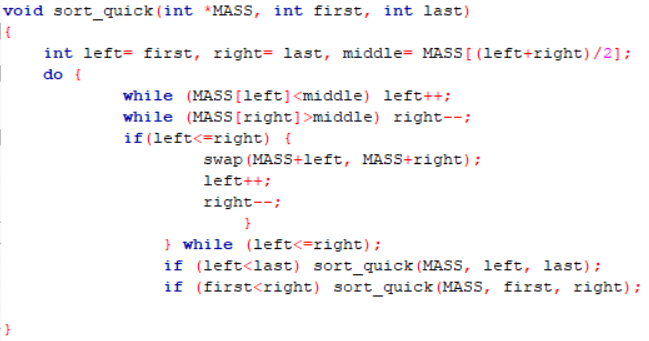
Быстрая сортировка.

Быстрая сортировка – один из самых быстрых универсальных алгоритмов сортировки массивов, O(n log n). В первую очередь производится перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы.

Алгоритм:

Выбрать опорный элемент из массива. Перераспределить элементы массива: элементы меньше опорного помещаются перед ним, больше или равные – после. Рекурсивно применить предыдущие шаги к двум подмассивам (с числом элементов больше 1) слева и справа от опорного элемента.

Реализация на языке программирования:



1. **Алгоритмы сортировки выбором. Методы улучшения. Словесное описание, реализация на языке программирования**

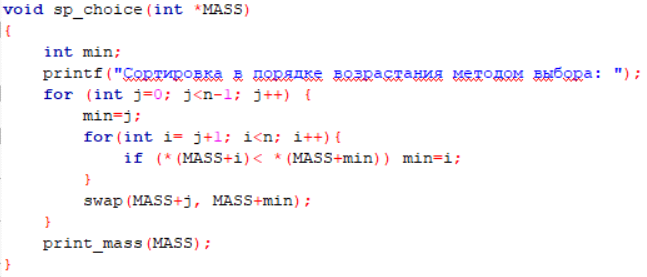
К сортировкам выбором относят:

* Сортировку выбором;
* Пирамидальную сортировку;

Сортировка выбором.

Находим минимальное значение (номер минимального значения). Выполняем обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции). Сортируем оставшиеся элементы, исключив из рассмотрения уже отсортированные элементы.

Реализация на языке программирования:



[Пирамидальная сортировка](https://ru.wikipedia.org/wiki/%D0%9F%D0%B8%D1%80%D0%B0%D0%BC%D0%B8%D0%B4%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0) сильно улучшает базовый алгоритм, используя структуру данных «[куча](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D1%83%D1%8E%D1%89%D0%B5%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE)» для ускорения нахождения и удаления минимального элемента.

Существует также двунаправленный вариант сортировки методом выбора, в котором на каждом проходе отыскиваются и устанавливаются на свои места и минимальное, и максимальное значения.

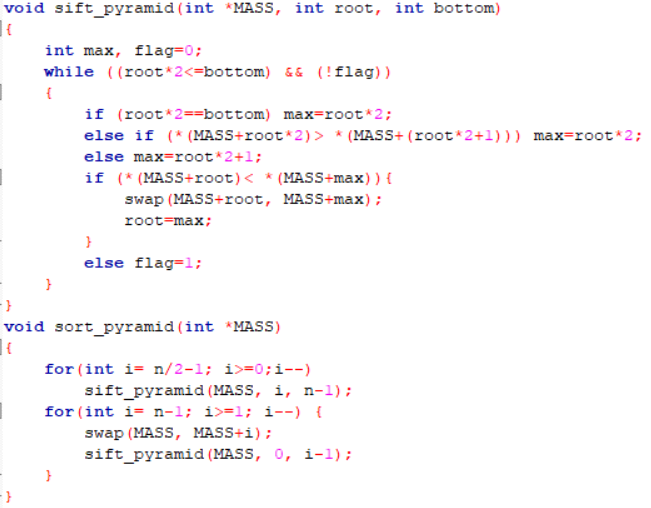
Пирамидальная сортировка:

Общая идея пирамидальной сортировки заключается в том, что сначала строится пирамида из элементов исходного массива, а затем осуществляется сортировка элементов. Выполнение алгоритма разбивается на 2 этапа.

Первый этап – построение пирамиды. Определяем правую часть дерева, начиная с n/2-1 (нижний уровень дерева). Берем элемент левее этой части массива и просеиваем его сквозь пирамиду по пути, где находятся меньшие его элементы, которые одновременно поднимаются вверх; из двух возможных путей выбираете путь через меньший элемент.

Второй этап – сортировка на построенной пирамиде. Берем последний элемент массива в качестве текущего. Меняем верхний (наименьший) элемент массива и текущий местами. Текущий элемент (он теперь верхний) просеиваем сквозь n-1 элементную пирамиду. Затем берем предпоследний элемент и т.д.

Реализация на языке программирования:



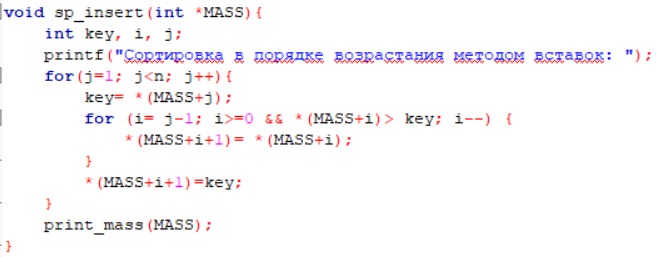
1. **Алгоритм сортировки вставками. Методы улучшения. Словесное описание и реализация на языке программирования.**

Выделяют следующие алгоритмы сортировки вставками:

* Сортировка вставками;
* Сортировка Шелла;
* Сортировка вставками.

Сортировка вставками (*Insertion Sort*) — это простой алгоритм сортировки. Суть его заключается в том что, на каждом шаге алгоритма мы берем один из элементов массива, находим позицию для вставки и вставляем. Стоит отметить что массив из 1-го элемента считается отсортированным.

Реализация на языке программирования:



Алгоритм можно слегка улучшить. Заметим, что на каждом шаге внутреннего цикла проверяются 2 условия. Можно объединить из в одно, поставив в начало массива специальный *сторожевой элемент*. Он должен быть заведомо меньше всех остальных элементов массива.

http://algolist.ru/sort/gif/9.gif

Тогда при j=0 будет заведомо верно a[0] <= x. Цикл остановится на нулевом элементе, что и было целью условия j>=0.

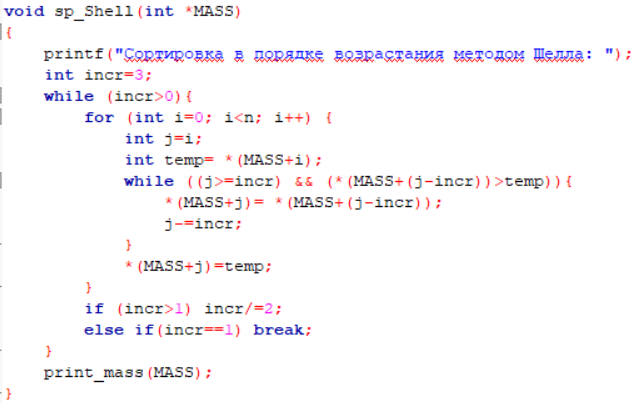
Таким образом, сортировка будет происходить правильным образом, а во внутреннем цикле станет на одно сравнение меньше.

Сортировка Шелла.

Сортировка Шелла – [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), являющийся усовершенствованным вариантом [сортировки вставками](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%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%B0%D0%BC%D0%B8). Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга.

