## Оглавление

[Оглавление 2](#_Toc517043695)

[**1.Использование функций. Создание и использование простой функции. Прототипы функций. Вызов по значению и вызов по ссылке.** 10](#_Toc517043696)

[**2.Использование функций. Использование указателей для связи между функциями. Рекурсия. Равноправность функций в языке Си.** 11](#_Toc517043697)

[**3.Использование функций. Параметры и аргументы функций. Формальные и фактические параметры. Аргумент типа void. Символьные параметры. Целочисленные параметры. Параметры в формате чисел с плавающей точкой. Параметры в формате чисел двойной длины. Массивы в качестве параметров. Аргументы по умолчанию.** 15](#_Toc517043698)

[**4.Использование функций. Возвращение значения функцией: оператор return. Типы функций. Функции типа void. Функции типа char. Функции типа int. Функции типа long. Функции типа float. Функции типа double.** 21](#_Toc517043701)

[**5. Использование функций. Важные возможности C++. Встраивание (inline). Перегрузка (overloading).** 26](#_Toc517043702)

[**6. Использование функций. Функции с переменным числом параметров. Многоточие (...). Задание числа дополнительных параметров с помощью первого параметра. Определение конца списка параметров с помощью параметра индикатора. Использование специального набора макроопределений. Список указателей переменной длины на char (конкатинация строк). Изменение параметров по числу и по типу. Дополнительные примеры функций с произвольным числом параметров.** 28](#_Toc517043703)

[**7. Использование функций. Аргументы функции main(). Строки. Целые числа. Числа с плавающей точкой.** 30](#_Toc517043704)

[**8. Использование функций. Области видимости. Локальные и глобальные переменные. Сложности в правилах области действия (scope rules). Неопределенные символы в программе на С. Использование переменной с файловой областью действия. Приоритет переменных с файловой и локальной областями действия. Проблемы области действия в C++. Операция уточнения области действия в C++.** 31](#_Toc517043705)

[**9. Использование функций. Математические функции** 34](#_Toc517043706)

[**10. Использование функций. Указатель на функцию. Указатели на функции. Указатели на методы. Указатель на функцию и динамическое связывание. Таблицы функций, вызов по имени. Указатель на функцию как средство параметризации алгоритма.** 35](#_Toc517043707)

[**11. Использование функций. Массив указателей на функции. Шаблоны функций в С++. Основные понятия. Параметры шаблонов функций.** 41](#_Toc517043708)

[**12. Использование функций. Шаблоны функций. Аргументы по умолчанию. Функции округления. Компиляция программ, состоящих из двух или более функций.** 46](#_Toc517043709)

[**13? Классы памяти. "Зоопарк" классов памяти. Автоматические переменные. Регистровые переменные. Статические переменные. Внешние переменные. Внешние статические переменные.** 47](#_Toc517043710)

[**14? Классы памяти. Объявление переменных на внутреннем уровне. Объявление переменных на внешнем уровне.** 49](#_Toc517043711)

[**15? Классы памяти. Переменные класса volatile. Ключевое слово mutable. Классы памяти и область действия. Правила области действия переменной. Операция уточнения области действия в C++. Выбор класса памяти.** 52](#_Toc517043712)

[**16? Классы памяти. Пространства имен. Функции и классы памяти. Объявления функций на внешнем уровне.** 54](#_Toc517043713)

[**17. Дополнительные приемы программирования. Совместимость типов. Определение совместимости типов в ANSI С.** 56](#_Toc517043714)

[**18. Дополнительные приемы программирования. Идентичный тип. Перечисляемые типы. Типы массивов. Типы функций. Типы структур и объединений. Типы указателей. Совместимость нескольких исходных файлов** 58](#_Toc517043715)

[**19. Дополнительные приемы программирования. Макроопределения. Определение макросов. Макросы и параметры. Сложности при раскрытии макросов. Создание и использование собственных макросов. Макросы, поставляемые вместе с компилятором. Выбор макроопределения или функции.** 64](#_Toc517043720)

[**20. Дополнительные приемы программирования. Директивы препроцессора. Директива #define. Директива #include. Заголовочные файлы. Правильное использование заголовочных файлов. Более эффективное использование заголовочных файлов. Новый стиль заголовков. Предварительная компиляция заголовочных файлов. Файлы limits.h и float.h.** 65](#_Toc517043721)

[**21. Дополнительные приемы программирования. Директивы #ifdef и #endif. Директива #undef. Директива #ifndef. Директива #if. Директива #else. Директива #elif.** 71](#_Toc517043722)

[**22. Дополнительные приемы программирования. Директива #line. Директива #error. Директива #pragma.** 73](#_Toc517043723)

[**23. Дополнительные приемы программирования. Оператор defined. Условная компиляция** 74](#_Toc517043724)

[**24. Дополнительные приемы программирования. Дополнительные операции препроцессора. Операция подстановки строки (#). Операция конкатенации (##). Операция подстановки символа (#@).** 75](#_Toc517043725)

[**25.Дополнительные приемы программирования. Обработка ошибок: perror(). Модели памяти. Модель tiny. Модель small. Модель medium. Модель compact. Модель large. Модель huge.** 76](#_Toc517043726)

[**26. Дополнительные приемы программирования. Модификаторы функций. Модификаторы cdecl и pascal.** 80](#_Toc517043727)

[**27. Файлы в C. Файлы и потоки. Связь с файлами. Закрытие потоков. Обработка ошибок в C и C++. Переменная errno и коды ошибок.** 81](#_Toc517043728)

[**28.Файлы в C. Переключение и работа с файлами. Переключение вывода. Переключение ввода. Комбинированное переключение. Конвейерная пересылка. Текстовые и бинарные (двоичные) файлы.** 85](#_Toc517043729)

[**29. Файлы в C. Потоковый ввод-вывод. Соединение и отсоединение потока от файла. Функция fopen. Функция fclose. Функция freopen.** 88](#_Toc517043730)

[**30. Файлы в C. Работа с индикаторами ошибки, позиции и конца файла (ferror, clearerr, feof, rewind, fseek, fsetpos, ftell, fgetpos). Блочный ввод-вывод (fwrite, fread).** 89](#_Toc517043731)

[**31. Файлы в C. Символьный ввод-вывод. Функции fputc, putc, fgetc, getc, ungetc. Функции fputs и fgets. Функции fprintf, fscanf.** 92](#_Toc517043732)

[**32. Файлы в C. Работа с буферами (setvbuf, setbuf, fflush). Стандартные потоки (putchar, getchar, puts, gets, perror). Служебные функции для работы с файлами (remove, rename, tmpfile, tmpnam).** 94](#_Toc517043733)

[**33.Файлы в C. Низкоуровневый ввод и вывод в С. Форматированный вывод. Использование функций printf() и fprintf().** 98](#_Toc517043734)

[**34. Файлы в C++. Структура стандартной библиотеки ввода-вывода. Предопределенные потоки. От файла STREAM.H к файлу IOSTREAM.H. Список классов iostream.** 100](#_Toc517043735)

[**35. Файлы в C++. Простота ввода/вывода в С ++. Потоки cin, cout и сеrr. Операции выделения ( >> ) и вставки ( << ). Опции ввода/вывода в С/С++.** 101](#_Toc517043736)

[**36. Файлы в C++. Операции и методы классов. Вывод символов в C++. Преобразование системы счисления в C++. Форматирование строк в C++. Форматирование чисел в C++. Файловый ввод и вывод в C++.** 102](#_Toc517043737)

[**37. Файлы в C++. Классы ios\_base и ios. Потоки вывода. Функции open, close и is\_open. Функция flush. Функция put. Функция seekp и tellp. Функция write. Функция str. Классы потокового вывода.** 106](#_Toc517043738)

[**38. Файлы в C++. Потоки ввода. Функция gcount. Функция get. Функция getline. Функция ignore. Функция peek. Функция putback. Функция read. Функция seekg. Функция sync. Функция unget. Функция str. Классы потокового ввода.** 110](#_Toc517043739)

[**39. Файлы в C++. Потоки ввода-вывода. Классы буферизированных потоков. Класс строковых потоков. Форматирование потока. Резидентные в памяти потоки. Буферы и синхронизация.** 112](#_Toc517043751)

[**40. Файлы в C++. Условные признаки файлов в C++. Опрос и установка состояния потока. Ошибки потоков. Часто применяемые функции. Двоичные файлы. Объединение программ на С и С++. Использование спецификатора extern "С".** 114](#_Toc517043758)

[**41. Файлы в C++. Манипуляторы. Манипуляторы без параметров. Манипуляторы с одним параметром. Манипуляторы с несколькими параметрами.** 118](#_Toc517043759)

[42. Функции для работы с файлами. Полезные функции для работы с файлами. clearerr(). fclose(). fcloseall(). fdopen(). feof(). ferror(). fflush(). fgetc(). fgetchar(). fgetpos(). 120](#_Toc517043760)

[43. Функции для работы с файлами. fgets(). filelength(). fileno(). flushall() fflush(). fopen(). fprintf(). fputc(). fputchar(). fputs(). fread(). 121](#_Toc517043761)

[44. Функции для работы с файлами. freopen(). fscanf(). fseek(). fsetpos(). fstat(). ftell(). fwrite(). getc(). getchar(). gets(). getw(). 122](#_Toc517043762)

[45. Функции для работы с файлами. perror(). printf(). putc(). putchar(). puts(). putw(). remove(). rename(). rewind(). scanf(). setbuf(). 123](#_Toc517043763)

[46. Функции для работы с файлами. setvbuf(). sprintf(). sscanf(). tmpfile(). tmpnam(). ungetc(). vfprintf(). vfscanf(). vprintf(). vsprintf(). vsscanf(). 123](#_Toc517043764)

[47. Функции для работы с файлами. Низкоуровневый ввод и вывод. close(). lseek(). open(). read(). unlink() (\_unlink()). write(). 123](#_Toc517043765)

[48. Структуры С и C++: синтаксис и правила. Структуры C++: расширения синтаксиса и правил. Доступ к элементам структуры. Инициализация структуры. Передача структур в функции. 129](#_Toc517043766)

[49. Массив структур. Описание массива структур. Определение элементов массива структур. Вложенные структуры. 131](#_Toc517043767)

[50. Использование указателей на структуры. Описание и инициализация указателя на структуру. Доступ к элементу структуры при помощи указателя. 132](#_Toc517043768)

[53. Структуры и битовые поля. Структуры. Их дальнейшее использование. Объединения. Объединения: синтаксис и правила. Создание простого объединения. 133](#_Toc517043769)

[54. Вспомогательные средства. Использование typedef. Использование enum. Сложные формы данных. Функции работы с датой и временем. 138](#_Toc517043770)