Сначала отдельно группируются и сортируются элементы, отстоящие друг от друга на 4 позиции. Такой процесс называется четвертной сортировкой. После первого прохода элементы перегруппировываются — теперь каждый элемент группы отстоит от другого на 2 позиции — и вновь сортируются (двойная сортировка). На третьем проходе идет обычная сортировка.

Реализация на языке программирования:



1. **Алгоритмы сортировок: подсчетом, расческой, гномья и другие гибридные сортировки.**

Сортировка подсчётом – [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), в котором используется диапазон чисел сортируемого [массива](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2_(%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)) ([списка](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0))) для подсчёта совпадающих элементов. Применение сортировки подсчётом целесообразно лишь тогда, когда сортируемые числа имеют (или их можно отобразить в) диапазон возможных значений, который достаточно мал по сравнению с сортируемым множеством, например, миллион натуральных чисел меньших 1000.

Сортировка подсчетом является специализированным алгоритмом, который работает невероятно быстро, если элементами данных являются целые числа со значениями, занимающими относительно узкий диапазон. Большая скорость сортировки подсчетом достигается за счет того, что при этом не применяются операции сравнения.

Пусть мы имеем массив А целых чисел, состоящий из n элементов. Примем min и max соответственно за минимальный и за максимальный элементы массива А. Тогда значения элементов А[0], А[1], ..., А[n-1] массива А будут принадлежать множеству {min, min+1, ..., max}, мощность которого не превосходит значения m=max–min+1. То есть в массиве А имеется не более mразличных элементов.

Сортировку подсчетом начнем с создания массива вспомогательного массива B размерности m, в котором элемент B[i] будет содержать количество вхождений элемента min+i в массиве А, где i =0, 1, …,m-1. Т. е. B[0] будет содержать количество элементов массива А, равных значению min, элемент B[1] – количество элементов массива А, равных значению min+1 и т.д.

После того как массив B будет заполнен, начинаем перезаписывать элементы в массиве A. Сначала запишем B[0] элементов со значением min, затем B[1] элементов со значением min+1 и т.д. После чего будем иметь отсортированный массив A.

Реализация на языке программирования:

void sp\_count(int \*MASS, int \*sort\_MASS)

{

int k;

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

k=0;

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

if (\*(MASS+j)< \*(MASS+i) || \*(MASS+j)== \*(MASS+i) && j<i) k++;}

\*(sort\_MASS+k)= \*(MASS+i);

}

print\_mass(sort\_MASS);

}

Гномья сортировка — [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), похожий на [сортировку вставками](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%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%B0%D0%BC%D0%B8), но в отличие от последней перед вставкой на нужное место происходит серия обменов, как в [сортировке пузырьком](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). В алгоритме сравниваются рядом стоящие элементы, если они стоят в нужном порядке, тогда мы переходим на следующий элемент массива, если нет, то мы их переставляем и переходим на предыдущий. Нет предыдущего элемента — идём вперед, нет следующего — значит мы закончили. Изначально мы находимся на втором элементе массива.

Реализация на языке программирования:

void sp\_gnom(int \*MASS)

{

printf("Сортировка в порядке возрастания гномьей сортировкой: ");

int i=1 ;

while (i<n){

if (i==0) i=1;

if (\*(MASS+i-1)<= \*(MASS+i))i++;

else {

swap(MASS+i, MASS+i-1);

i--;

}

}

print\_mass(MASS);

}

В «[пузырьке](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)», «[шейкере](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%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D1%88%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%D0%BC)» и «чётно-нечетной» при переборе массива сравниваются соседние элементы. Основная идея «расчёски» в том, чтобы первоначально брать достаточно большое расстояние между сравниваемыми элементами и по мере упорядочивания массива сужать это расстояние вплоть до минимального. Таким образом, мы как бы причёсываем массив, постепенно разглаживая на всё более аккуратные пряди. Первоначальный разрыв между сравниваемыми элементами лучше брать с учётом специальной величины, называемой фактором уменьшения, оптимальное значение которой равно примерно 1,247. Сначала расстояние между элементами равно размеру массива, разделённого на фактор уменьшения (результат округляется до ближайшего целого). Затем, пройдя массив с этим шагом, необходимо поделить шаг на фактор уменьшения и пройти по списку вновь. Так продолжается до тех пор, пока разность индексов не достигнет единицы. В этом случае массив досортировывается обычным пузырьком.

Оптимальное значение фактора уменьшения 1,247...=, где e – основание натурального логарифма, а ф – золотое сечение.

Реализация на языке программирования:

void sp\_comb(int \*MASS)

{

printf("Сортировка в порядке возрастания методом расчески");

if (MASS && n)

{

int jump=n, i;

int swapped=1;

while (jump>1 || swapped)

{

if (jump>1)

jump/=1.24733;

swapped=0;

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

{

if (\*(MASS+i+jump)<\*(MASS+i)) {

swap(MASS+i, MASS+i+jump);

swapped=1;

}

}

}

}

print\_mass(MASS);

}

Четно-нечетная сортировка.

Алгоритм.

Заводится флаг, определяющий отсортирован ли массив. В начале итерации ставится в состояние «истина», далее каждый нечётный элемент сверяется с последующим и если они стоят в неправильном порядке (предыдущий больше следующего), то они меняются местами, и флаг ставится в состояние «ложь». То же самое делается с чётными элементами. Алгоритм не прекращает работу, пока флаг не останется в состоянии «истина».

Реализация на языке программирования:

void sp\_even\_odd(int \*MASS)

{

printf("Сортировка в порядке возрастания четной-нечетной сортировкой: ");

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

for (int j=(i%2)? 0 : 1; j+1<n; j+=2){

if (\*(MASS+j)> \*(MASS+j+1)) swap(MASS+j, MASS+j+1);

}

}

print\_mass(MASS);

}

1. **Типы данных пользователя. Понятие структуры. Указатель на структуру. Доступ к полям структуры. Объявление переменной-структуры. Инициализация переменной-структуры**

В языке С имеется пять способов создания пользовательских типов данных. Пользовательские типы данных можно создать с помощью:

* *структуры* — группы переменных, имеющей одно имя и называемой *агрегатным* типом данных. (Кроме того, еще известны термины *соединение* (*compound*) и *конгломерат* (*conglomerate*).);
* *объединения*, которое позволяет определять один и тот же участок памяти как два или более типов переменных;
* *битового поля*, которое является специальным типом элемента структуры или объединения, позволяющим легко получать доступ к отдельным битам;
* *перечисления* — списка поименованных целых констант;
* ключевого слова **typedef**, которое определяет новое имя для существующего типа.

**Структура** — это совокупность переменных, объединенных под одним именем. С помощью структур удобно размещать в смежных полях связанные между собой элементы информации. ***Объявление структуры*** создает шаблон, который можно использовать для создания ее объектов (то есть экземпляров этой структуры). Переменные, из которых состоит структура, называются ***членами***. (Члены структуры еще называются ***элементами*** или ***полями***.)

Как правило, члены структуры связаны друг с другом по смыслу. Например, элемент списка рассылки, состоящий из имени и адреса логично представить в виде структуры. В следующем фрагменте кода показано, как объявить структуру, в которой определены поля имени и адреса. Ключевое слово **struct** сообщает компилятору, что объявляется (еще говорят, "декларируется") структура.

**struct addr**

**{**

**char name[30];**

**char street[40];**

**char city[20];**

**char state[3];**

**unsigned long int zip;**

**};**

Обратите внимание, что **объявление** завершается точкой с запятой, потому что объявление структуры является оператором. Кроме того, ***тег*** структуры **addr** идентифицирует эту конкретную структуру данных и является спецификатором ее типа.