[55.Динамические структуры данных C++. Линейные однонаправленные списки. Общие сведения. Однонаправленные списки без заглавного звена. Построение списка с заглавным звеном. Удаление списка из памяти. 147](#_Toc517043772)

[56.Операции над списками с заглавным звеном. Поиск звена. Включение звена после звена (1-й случай). Включение звена с заданным информационным полем перед звеном (2-й случай). Удаление звена после звена (1-й случай). Удаление звена на которое указывает ссылка (2-й случай). 155](#_Toc517043778)

[57.Динамические структуры данных C++. Ортогональные списки. Реализация операций над ортогональными списками. 166](#_Toc517043784)

[58.Динамические структуры данных C++. Кольцевые списки. Построение и вывод кольца. Основные операции. 172](#_Toc517043787)

[59.Динамические структуры данных C++. Списки магазинного типа. Списки магазинного типа. Очереди. Формирование очереди. Добавление звена к очереди. Удаление звена из очереди. 177](#_Toc517043790)

[60.Динамические структуры данных C++. Стек. Формирование стека. Включение звена в стек. Удаление звена из стека. 184](#_Toc517043796)

[61.Динамические структуры данных C++. Дек. 190](#_Toc517043800)

[**63. Динамические структуры данных C++. Вставка звена в двунаправленный список (1-й случай). Вставка звена в двунаправленный список (2-й случай). Удаление звена из двунаправленного списка. Указатель на удаляемое звено (1-й случай). Удаление звена из двунаправленного списка. После звена (2-й случай).** 192](#_Toc517043801)

[**64. Динамические структуры данных C++. Двунаправленные кольцевые списки.** 204](#_Toc517043802)

[***65. Динамические структуры данных C++. Деки на базе двунаправленных списков. Формирование дека и его просмотр. Добавление звена в начало дека. Добавление звена в конец дека. Удаление звена из дека слева. Удаление звена из дека справа.*** 210](#_Toc517043803)

[***66. Динамические структуры данных C++. Дерево. Бинарные деревья. Основная терминология. Бинарные деревья поиска. Построение бинарного дерева поиска (рекурсивный алгоритм). Анализ алгоpитма поиска с включениями. Дерево отрезков*** 218](#_Toc517043804)

[***67. Динамические структуры данных C++. Дерево. Обход бинарного дерева. Левосторонний обход бинарного дерева поиска. Концевой обход бинарного дерева поиска. Обратный обход бинарного дерева поиска.*** 220](#_Toc517043805)

[**72 Структуры данных: двоичная куча (binary heap). Введение. Реализация двоичной кучи (binary heap). Реализация класса кучи. Конструктор кучи.** 221](#_Toc517043806)

[**73 Структуры данных: двоичная куча (binary heap). Добавление элемента кучи. Вывод элементов кучи. Упорядочение кучи. Удаление вершины кучи (максимального элемента).** 223](#_Toc517043807)

[**74. Рекурсия и рекурсивные алгоритмы. Сущность рекурсии. Виды рекурсии. Прямая рекурсия. Косвенная рекурсия. Линейная рекурсия. Ветвящаяся рекурсия. Бесконечная рекурсия. Сложная рекурсия. Опережающее описание.** 226](#_Toc517043808)

[**75. Рекурсия и рекурсивные алгоритмы. Имитация работы цикла с помощью рекурсии. Пример 1. Инкремент счетчика итераций. Пример 2. Декремент счетчика итераций. Пример 3. Двойной цикл. Пример 4: Перевод числа в двоичную систему.** 228](#_Toc517043809)

[**76. Рекурсия и рекурсивные алгоритмы. Рекуррентные соотношения. Рекурсия и итерация. Произвольное количество вложенных циклов. Примеры рекурсивных алгоритмов.** 229](#_Toc517043810)

[**77. Рекурсия и рекурсивные алгоритмы. Избавление от рекурсии. Явное использование стека. Запоминание последовательности рекурсивных вызовов. Определение узла дерева по его номеру.** 231](#_Toc517043811)

[**78.Хеш-таблицы. Ключевые термины темы. Простое представление хеш-таблиц. Практическое применение хеш-таблиц.** 232](#_Toc517043812)

[***79*. Хеш-таблицы. Алгоритмы хеширования данных. Таблица прямого доступа. Метод остатков от деления. Метод функции середины квадрата. Метод свертки. Открытое хеширование. Закрытое хеширование (открытая индексация).** 234](#_Toc517043813)

[**80. Хеш-таблицы. Основные, наиболее часто используемые функции хэширования.** 236](#_Toc517043814)

[81. Хеш-таблицы. Коллизи (они же столкновения). Пример коллизий. Методы разрешения коллизий. Борьба с коллизиями. Метод цепочек. Открытая индексация (или закрытое хеширование). Переполнение таблицы и рехеширование. Оценка качества хеш-функции. 237](#_Toc517043815)

[82 Хеш-таблицы. Организация данных для ускорения поиска по вторичным ключам. Инвертированные индексы. Битовые карты. 238](#_Toc517043818)

[**83. Сортировка данных. Основные понятия и определения. Оценка эффективности алгоритмов. Нотация Big-O. Формулировка задачи. Оценка алгоритма сортировки. Нотация "большое О". Список алгоритмов сортировки.** 238](#_Toc517043819)

[**84.** **Сортировка данных. Сортировка пузырьком (Bubble sort). Описание алгоритма. Реализация. Анализ алгоритма.** 239](#_Toc517043820)