В данном случае *на самом деле никакая переменная не создается*. Всего лишь определяется вид данных. Когда вы объявляете структуру, то определяете агрегатный *тип*, а не переменную. И пока вы не объявите переменную этого типа, то существовать она не будет. Чтобы **объявить переменную** (то есть физический объект) типа **addr**, напишите следующее:

**struct addr addr\_info;**

В этом операторе объявлена переменная типа **addr**, которая называется **addr\_info**. Таким образом, **addr** описывает вид структуры (ее тип), a **addr\_info** является экземпляром (объектом) этой структуры.

Когда объявляется переменная-структура, компилятор автоматически выделяет количество памяти, достаточное, чтобы разместить все ее члены. На рис.1 показано, как **addr\_info** размещена в памяти; в данном случае предполагается, что целые переменные типа **long** занимают по 4 байта.

|  |
| --- |
| **+------------------------------------------+**  **|Name (имя) 30 байт |**  **+------------------------------------------+**  **+-------------------------------------------------+**  **|Street (улица) 40 байт |**  **+-------------------------------------------------+**  **+-----------------------------------+**  **|City (город) 20 байт |**  **+-----------------------------------+**  **+---------------------+**  **|State (штат) 3 байта |**  **+---------------------+**  **+----------------------------+**  **|Zip (код) 4 байта |**  **+----------------------------+** |
| *Рис.1. Расположение в памяти структуры***addr\_info** |

Одновременно с объявлением структуры можно объявить одну или несколько переменных. Например,

**struct addr {**

**char name[30];**

**char street[40];**

**char city[20];**

**char state[3];**

**unsigned long int zip;**

**} addr\_info, binfo, cinfo;**

**определяет тип структуры**, называемый addr, и объявляет переменные этого типа **addr\_info**, **binfo** и **cinfo**. Важно понимать, что каждая переменная-структура содержит собственные копии членов структуры. Например, поле **zip**в**binfo** отличается от поля **zip**в**cinfo**. Изменения в **zip**из**binfo** не повлияют на содержимое поля **zip**, находящегося в **cinfo**.

Если нужна только одна переменная-структура, то ***тег*** структуры является лишним. В этом случае наш пример объявления можно переписать следующим образом:

**struct {**

**char name[30];**

**char street[40];**

**char city[20];**

**char state[3];**

**unsigned long int zip;**

**} addr\_info;**

В этом случае объявляется одна переменная с именем **addr\_info**, причем ее поля указаны в структуре, которая предшествует этому имени.

Общий вид **объявления** структуры такой:

**struct *тег* {**

***тип имя-члена;***

***тип имя-члена;***

***тип имя-члена;***

***.***

***.***

***.***

**} *переменные-структуры;***

причем ***тег*** или ***переменные-структуры*** могут быть пропущены, но только не оба одновременно.

**Инициализация переменных-структур**

Может выполняться непосредственно за объявлением переменных в виде списка начальных значений:

**struct point {**

**char name;**

**double x;**

**double y; } t={‘O’,0,0};**

1. **Доступ к членам структуры. Массив переменных структуры. Указатель на структуру. Доступ к полям структуры посредством указателя.**

Доступ к членам структуры

Доступ к отдельным членам структуры осуществляется с помощью оператора **.** (который обычно называют *оператором точка* или *оператором доступа к члену структуры*). Например, в следующем выражении полю zip в уже объявленной переменной-структуре addr\_info присваивается значение ZIP-кода, равное 12345:

addr\_info.zip = 12345;

Этот отдельный член определяется именем объекта (в данном случае addr\_info), за которым следует точка, а затем именем самого этого члена (в данном случае zip). В общем виде использование оператора точка для доступа к члену структуры выглядит таким образом:

*имя-объекта.имя-члена*

Поэтому, чтобы вывести ZIP-код на экран, напишите следующее:

printf("%d", addr\_info.zip);

Будет выведен ZIP-код, который находится в члене zip переменной-структуры addr\_infо.

Точно так же в вызове gets() можно использовать массив символов addr\_infо.name:

gets(addr\_info.name);

Таким образом, в начало name передается указатель на символьную строку.

Так как name является массивом символов, то чтобы получить доступ к отдельным символам в массиве addr\_info.name, можно использовать индексы вместе с name. Например, с помощью следующего кода можно посимвольно вывести на экран содержимое addr\_info.name:

for(t=0; addr\_info.name[t]; ++t)

putchar(addr\_info.name[t]);

Обратите внимание, что индексируется именно name (а не addr\_info). Помните, что addr\_info — это имя всего объекта-структуры, a name — имя элемента этой структуры. Таким образом, если требуется индексировать элемент структуры, то индекс необходимо указывать после имени этого элемента.

Массивы структур

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. Например, чтобы объявить 100-элементный массив структур типа addr, который был определен ранее, напишите следующее:

struct addr addr\_list[100];

Это выражение создаст 100 наборов переменных, каждый из которых организован так, как определено в структуре addr.

Чтобы получить доступ к определенной структуре, указывайте имя массива с индексом. Например, чтобы вывести ZIP-код из третьей структуры, напишите следующее:

printf("%d", addr\_list[2].zip);

Как и в других массивах переменных, в массивах структур индексирование начинается с 0.

Для справки: 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. Таким образом, в результате выполнения следующего выражения первому символу члена name, находящегося в третьей структуре из addr\_list, присваивается значение 'X'.

addr\_list[2].name[0] = 'X';

Указатели на структуры

В языке С указатели на структуры также официально признаны, как и указатели на любой другой вид объектов. Однако указатели на структуры имеют некоторые особенности, о которых и пойдет речь.

Объявление указателя на структуру

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;

Использование указателей на структуры

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. В этой главе рассматривается первый случай.

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. (Вспомните, что аргументы передаются функциям через стек.) 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 проблему? Надо передавать не саму структуру, а указатель на нее.

Когда функции передается указатель на структуру, то в стек попадает только адрес структуры. В результате вызовы функции выполняются очень быстро. В некоторых случаях этот способ имеет еще и второе преимущество: передача указателя позволяет функции модифицировать содержимое структуры, используемой в качестве аргумента.

Чтобы получить адрес переменной-структуры, необходимо перед ее именем поместить оператор &. Например, в следующем фрагменте кода

struct bal {

float balance;

char name[80];

} person;

struct bal \*p;/\* объявление указателя на структуру \*/

адрес структуры person можно присвоить указателю p:

p = &person;

Чтобы с помощью указателя на структуру получить доступ к ее членам, необходимо использовать оператор стрелка **->**. Вот, например, как можно сослаться на поле balance:

p->balance;

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.

1. **Понятие объединения. Объявление и использование объединений. Назначение, объявление и использование перечислений.**

Объединение — это место в памяти, которое используется для хранения переменных, разных типов. Объединение дает возможность интерпретировать один и тот же набор битов не менее, чем двумя разными способами. Когда переменная объявляется с ключевым словом union, компилятор автоматически выделяет столько памяти, чтобы в ней поместился самый большой член нового объединения.

Объявление объединения:

union имя\_объединения {

набор\_полей; }

Для получения доступа к члену объединения используется тот же синтаксис, что и для структур: операторы точки и стрелки. При работе непосредственно с объединением следует пользоваться точкой. А при получении доступа к объединению с помощью указателя нужен оператор стрелка. Например, чтобы присвоить целое значение 10 элементу i из cnvt, напишите

cnvt.i = 10;

В следующем примере функции func1 передается указатель на cnvt:

void func1(union u\_type \*un)

{

un->i = 10; /\* присвоение cnvt значение 10 с помощью указателя \*/

}

Объединения часто используются тогда, когда нужно выполнить специфическое преобразование типов, потому что хранящиеся в объединениях данные можно обозначать совершенно разными способами. Например, используя объединения, можно манипулировать байтами, составляющими значение типа double, и делать так, чтобы менять его точность или выполнять какое-либо необычное округление.