[**85.** **Сортировка данных. Сортировка вставками (Insertion sort). Описание алгоритма. Реализация. Анализ алгоритма.** 239](#_Toc517043821)

[**86.** **Сортировка данных. Сортировка выбором (Selection sort). Описание алгоритма. Реализация. Анализ алгоритма.** 240](#_Toc517043822)

[**87. Сортировка данных. Сортировка слиянием (Merge sort) [Рекурсивные алгоритмы]. Описание алгоритма. Реализация. Анализ алгоритма.** 241](#_Toc517043823)

[**88.Сортировка данных. Быстрая сортировка (Quicksort) [Рекурсивные алгоритмы]. Описание алгоритма. Реализация. Анализ** 242](#_Toc517043824)

[**89. Сортировка данных. Сортировка подсчётом. Простой алгоритм. Алгоритм со списком. Алгоритм с двусвязным списком. Сортировка вставками.** 244](#_Toc517043825)

[**90 Сортировка данных. Устойчивый алгоритм. Обобщение на произвольный целочисленный диапазон. Анализ. Квадратичный алгоритм сортировки подсчётом. Анализ.** 247](#_Toc517043827)

[**91. Сортировка данных. Пирамидальная сортировка. Суть метода. Просеивание. Структура пирамиды. Построение пирамиды.** 248](#_Toc517043828)

[**92.Сортировка данных. Поразрядная сортировка.** 251](#_Toc517043829)

# **1.Использование функций. Создание и использование простой функции. Прототипы функций. Вызов по значению и вызов по ссылке.**

Функцию можно рассматривать как операцию, определенную пользователем. В общем случае она задается своим именем. Операнды функции, или формальные параметры, задаются в списке параметров, через запятую. Такой список заключается в круглые скобки. Результатом функции может быть значение, которое называют возвращаемым. Об отсутствии возвращаемого значения сообщают ключевым словом void. Действия, которые производит функция, составляют ее тело; оно заключено в фигурные скобки. Тип возвращаемого значения, ее имя, список параметров и тело составляют определение функции. Ниже представлен пример кода где используется простая функция которая выводит сумму значений a и b.

void sum(int a,int b)

{

cout << a + b << endl ;

}

int main()

{

setlocale(LC\_CTYPE, "rus");

int a;

int b;

cout << "Введите числа a и b" << endl;

cin >> a;

cin >> b;

sum(a,b);

return 0;

}

**Прототип функции** описывает ее интерфейс и состоит из типа возвращаемого функцией значения, имени и списка параметров.

**Тип возвращаемого функцией значения** бывает встроенным, как int или double, составным, как int& или double\*, или определенным пользователем – перечислением или классом. Можно также использовать специальное ключевое слово void, которое говорит о том, что функция не возвращает никакого значения.

**Список параметров** не может быть опущен. Функция, которая не требует параметров, должна иметь пустой список либо список, состоящий из одного ключевого слова void. Например, следующие объявления эквивалентны:

int fork();

int fork( void );

Такой список состоит из названий типов, разделенных запятыми. После имени типа может находиться имя параметра, хотя это и необязательно. В списке параметров не разрешается использовать сокращенную запись, соотнося одно имя типа с несколькими параметрами:

int manip( int vl, v2 ); // ошибка

int manip( int vl, int v2 ); // правильно

Имена параметров не могут повторяться. Имена, фигурирующие в определении функции, нужно использовать в ее теле.

Имена параметров в объявлении и в определении одной и той же функции не обязаны совпадать.

**Передача аргументов.**

Передача данных может происходить по ссылке или по значению.

Когда в функцию передаётся объект по ссылке, то передаётся фактически этот объект. Когда же мы передаём объект по значению, то в функцию передаётся его копия.  
Допустим, есть такой код:

void func\_1(int a) // передача по значению

{}

void func\_2(int & a) // передача по ссылке

{}

# **2.Использование функций. Использование указателей для связи между функциями. Рекурсия. Равноправность функций в языке Си.**

Использование указателей для связи между функциями

I Variant

Указатели служат средством, обеспечивающим правильную работу функции, которая осуществляет обмен значениями переменных.

II Variant

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

|  |  |
| --- | --- |
| 1  2 | // объявление указателя на функцию  /\*тип данных\*/ (\* /\*имя указателя\*/)(/\*список аргументов функции\*/); |

Тип данных определяем такой, который будет возвращать функция, на которую будет ссылаться указатель. Символ указателя и его имя берутся в круглые скобочки, чтобы показать, что это указатель, а не функция, возвращающая указатель на определённый тип данных.  После имени указателя идут круглые скобки, в этих скобках перечисляются все аргументы через запятую как в объявлении прототипа функции. Аргументы наследуются от той функции, на которую будет ссылаться указатель.

**Рекурсия**

Функция, которая прямо или косвенно вызывает сама себя, называется рекурсивной.

Приведем пример. Факториалом числа n является произведение натуральных чисел от 1 до n. Так, факториал 5 равен 120: 1 ? 2 ? 3 ? 4 ? 5 = 120.  
Вычислять факториал удобно с помощью рекурсивной функции:

unsigned long

factorial( int val ) {

if ( val > 1 )

return val \* factorial( val-1 );

return 1;

}

**Равноправность функций в языке Си**

   Все функции в программе, написанной на языке Си, равноправны: каждая из них может вызывать любую другую функцию и в свою очередь каждая может быть вызвана любой другой функцией.