Чтобы получить представление о полезности объединений в случаях, когда нужны нестандартные преобразования типа, подумайте над проблемой записи целых значений типа short в файл, который находится на диске.

В стандартной библиотеке языка С не определено никакой функции, специально предназначенной для выполнения этой записи.

Хотя данные любого типа можно записывать в файл, пользуясь функцией fwrite(), но было бы нерационально применять этот способ для такой простой операции, как запись на диск целых значений типа short, так как получится чрезмерный перерасход ресурсов. А вот, используя объединение, можно легко создать функцию putw(), которая по одному байту будет записывать в файл двоичное представление целого значения типа short. (В этом примере предполагается, что такие значения имеют длину 2 байта каждое.)

Перечисление — это набор именованных целых констант. Перечисления довольно часто встречаются в повседневной жизни. Вот, например, перечисление, в котором приведены названия монет, используемых в Соединенных Штатах:

penny (пенни, монета в один цент), nickel (никель, монета в пять центов), dime (монета в 10 центов), quarter (25 центов, четверть доллара), half-dollar (полдоллара), dollar (доллар)

Перечисления определяются во многом так же, как и структуры; началом объявления перечислимого типа служит ключевое слово enum. Перечисление в общем виде выглядит так:

enum тег {список перечисления} список переменных;

Здесь тег и список переменных не являются обязательными. (Но хотя бы что-то одно из них должно присутствовать.) Следующий фрагмент кода определяет перечисление с именем coin (монета):

enum coin { penny, nickel, dime, quarter,

half\_dollar, dollar};

Тег перечисления можно использовать для объявления переменных данного перечислимого типа. Вот код, в котором money (деньги) объявляется в качестве переменной типа coin:

enum coin money;

С учетом этих объявлений совершенно верными являются следующие операторы:

money = dime;

if(money==quarter) printf("Денег всего четверть доллара.\n");

Главное, что нужно знать для понимания перечислений — каждый их элемент представляет целое число. В таком виде элементы перечислений можно применять везде, где используются целые числа. Каждому элементу дается значение, на единицу большее, чем у его предшественника. Первый элемент перечисления имеет значение 0. Поэтому, при выполнении кода

printf("%d %d", penny, dime);

на экран будет выведено 0 2.

Однако для одного или более элементов можно указать значение, используемое как инициализатор. Для этого после перечислителя надо поставить знак равенства, а затем — целое значение. Перечислителям, которые идут после инициализатора, присваиваются значения, большие предшествующего. Например, следующий код присваивает quarter значение 100:

enum coin { penny, nickel, dime, quarter=100,

half\_dollar, dollar};

вот какие значения появились у этих элементов:

penny 0

nickel 1

dime 2

quarter 100

half\_dollar 101

dollar 102

Относительно перечислений есть одно распространенное, но ошибочное мнение. Оно состоит в том, что их элементы можно непосредственно вводить и выводить. Это не так. Например, следующий фрагмент кода не будет выполняться так, как того ожидают многие неопытные программисты:

/\* этот код работать не будет \*/

money = dollar;

printf("%s", money);

Здесь dollar — это имя для значения целого типа; это не строка. Таким образом, попытка вывести money в виде строки по существу обречена. По той же причине для достижения нужных результатов не годится и такой код:

/\* этот код неправильный \*/

strcpy(money, "dime");

То есть строка, содержащая имя элемента, автоматически в этот перечислитель не превратится.

На самом же деле создавать код для ввода и вывода элементов перечислений — это довольно-таки скучное занятие (но его можно избежать лишь тогда, когда будет достаточно именно целых значений этих перечислителей). Например, чтобы выводить название монеты, вид которой находится в money, потребуется следующий код:

switch(money) {

case penny: printf("пенни");

break;

case nickel: printf("никель");

break;

case dime: printf("монета в 10 центов");

break;

case quarter: printf("четверть доллара");

break;

case half\_dollar: printf("полдоллара");

break;

case dollar: printf("доллар");

}

Иногда можно объявить строчный массив и использовать значение перечисления как индекс при переводе этого значения в соответствующую строку. Например, следующий код также выводит нужную строку:

char name[][12]={

"пенни",

"никель",

"монета в 10 центов",

"четверть доллара",

"полдоллара",

"доллар"

};

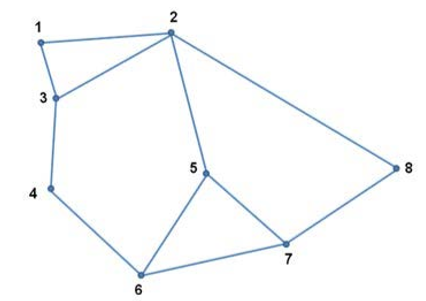
printf("%s", name[money]);

Конечно, он будет работать только тогда, когда не инициализирован ни один из элементов перечисления, так как строчный массив должен иметь индекс, который начинается с 0 и возрастает каждый раз на 1.

Так как при операциях ввода/вывода необходимо специально заботиться о преобразовании перечислений в их строчный эквивалент, который можно легко прочитать, то перечисления полезнее всего именно в тех процедурах, где такие преобразования не нужны. Например, перечисления часто применяются, чтобы определить таблицы соответствия символов в компиляторах.

1. **Понятие графа. Элементы графа. Неориентированный, ориентированный, взвешенный, невзвешенный граф. Правила обхода графа. Матрица смежности**

Граф − это множество точек (вершин), некоторые из которых соединены линиями (ребрами).



Вершины графа можно именовать или нумеровать. Для программирования предпочтительнее нумерация.

Неориентированный граф − граф, в котором ребра не имеют направлений.

Неориентированный граф – пара G= (V, E) из конечного множества вершин V и множества ребер E, элементами которого являются пары вершин графа G.



Ориентированный граф − граф, в котором ребра (дуги) имеют направления.

Ориентированный граф (орграф) − это пара G= (V, E) из конечного множества вершин V и множества дуг E, элементами которого являются упорядоченные пары вершин графа G.

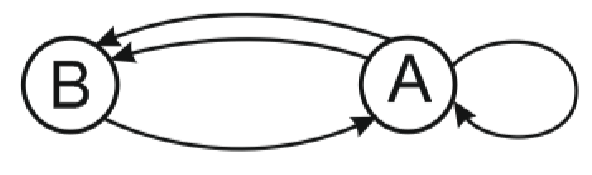


В ориентированном графе разрешаются:

• петли − дуга из вершины в саму себя

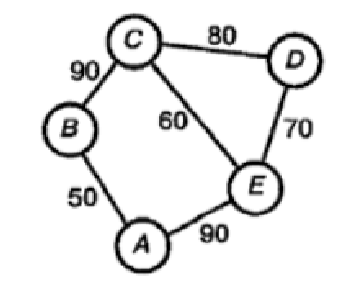
• кратные дуги − несколько дуг из одной вершины в другую

• встречные дуги (например, из вершины А в В и из вершины В в А)

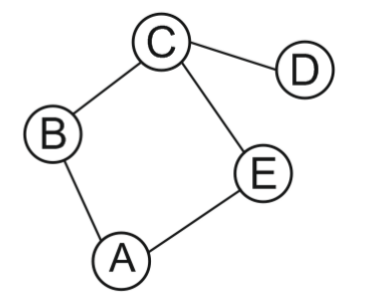


Различают:

* Взвешенный граф − граф, каждое ребро которого имеет свой вес − некоторое число, поставленное ему в соответствие.



* Невзвешенный граф - граф, ребра которого не имеют веса или имеют одинаковый вес.



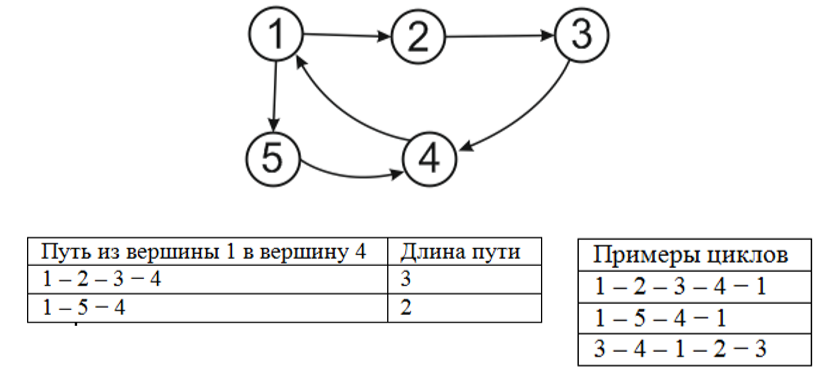
Элементы теории графов:

* + Путь в графе − конечная последовательность вершин (не обязательно различных), в которой каждые две соседние вершины соединены ребром;
  + Цикл − путь, в котором начальная и конечная вершины совпадают;
  + Длина пути − число ребер, по которым проходит путь (для невзвешенного графа), или сумма весов ребер взвешенного графа, по которым проходит путь (маршрут);
  + Смежные вершины − вершины, соединённые ребром.

Обход графа – это процедура перебора (посещения) всех вершин.

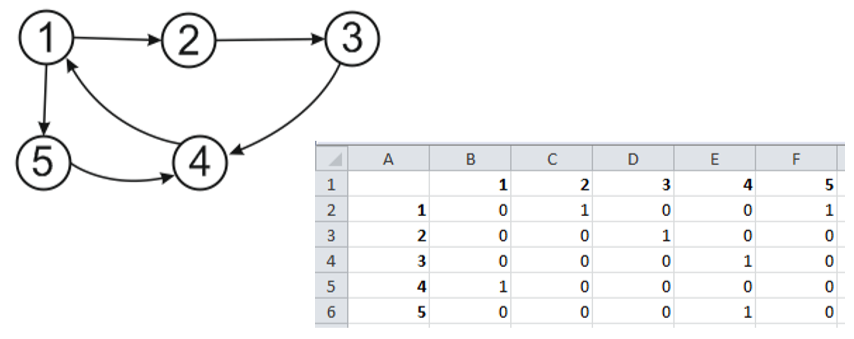
Пример:

* + Найти путь и его длину из вершины 1 в вершину 4.
  + Найти циклы из вершин 1 и 3

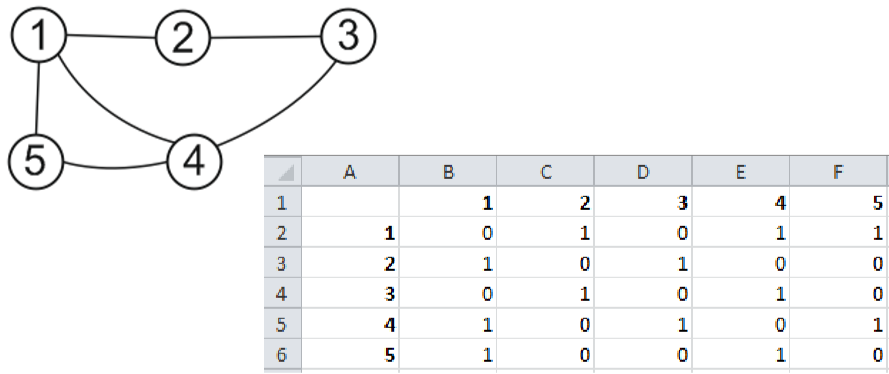


Матрица смежности — один из способов представления графа в виде матрицы.

Построение матрицы смежности для орграфа:

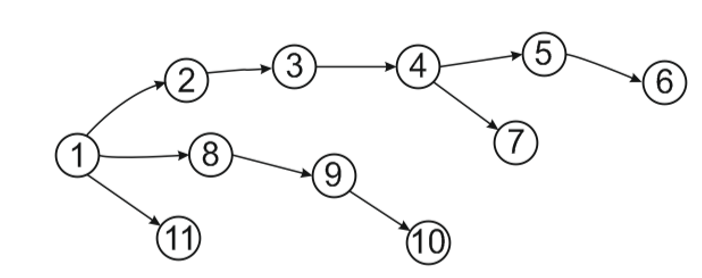


Построение матрицы смежности для неориентированного графа:

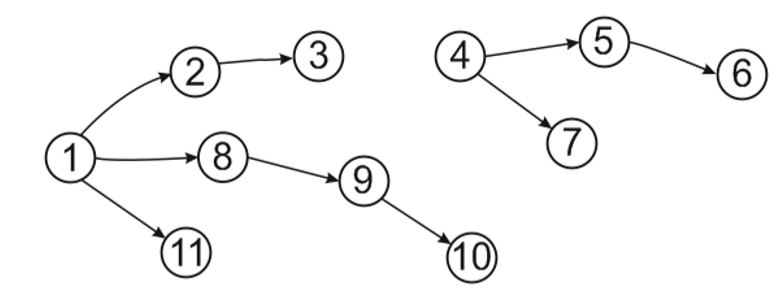


Связный граф − граф, в котором из каждой вершины существует путь в любую другу.

Дерево − связанный граф, в котором не существует циклов.

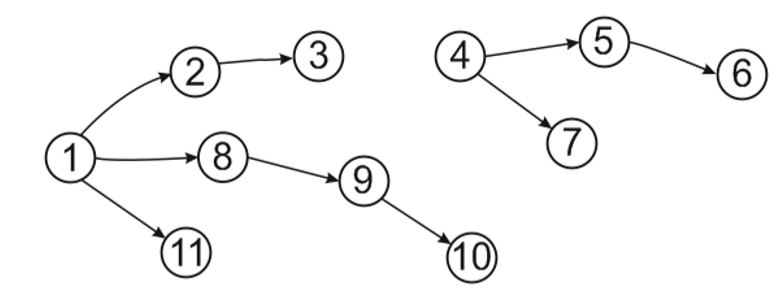


Несвязный граф - граф, в котором существует пара вершин, от одной из которых не существует пути до другой.

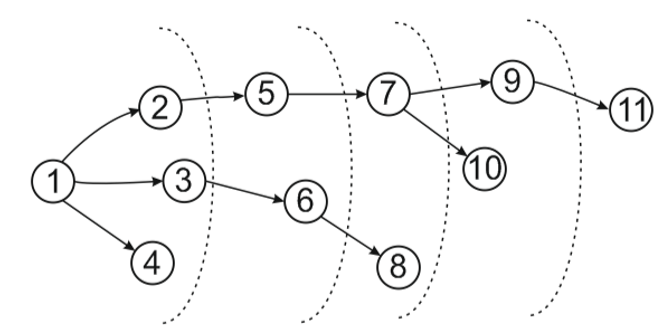


Правила обхода графа:

* В глубину − удобно описывать с помощью структуры данных «стек».



* В ширину − удобно описывать с помощью структуры данных «очередь». Обход орграфа в ширину называют также волновым алгоритмом



Обход невзвешенного графа в глубину с помощью структуры данных «стек»:

Стек (stack − стопка) − это абстрактная структура данных с последовательным доступом к элементам по принципу LIFO, Last In − First Out, «последним пришел, первым обслужен». Добавление элемента возможно лишь в конец стека, выборка — только из конца стека.