У функции **main( )** есть специфика, она заключается в том, что после "сборки" программы, состоящей из нескольких функций, ее выполнение начинается с первого оператора функции **main( )**. Но этим ее исключительность и ограничивается.

# **3.Использование функций. Параметры и аргументы функций. Формальные и фактические параметры. Аргумент типа void. Символьные параметры. Целочисленные параметры. Параметры в формате чисел с плавающей точкой. Параметры в формате чисел двойной длины. Массивы в качестве параметров. Аргументы по умолчанию.**

**Параметры и аргументы функций**

Параметр функции – это переменная, которая используется в функции и значение которой предоставляет caller (вызывающий объект). Параметры указываются при объявлении функции в круглых скобках. Если их много, то они перечисляются через запятую.

Например:

|  |
| --- |
| // Эта функция не имеет параметров  void doPrint()  {     std::cout << "In doPrint()" << std::endl;  }    // Эта функция имеет один параметр типа int - a  void printValue(int a)  {     std::cout << a << std::endl;  }    // Эта функция имеет два параметра типа int: a и b  int add(int a, int b)  {     return a + b;  } |

Параметры каждой функции действительны только внутри этой функции. Поэтому, если printValue() и add() имеют параметр с именем a, то это не означает, что произойдет конфликт имен. Эти параметры считаются отдельными и никак не взаимодействуют друг с другом.

**Аргумент** – это значение, которое передается из caller-а в функцию. Указывается в скобках при *вызове*функции в программе:

|  |  |
| --- | --- |
|  | printValue(7); // 7 – аргумент функции printValue()  add(4, 5); // 4 и 5 – аргументы функции add() |

Аргументы также перечисляются через запятую. Количество аргументов должно совпадать с количеством параметров, иначе компилятор выдаст сообщение об ошибке.

# **Формальные и фактические параметры функции.**

Формальные параметры – это параметры, которые принимают значение, переменные функции от функционального вызова. Формальные параметры объявляются в списке параметров в начале описания функции. Список параметров определяет имена параметров и порядок, в котором они принимают значение при вызове функции. Формальные параметры – входные и выходные. Входные – те, которые должны быть известны для вычисления в функции. Выходные – результат.

Список параметров состоит из нуля или более идентификаторов, разделенных запятой.

Аргументы, задаваемые при вызове функции, называются фактическими параметрами.

Примеры объявления функций maximum:

Intmaximum(intx;inty;intz– фактические параметр) { формальные параметры }

Аргумент типа void

 Отсутствие списка аргументов функции должно быть указано явно при помощи ключевого слова void.

Символьные параметры

Символьный литерал состоит из символьной константы. Она представлена символом, заключенным в одинарные кавычки. Существует четыре типа символьных литералов:

* Узкие символьные литералы типа char, например 'a'.
* Расширенные символьные литералы типа wchar\_t, например L'a'.
* Расширенные символьные литералы типа char16\_t, например u'a'.
* Расширенные символьные литералы типа char32\_t, например U'a'.

Для символьных литералов могут использоваться любые символы, за исключением зарезервированных символов обратной косой черты ("\"), одинарных кавычек (') или новой строки. Символы можно указывать с помощью универсальных имен символов, при условии что тип является достаточно крупным для размещения символа.

Целочисленные параметры

Целочисленные литералы начинаются с цифры и не имеют дробных частей или экспонент. Целочисленные литералы можно задавать в десятеричной, восьмеричной или шестнадцатеричной форме. Они могут обозначать знаковые или беззнаковые, а также длине или короткие типы.

Если нет ни префикса, ни суффикса, компилятор будет предоставлять значению целочисленного литерала тип int (32 бита), если значение помещается в этот размер; в противном случае он будет предоставлять значению тип long long (64 бита).

Параметры в формате чисел с плавающей точкой

Литералы с плавающей запятой задают значения, которые должны иметь дробную часть. Эти значения содержат десятичные точки (**.**) и могут содержать показатели степени.

Литералы с плавающей запятой имеют «мантиссу», которая определяет значение числа, «экспоненту», которая определяет порядок числа, и необязательный суффикс, задающий тип константы. Мантисса задается как последовательность цифр, затем точка, за которой следует необязательная последовательность цифр, представляющая дробную часть числа.

По умолчанию литералы с плавающей запятой имеют тип **double**. С помощью суффиксов **f** или **l** литерал можно определить как **float** или long double соответственно.

**Параметры в формате чисел двойной длины**

Тип чисел двойной длины double обеспечивает очень большую точность чисел с плавающей точкой. Все функции, описанные в заголовочном файле math.h, получают и возвращают значения типа double. В своих программах вы можете использовать и другие математические функции

**Массивы в качестве параметров**

Передача в функцию одномерного массива

Массив в подпрограмму всегда передаётся по адресу, поэтому достаточно при вызове функции указать адрес начала массива или адрес того элемента, начиная с которого предполагается обрабатывать массив.

**Аргументы по умолчанию.**

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

# **4.Использование функций. Возвращение значения функцией: оператор return. Типы функций. Функции типа void. Функции типа char. Функции типа int. Функции типа long. Функции типа float. Функции типа double.**

Оператор return.

Для возвращения результата из функции применяется оператор **return**. Этот оператор имеет две формы:

|  |  |
| --- | --- |
| 1  2 | return;  return выражение; |

Первая форма используется, если в качестве возвращаемого типа функции применяется тип **void**.