Пример − железнодорожный тупик Правила обхода: 1) перебирать соседей от меньшего номера к большему 2) перебирать соседей от большего номера к меньшему

Обход невзвешенного графа в ширину с помощью структуры данных «очередь»:

Очередь (queue) − это абстрактная структура данных с последовательным доступом к элементам по принципу FIFO (First In − First Out) − «первым пришел, первым обслужен». Добавление элемента возможно лишь в конец очереди, выборка − только из начала очереди, при этом выбранный элемент из очереди удаляется.

Очереди широко используются в алгоритмах обработки данных: очереди печати, буфер ввода с клавиатуры, алгоритмы работы с графами.

1. **Динамические структуры данных. Стек, очередь, список. Работа со стеком: добавление, удаление, поиск, сортировка элементов стека.**

Объект данных считается динамической структурой, если его размер, взаимное расположение и взаимосвязи его элементов изменяются в процессе выполнения программы.

Список (list) – последовательность однотипных данных, работа с которыми ведется в оперативной памяти. В процессе работы список может изменять свой размер. Наибольшее распространение получили две формы работы со списком – очередь и стек.

Стек (stek) – список с одной точкой входа. Данные добавляются в список и удаляются из него только с одной стороны последовательности (вершины стека). Таким образом реализуется принцип «последний пришел – первым вышел».

Очередь (turn) – список с одной или двумя точками входа. Данные добавляются в конец очереди, а извлекаются из начала очереди. Таким образом реализуется принцип «первый пришел – первый вышел».

Работа со стеком:

Добавление элемента в стек

tlist \*AddStack(tlist \*sp, int inf)

{

tlist \*spt = new tlist;

spt->inf = inf;

spt->a = sp; return spt;

}

Чтение элемента с удалением

tlist \*ReadStackD(tlist \*sp, int &inf)

{

if (sp == NULL) return NULL;

tlist \*spt = sp;

inf= sp->inf;

sp = sp->a;

delete spt;

return sp;

}

Удаление элемента, следующего за текущим

void DelStackAfter(tlist \*sp)

{

if (sp->a == NULL) return ;

tlist \*spt = sp->a;

sp->a = sp->a->a;

delete spt;

}

Добавление элемента в стек после текущего

void AddStackAfter(tlist \*sp, int inf)

{

tlist \*spt = new tlist;

spt->inf = inf;

if (sp->a == NULL) spt->a = NULL;

else spt->a = sp->a;

sp->a = spt;

}

Удаление всего стека

tlist \*DelStackAll(tlist \*sp)

{

tlist \*spt;

int inf;

while(sp != NULL)

{

spt = sp;

inf = sp->inf;

cout << inf << endl;

sp = sp->a;

delete spt;

}

return NULL;

}

Обмен элементов, следующих за текущим

void RevStackAfter(tlist \*sp)

{

tlist \*spt = sp->a->a;

sp->a->a = spt->a;

spt->a = sp->a;

sp->a = spt;

}

Сортировка с обменом адресами методом пузырька

void SortStackAfter(tlist \*sp)

{

if (sp->a->a == NULL) return;

tlist \*spt = NULL, \*spm;

do

{

for (spm=sp; spm->a->a != spt; spm=spm->a)

if (spm->a->inf > spm->a->a->inf) RevStackAfter(spm);

spt = spm->a;

}

while (sp->a->a != spt);

}

Сортировка стека

tlist \*SortStack(tlist \*sp)

{

tlist \*spt = new tlist;

spt->a = sp;

sp = spt;

SortStackAfter(sp);

sp = sp->a;

delete spt;

return sp;

}

Поиск в стеке

tlist \* PoiskStack(tlist \*sp, int x) // Поиск

{

if (sp==NULL) return NULL;

tlist \*spt=sp;

while (spt->inf != x && spt->a != NULL) spt=spt->a;

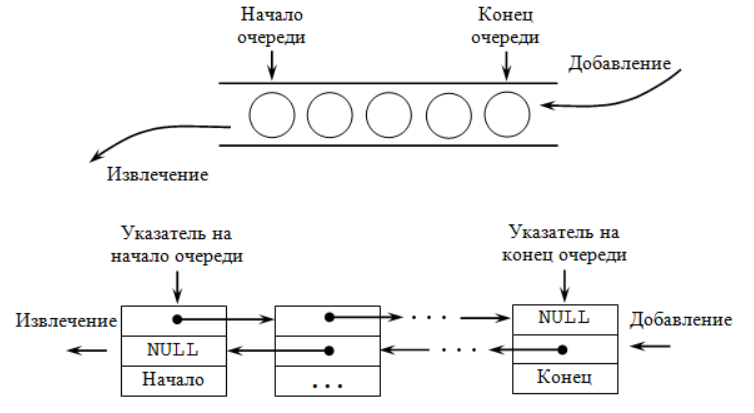
if (spt->inf == x) return spt;

else return NULL;

}

1. **Однонаправленные очереди. Добавление, удаление элементов. Удаление очереди.**

Очередь – это структура данных, представляющая собой последовательность элементов, образованная в порядке их поступления. Каждый новый элемент размещается в конце очереди; элемент, стоящий в начале очереди, выбирается из нее первым. В очереди используется принцип доступа к элементам *FIFO* ( *First Input – First Output*, "первый пришёл – первый вышел"). В очереди доступны два элемента (две позиции): начало очереди и конец очереди. Поместить элемент можно только в конец очереди, а взять элемент только из ее начала. Примером может служить обыкновенная очередь в магазине.



Добавление элемента в очередь

void Addoch(tlist \*\*sp,tlist \*\*spk, int inf)

{

tlist \*spt = new tlist;

spt->inf = inf;

spt->a = NULL;

if (\*spk == NULL) // Если нет элементов

\*sp = \*spk = spt;

else

{

(\*spk)->a = spt;

\*spk = spt;

}

return;

}

Подключение:

sp = spk = NULL;

Addoch(&sp, &spk, информация);

Чтение элемента с удалением

tlist \*ReadochD(tlist \*sp, int &inf)

{

if (sp == NULL) return NULL;

inf = sp->inf;

tlist \*spt = sp;

sp = sp->a;

delete spt;

return sp;

}

Удаление элемента, следующего за текущим

void DelOchAfter(tlist \*sp)

{

if (sp->a == NULL) return;

tlist \*spt = sp->a;

sp->a = sp->a->a;

delete spt;

}

Удаление всей очереди

void DelOchAll(tlist \*\*sp, tlist \*\*spk) // Удаление всей очереди

{

tlist \*spt;

int inf;

while(\*sp != NULL)

{

spt = \*sp;

inf = (\*sp)->inf;

cout << inf << endl;

\*sp = (\*sp)->a;

delete spt;

}

\*spk = NULL;

}

1. **Работа с двусвязными списками. Добавление, удаление элементов**

Двусвязанный список состоит из структур, содержащих поля для хранения адресов предыдущего и последующего элементов. Такая организация позволяет осуществлять перемещение по списку в любом направлении.

Объявление двусвязанной структуры:

struct tlistdbl

{

int inf;

tlistdbl \*left;

tlistdbl \*right;

}

\*sp;

Чтобы избавиться от необходимости написания алгоритмов обработки крайних элементов, создается каркас двусвязанной структуры, состоящий из двух крайних, не имеющих информационной части, элементов. После этого любые добавляемые в список элементы будут являться внутренними элементами этого списка.



Добавление элемента после заданного

void AddochdRight(tlistdbl \*sp, int inf)

{

tlistdbl \*spt = new tlistdbl;

spt->inf = inf;

spt->left = sp;

spt->right = sp->right;

sp->right = spt;

spt->right->left = spt;

return;

}

Добавление элемента перед заданным

void AddochdLeft(tlistdbl \*sp, int inf)

{

tlistdbl \*spt = new tlistdbl;

spt->inf = inf;

spt->left = sp->left;

spt->right = sp;

spt->left->right = spt;

sp->left = spt;

return;

}

Чтение и удаление элемента с адресом

sp int ReadochdD(tlistdbl \*sp)

{

int inf = sp->inf;

sp->left->right = sp->right;

sp->right->left = sp->left;

delete sp;

return inf;

}

Удаление всего списка

void DelOchdAll(tlistdbl \*\*sl, tlistdbl \*\*sr)

{

tlistdbl \*spt = (\*sl)->right;

while(spt != \*sr)

{

cout << ReadochdD(spt) << endl;

spt = (\*sl)->right;

}

delete \*sl;

\*sl = NULL;

delete \*sr;

\*sr = NULL;

return;

}

1. **Сортировка слиянием двусвязного списка**

Разбиение списка на 2 списка

void div2Ochd(tlistdbl \*sl, tlistdbl \*sr,tlistdbl \*\*slL, tlistdbl \*\*srL,tlistdbl \*\*slR, tlistdbl \*\*srR)

{

NewOchd(slL,srL);

NewOchd(slR,srR);

tlistdbl \*spt = sl->right;

while(spt != sr)

{

AddochdLeft(\*srL, ReadochdD(spt));

spt = sl->right;

if (spt != sr)

{

AddochdLeft(\*srR, ReadochdD(spt));

spt = sl->right;

}

}

delete sl;

delete sr;

}

Слияние двух отсортированных списков

void slipOchd(tlistdbl \*\*sl, tlistdbl \*\*sr,tlistdbl \*slL, tlistdbl \*srL,tlistdbl \*slR, tlistdbl \*srR)

{

NewOchd(sl,sr);

tlistdbl \*sptL = slL->right;

tlistdbl \*sptR = slR->right;

while ((sptL != srL) && (sptR != srR))

{

if (sptL->inf < sptR->inf)

{

AddochdLeft(\*sr, ReadochdD(sptL));

sptL = slL->right;

}

else

{

AddochdLeft(\*sr, ReadochdD(sptR));

sptR = slR->right;

}

}

while (sptL != srL)

{

AddochdLeft(\*sr, ReadochdD(sptL));

sptL = slL->right;

}

delete slL;

delete srL;

while (sptR != srR)

{

AddochdLeft(\*sr, ReadochdD(sptR));

sptR = slR->right;

}

delete slR;

delete srR;

}

Сортировка

void SotrSlipOchd(tlistdbl \*\*sl, tlistdbl \*\*sr)

{

tlistdbl \*slL, \*srL,\*slR, \*srR;

if ((\*sl)->right->right == \*sr) return;

div2Ochd(\*sl, \*sr, &slL, &srL, &slR, &srR);

SotrSlipOchd(&slL, &srL);

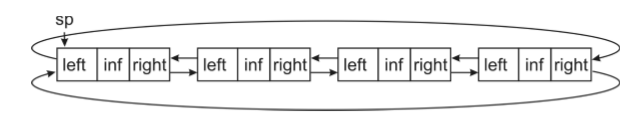
SotrSlipOchd(&slR, &srR);

slipOchd(sl, sr, slL, srL, slR, srR);

}

1. **Работа с двусвязными циклическими списками**

Циклические списки – одно- или двунаправленные очереди, в которых последний элемент указывает на начало очереди. Рассмотрим циклическую двунаправленную очередь. Понятия начала и конца очереди здесь не имеют смысла, достаточно знать адрес любого элемента очереди.



Добавление элемента в циклический список

tlistdbl \*AddochdC(tlistdbl \*sp, int inf) //

{

tlistdbl \*spt = new tlistdbl;

spt->inf = inf;

if (sp == NULL)

{

spt->left = spt;

spt->right = spt;

}

else

{

spt->right = sp->right;

spt->left = sp->right ->left;

sp->right ->left = spt;

sp->right = spt;

}

return spt;

}

Подключение

tlistdbl \*sp = NULL;

sp = AddochdC(sp, информация);

Просмотр списка:

while (условие)

{

cout << sp->inf << endl;

sp = sp ->right;

}

Удаление всего списка

tlistdbl \*DelOchdCAll(tlistdbl \*sp)

{

tlistdbl \*spt;

while(sp->right != sp)

{

cout << sp->inf << endl;

sp->left->right = sp->right;

sp->right->left = sp->left;

spt = sp;

sp = sp->right;

delete spt;

}

cout << sp->inf << endl;

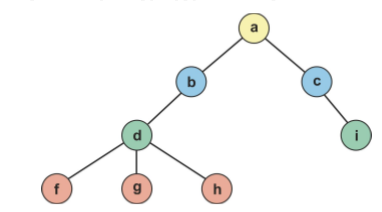
delete sp;

return NULL;

}

1. **Древовидные структуры данных. Использование древовидных структур. Двоичное дерево поиска.**

Рассмотрим древовидную структуру данных



Все данные называются узлами.

Связи между узлами называется ветвями.

Самый верхний узел – корень дерева (а).

Узлы, из которых не выходят связи, – листы дерева ( f, g, h, i ).

Узел, находящийся непосредственно над другим, называется родительским узлом (для узла d узел b является родительским). Узел, находящийся непосредственно ниже, называется дочерним (для узла b узел d является дочерним).

Все узлы, находящиеся выше рассматриваемого, являются его предками (для узла d предки b и a), а все узлы, находящиеся ниже, – потомками (для узла b потомки – d, f, g, h).

Узлы, имеющие одного и того же родителя, называются сестринскими (f, g, h).

Узел, не являющийся листом, называется внутренним (b или d или с или а).

Порядок узла (или степень узла) – количество дочерних узлов (для узла b порядок 1, для узла d порядок 3).

Степень дерева – это максимальный порядок его узлов (рассматриваемое дерево имеет третий порядок). Дерево второй степени называется бинарным или двоичным. Дерево степени три называется троичным деревом.

Глубина узла – число предков плюс единица (например, для узла d глубина равна 3).

Глубина дерева – наибольшая глубина всех узлов (для данного дерева – 4).

Использование древовидных структур.

Для работы с древовидными структурами используется следующая конструкция рекурсивного типа:

struct ttree

{

tinf inf;

ttree \*a1;

ttree \*a2;

…

ttree \*an;

}

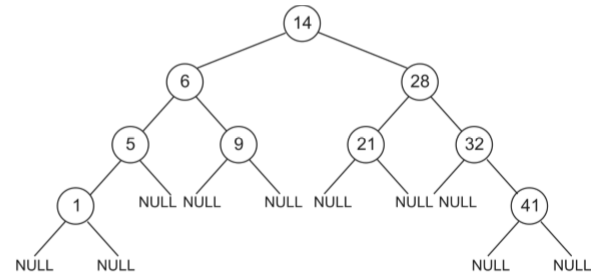
\*proot, \*p;

Непосредственное заполнение даже небольшого дерева требует довольно громоздкой последовательности команд. Поэтому для работы с деревьями используют набор специфических алгоритмов.

Обходом дерева называется последовательное обращение ко всем его узлам.

Двоичное дерево поиска

Если ключевые поля в дереве расположены таким образом, что для любого узла значения ключа у левого преемника меньше, чем у правого, то такое дерево называется двоичным деревом поиска. Предположим, что имеется набор данных, упорядоченных по ключу: key: 1, 5, 6, 9, 14, 21, 28, 32, 41. Для таких данных двоичное дерево поиска выглядит следующим образом



Эффективность поиска информации в такой динамической структуре данных сравнима с эффективностью двоичного поиска в массиве.

Дерево, у которого узлы, имеющие только одну дочь, располагаются не выше двух последних уровней, называется сбалансированным деревом.