Вторая форма оператора return применяется для возвращения результата из функции. Если функция имеет в качестве возвращаемого типа любой тип, отличный от void, то такая функция обязятельно должна возвратить некоторое значение с помощью оператора return. Причем значение, которое возвращается оператором return, должно соответствовать возвращаемому типу функции, либо допускать неявное преобразование в этот тип.

Типы функций

Тип функции определяется типом возвращаемого ею значения, а не типом ее аргументов. Если указание типа отсутствует, то по умолчанию считается, что функция имеет тип int. Если значения функции не принадлежат типу int, то необходимо указать ее тип в двух местах.

1. Описать тип функции в ее определении:

char pun(ch, n) /\* функция возвращает символ \*/

int n;

char ch;

float raft(num) /\* функция возвращает величину типа float \*/

int num;

2. Описать тип функции также в вызывающей программе. Описание функции должно быть приведено наряду с описаниями переменных программы; необходимо только указать скобки (но не аргументы) для идентификации данного объекта как функции.

main()

{

char, rch, pun();

float raft;

Тип возвращаемого функцией значения бывает встроенным, как int или double, составным, как int& или double\*, или определенным пользователем – перечислением или классом. Можно также использовать специальное ключевое слово void, которое говорит о том, что функция не возвращает никакого значения.

Функции типа void

Если ключевое слово void указывает возвращаемый тип функции, оно означает, что данная функция не возвращает никакого значения. Если оно используется для списка параметров функции, оно означает, что функция не принимает никаких параметров. Если оно используется в объявлении указателя, оно означает, что указатель является "универсальным".

Функции типа char

Тип char был исходным символьным типом в C и C++. Он может использоваться для хранения символов из набора символов ASCII, а также из любого набора символов ISO-8859 и UTF-8. Тип unsigned char часто используется для представления данных типа byte, не являющегося встроенным в C++. Тип char не подходит для текста во многих языках. В современных программах для текста обычно используется один из типов с расширенными символами.

Функции типа int

Функции типа int возвращает целочисленные данные типа int.

Функции типа long

Следующий пример представляет собой программу на C++, получающую в качестве параметра целочисленную переменную и возвращающую значение типа long.

Эта функция получает целые числа и возвращает значения,

равные заданной степени числа 2

#include <iostream>

#include <conio.h>

#include <stdio.h>

using namespace std;

long lpower(int ivalue);

main()

{

int k;

long lanswer;

for(k=0;k<21;k++)

{

lanswer=lpower(k);

// 2 в степени ... равно ...

cout << "2 raised to the " << k

<< " power is " << lanswer << endl;

}

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

long lpower(int ivalue)

{

int t;

long lseed=1;

for(t=0;t<ivalue;t++)

lseed\*=2;

return(lseed);

}

Эта функция умножает исходное число само на себя столько раз, сколько нужно для возведения в заданную степень.

Функции типа float

В следующем примере массив типа float передается в качестве параметра в функцию, которая возвращает значение типа float. В этом примере на C++ определяется произведение всех элементов массива.

Программа на C++ иллюстрирует использование функции типа float. Функция получает массив чисел типа float и возвращает их произведение в виде числа с плавающей точкой

#include <iostream>

#include <conio.h>

#include <stdio.h>

using namespace std;

float fproduct(float farray[]);

main()

{

float fmyarray[7]={(float) 4.3,(float) 1.8,(float) 6.12,(float) 3.19,

(float) 0.01,(float) 0.1,(float) 9876.2};

float fmultiplied;

fmultiplied=fproduct(fmyarray);

// Произведение всех введенных чисел равно

cout << "The product of all array enteris is: "

<< fmultiplied << endl;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

float fproduct(float farray[])

{

int i;

float fpartial;

fpartial=farray[0];

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

fpartial\*=farray[i];

return(fpartial);

}

Функции типа double

В следующем примере на С передаются и возвращаются значения типа double. Функция dtrigcosine() преобразует значение угла, выраженное в градусах, в косинус этого угла.

/\*Программа на С иллюстрирует использование функции типа double.

Функция считывает целые числа от 0 до 90 и возвращает значение

косинуса каждого числа\*/

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <math.h>

using namespace std;

const double dPi=3.14159265359;

double dtrigcosine(double dangle);

main()

{

int j;

double dcosine;

for(j=0;j<91;j++)

{

dcosine=dtrigcosine((double) j);

/\* Косинус ... градусов равен \*/

printf("The cosine of %d degrees is %19.18lf \n",j,dcosine);

}

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

double dtrigcosine(double dangle)

{

double dpartial;

dpartial=cos((dPi/180.0)\*dangle);

return(dpartial);

}

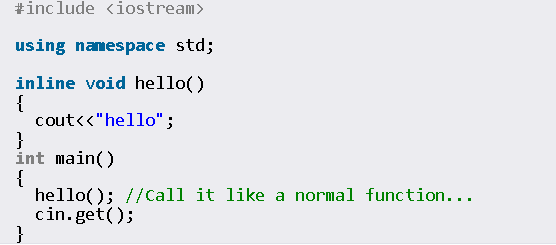
# **5. Использование функций. Важные возможности C++. Встраивание (inline). Перегрузка (overloading).**

Встраивание(inline)

Встроенные функции во многом похожи на заполнитель. После того как вы определите встроенную функцию с помощью ключевого слова inline, всякий раз когда вы будете вызывать эту функцию, компилятор будет заменять вызов функции фактическим кодом из функции.

Вызовы функций занимают больше времени, чем написание всего кода без функции. Используя встроенные функции для замены обычных вызовов функций, вы также значительно увеличите размер вашей программы.

Использовать ключевое слово inline легко, просто поставьте его перед именем функции. Затем, используйте её как обычную функцию.



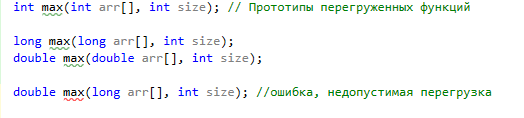
Однако, как только программа будет скомпилирована, вызов hello(); будет заменен на код функции.

Встроенные функции очень хороши для ускорения программы, но если использовать их слишком часто или с большими функциями, будет большая программа. Иногда большие программы менее эффективны, и поэтому они будут работать медленнее, чем раньше. Встроенные функции лучше всего подходят для небольших функций, которые часто вызываются.

Компилятор может игнорировать попытки сделать функцию встроенной.

Перегрузка(overloading)

В С++ возможно определение нескольких функций с одинаковым именем, но с разными типами формальных параметров и результата. При этом транслятор выбирает соответствующую функцию по типу аргументов. Например :

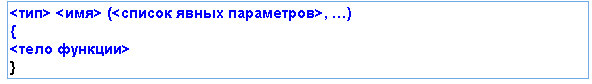


Приведённые функции разделяют одно общее имя, но разные списки параметров. Перегруженные функции можно различать по наличию параметров . Функции не могут быть перегружены, если описание их параметров отличается только модификатором *const* или наличием ссылки. Разные типы возвращаемых значений не могут адекватно различать функции. Поэтому функция double max (long arr[], int size); неотличима от long max(long arr[], int size); функции.

Назначение перегрузки – разрешить выполнять одно и то же действие с разными операндами. Если имеется серия функций, по сути выполняющих одно и то же, то их необходимо перегружать.

# **6. Использование функций. Функции с переменным числом параметров. Многоточие (...). Задание числа дополнительных параметров с помощью первого параметра. Определение конца списка параметров с помощью параметра индикатора. Использование специального набора макроопределений. Список указателей переменной длины на char (конкатинация строк). Изменение параметров по числу и по типу. Дополнительные примеры функций с произвольным числом параметров.**

В С++ можно создавать функции с переменным числом параметров. Параметры, их количество и типы, становятся известны только во время вызова функции. Формат описания функции с переменным числом параметров:



При создании функций с переменным числом параметров необходимо предусмотреть способ определения количества параметров и их типов. Используется для этих целей два способа:

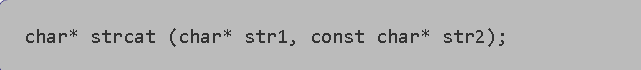
1. Один из параметров определяет число параметров функции;
2. В списке явных параметров задаётся параметр, указывающий на конец списка параметров

Если не один из этих способов не используется, тогда можно использовать **специальный набор макроопределений**.

Параметры функции помещаются в стек, при этом первый параметр оказывается в вершине стека. Переход от одного параметра к другому осуществляется с помощью указателей.

**Конкатенация (объединение) строк. Функция strcat.**

Функция библиотеки <cstring> strcat позволяет объединять две строки в одну.Функция strcat имеет прототип:



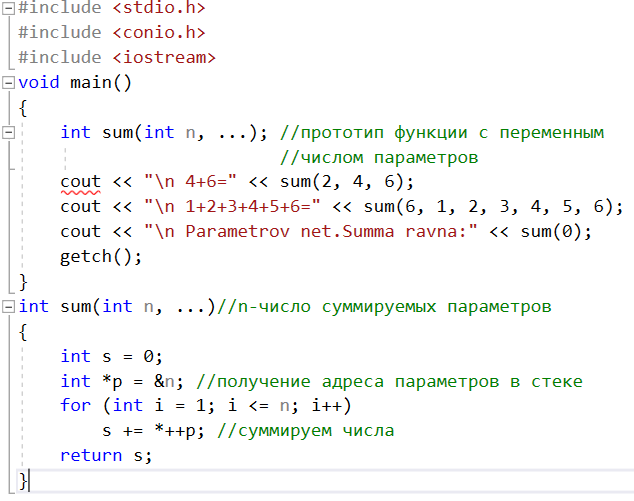
В результате работы функции содержимое строки, на которую указывает str2 присоединяется к содержимому строки, на которую указывает str1. Указатель str1 будет указывать на результирующую строку. Велечина массива str1 должна быть достаточной, чтобы хранить там объединенные строки.

**Функция strncat**

Функция strncat объединяет строки только до указанного в третьем параметре количества символов включительно. Эта функция имеет прототип:

****

**Пример 1.** Задание числа дополнительных параметров с помощью первого параметра. Функция вычисляет сумму значений дополнительных параметров. Список явных параметров состоит из одного параметра, который задаёт число дополнительных параметров.



# **7. Использование функций. Аргументы функции main(). Строки. Целые числа. Числа с плавающей точкой.**

В обоих языках, С и C++, имеется возможность обработки аргументов командной строки, которые представляют собой параметры, вводимые вместе с именем программы при ее вызове из командной строки операционной системы. Эта возможность позволяет передавать аргументы непосредственно вашей программе без дополнительных запросов из этой программы.

**Строки**

Аргументы командной строки передаются как символьные строки, что облегчает работу с ними.