Для работы с двоичным деревом поиска используется следующая конструкция рекурсивного типа:

struct ttree

{

int inf;

ttree \*left;

ttree \*right;

}

\*proot;

1. **Понятие хэширования. Схемы хэширования. Хэш-таблица с линейной адресацией.**

Для решения задачи быстрого поиска был придуман алгоритм хеширования (hashing), при котором ключи данных записываются в особую хеш-таблицу. Затем при помощи некой простой функции i = h(key) алгоритм хеширования определяет положение искомого элемента в таблице по значению его ключа.

Рассмотрим пример.

Имеется массив из 7 элементов, значения ключей которых находятся в диапазоне 0…15.

mas[0].key = 5;

mas[1].key = 15;

mas[2].key = 1;

mas[3].key = 10;

mas[4].key = 8;

mas[5].key = 3;

mas[6].key = 11;

Допустим, что надо найти элемент с ключом 3. Для этого метод линейного поиска сделает 6 шагов, а для использования двоичного поиска потребуется предварительная сортировка. Количество шагов зависит от способа сортировки, но затраты в этом случае будут выше, чем при линейном поиске.

Для ускорения поиска создадим новый массив (хеш-таблицу), в котором номер элемента будет равен значению ключа: H[ Mas[i].key ] = Mas[i];

Все неиспользуемые элементы массива H имеют значение –1:

H[0].key = -1; H[8].key = 8;

H[1].key = 1; H[9].key = -1;

H[2].key = -1; H[10].key = 10;

H[3].key = 3; H[11].key = 11;

H[4].key = -1; H[12].key =-1;

H[5].key = 5; H[13].key = -1;

H[6].key = -1; H[14].key = -1;

H[7].key = -1; H[15].key = 15;

При такой организации для нахождения любого элемента достаточно сделать только один шаг. Для удаления элемента достаточно поставить значение –1 в соответствующее поле.

Для реальных задач такой подход неприемлем, т. к. размер массива должен быть достаточен для размещения элемента с максимальным ключом, что существенно увеличивает размер хеш-таблицы. Например, для хранения телефонной базы с семизначными номерами необходим массив из 9 999 999 элементов. Для уменьшения размера хеш-таблицы используются различные схемы хеширования.

Схемы хэширования.

Для уменьшения количества элементов в хеш-таблице используют различные алгоритмы сжатия ключей. Однако в этом случае высока вероятность того, что несколько различных элементов получат одинаковый номер в хештаблице. Для решения данной проблемы схема хеширования должна иметь алгоритм разрешения конфликтов, который определяет поведение программы в случае, если новый ключ попадает на уже занятую позицию. Методов разрешения конфликтов достаточно много, однако все они имеют одинаковую структуру:

1. С использованием значения ключа вычисляется номер позиции в хеш-таблице.

2. Если полученная позиция уже занята, то алгоритм разрешения конфликтов находит новую позицию.

3. Если новая позиция тоже занята, повторяется п. 2 до тех пор, пока не будет найдена свободная позиция.

Алгоритм размещения, использующий заданный метод разрешения конфликтов, размещает элементы в хеш-таблице.

Алгоритм поиска находит по значению ключа позицию искомого элемента в хеш-таблице. Если полное значение ключа элемента не совпадает с искомым ключом, то осуществляется дальнейший поиск, начиная с найденной позиции, в соответствии с алгоритмом разрешения конфликтов.

Хэш-таблица с линейной адресацией

Используется функция хеширования i = Key % M. Алгоритм разрешения конфликтов следующий: если найденная позиция i уже занята, то ищется первая незанятая позиция.

Например, имеется следующий массив:

Mas[0].key = 5;

Mas[1].key = 15;

Mas[2].key = 3;

Mas[3].key = 10;

Mas[4].key = 125;

Mas[5].key = 333;

Mas[6].key = 11;

Mas[7].key = 437;

Данные размещаются в хеш-таблице. Функция размещения имеет вид i = key % 10.

Получается следующая хеш-таблица:

H[0] = 10

H[1] = 11

H[2] = -1

H[3] = 3

H[4] = 333

H[5] = 5

H[6] = 15

H[7] = 125

H[8] = 437

H[9] = -1

Достоинство: простой алгоритм вставки и поиска элементов.

Недостатки:

1. Невозможность изменения размера хеш-таблицы.

2. Сложный алгоритм удаления элемента, т. к. удаление элемента часто приводит к необходимости перестройки всей таблицы. Для преодоления данного недостатка можно использовать несколько состояний ячейки: «занята», «не занята», «удалена». Если во время поиска алгоритм попадает на ячейку со статусом «удалена», то поиск продолжается далее. При добавлении данных ячейка со статусом «удалена» считается свободной.

3. Если данные в таблице расположены неравномерно, то скорость поиска может быть очень плохой. Для преодоления данного недостатка можно использовать следующую хеш-функцию: i = (key + M) % 10, где M – простое число, которое может быть сгенерировано датчиком случайных чисел. Для правильной работы датчик должен всегда устанавливаться в одинаковое начальное положение.

1. **Хэш-таблицы с квадратичной и произвольной адресацией, с двойным хэшированием. Хэш-таблица на основе связанным списков.**

Хэш таблицы с квадратично и произвольной адресацией.

Методы содержат такую же функцию хеширования, что и в методе с линейной адресацией, однако поиск свободной ячейки у них отличен от единицы. Алгоритм разрешения конфликтов в методе с квадратичной адресацией следующий: если найденная позиция i уже занята, то ищется первая незанятая позиция по формуле: i=i+p2(p – номер попытки).

В методе с произвольной адресацией незанятая позиция ищется по формуле: i=i+rp(r – заранее сгенерированный массив случайных чисел; p – номер попытки). По сравнению с линейной адресацией данные методы дают более равномерное распределение данных в таблице, однако работают несколько медленнее.

Хэш-таблица с двойным хэшированием.

В отличие от метода с линейной адресацией данный метод использует две хеш-функции:

1. Нахождение позиции элемента в хеш-таблице i = Key % M.

2. Если ячейка с найденным номером i свободна, то перейти к п. 6, иначе – к п. 3.

3. Вычисление значения с = 1 + (Key % (M –2)).

4. Нахождение новой позиции элемента в хеш-таблице i=i-c. Если i<0, то i=i+M.

5. Если ячейка с найденным номером i свободна, то перейти к п. 6, иначе – к п. 4.

6. Вставить элемент в найденную позицию. По сравнению с предыдущими данный метод из-за независимых друг от друга цепочек поиска свободной ячейки дает более равномерное распределение данных в хеш-таблице. Усложнение алгоритма приводит к снижению скорости его работы.

Хэш-таблица на основе связанных списков.

Одним из наиболее эффективных методов разрешения конфликтов состоит в том, что элементы, попадающие на одну и ту же позицию, размещаются в связанных списках. Например, имеется следующий массив:

Mas[0].key = 5;

Mas[1].key = 15;

Mas[2].key = 3;

Mas[3].key = 10;

Mas[4].key = 125;

Mas[5].key = 333;

Mas[6].key = 11;

Mas[7].key = 437;

Данные размещаются в хеш-таблице. Функция размещения имеет вид i = key % 10.

Получается следующая хеш-таблица:

H[0] ← 10

H[1] ← 11

H[2] ← NULL

H[3] ← 3 ← 333

H[4] ← NULL

H[5] ← 5 ← 15 ← 125

H[6] ← NULL

H[7] ← 437

H[8] ← NULL

H[9] ← NULL

Достоинства:

1. Достаточно простой алгоритм вставки и поиска элементов.

2. Связанная таблица не может быть переполнена.

Недостаток: плохая работа с неравномерно размещенными данными.