Имена (включая имя запускаемого файла), введенные с командной строки, печатаются на экране в том же порядке и формате.

**Целые числа.**

Во многих программах желательно иметь возможность ввода из командной строки целых чисел; это может быть, к примеру, программа для вычисления средних оценок студентов. В таких случаях символьная информация в кодах **ASCII** должна быть преобразована в целые значения. Поскольку на самом деле число является символьной строкой, оно преобразуется в целое при помощи библиотечной функции **atoi().**

**Числа с плавающей точкой.**

Следующий пример на С позволяет ввести из командной строки значения нескольких углов. Вычисляются косинусы углов и печатаются на экране. Поскольку у значений углов тип **float**, они могут выглядеть по-разному, например: **12.0, 45.78, 0.12345** или **15**. В данном случае строковое представление наших углов преобразуется в число с плавающей точкой при помощи библиотечной функции **atof().** В программе для вычисления косинуса в функции **printf()** используется функция **cos().**

# **8. Использование функций. Области видимости. Локальные и глобальные переменные. Сложности в правилах области действия (scope rules). Неопределенные символы в программе на С. Использование переменной с файловой областью действия. Приоритет переменных с файловой и локальной областями действия. Проблемы области действия в C++. Операция уточнения области действия в C++.**

В С++ поддерживается три их типа:локальная область видимости, область видимости пространства имен и область видимости класса.

**Локальная область** – это часть исходного текста программы, содержащаяся в определении функции (или в блоке). Любая функция имеет собственную такую часть, и каждая составная инструкция (или блок) внутри функции также представляет собой отдельную локальную область.

**Область видимости пространства имен** – часть исходного текста программы, не содержащаяся внутри объявления или определения функции или определения класса. Самая внешняя часть называется глобальной областью видимости или глобальной областью видимости пространства имен.

Объекты, функции, типы и шаблоны могут быть определены в глобальной области видимости. Разрешено задать пользовательские пространства имен, заключенные внутри глобальной области с помощью определения пространства имен. Каждое такое пространство является отдельной областью видимости. Пользовательское пространство, как и глобальное, может содержать объявления и определения объектов, функций, типов и шаблонов, а также вложенные пользовательские пространства имен.

Области видимости и разрешение имен – понятия времени компиляции. Они применимы к отдельным частям текста программы. Компилятор интерпретирует текст программы согласно правилам областей видимости и правилам разрешения имен. Локальная область видимости – это часть исходного текста программы, содержащаяся в определении функции. Все функции имеют свои локальные области видимости. Такие области могут быть вложенными.

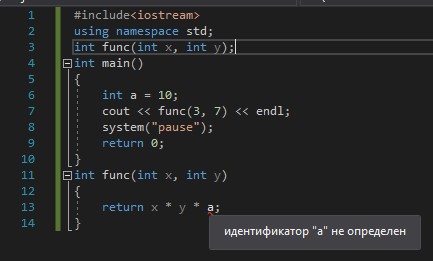
Из-за порядка просмотра областей видимости в процессе разрешения имен объявление из внешней области может быть скрыто объявлением того же имени во вложенной области.

**Сложности в правилах области действия (scope rules).**

Если используются переменные с различной областью действия, то можно столкнуться с побочными эффектами. Например, может существовать переменная как с файловой, так и с локальной областью действия. Правила области действия констатируют, что переменная с локальной областью действия имеет приоритет по сравнению с переменной с файловой областью действия.

**Неопределенные символы в программе на С (Пример на C++).**

Когда функция **func()** пытается обратиться к переменной a, она ее не находит. Потому что область действия этой переменной локальна для функции **main()**.



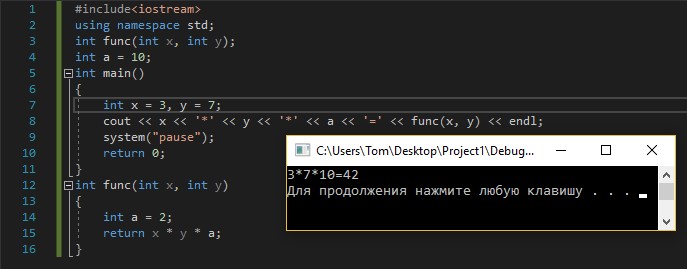
**Использование переменной с файловой областью действия.**

**Переменная с файловой областью действия** - это глобальная переменная, видимая в пределах этого файла.

**Приоритет переменных с файловой и локальной областями действия.**

Правила области действия констатируют, что у переменной, имеющей как локальную, так и файловую область действия, используется ее локальное, а не глобальное значение.

**Проблемы области действия в C++**.



**9. Использование функций. Математические функции**.

В С++ определены в заголовочном файле **<cmath>** функции выполняющие некоторые часто используемые математические задачи.

|  |  |  |  |
| --- | --- | --- | --- |
| **Функция** | **Описание** | **Пример** |  |
| **abs(a)** | модуль или абсолютное значение от **а** | abs(-3.0)=3.0 abs(5.0)= 5.0 |  |
| **sqrt(a)** | корень квадратный из **а,** причём **а** не отрицательно | sqrt(9.0)=3.0 |  |
| **pow(a, b)** | возведение  **а** в степень **b** | pow(2,3)=8 |  |
| **ceil(a)** | округление **а** до наименьшего целого, но не меньше чем **а** | ceil(2.3)=3.0 ceil(-2.3)=-2.0 |  |
| **Функция** | **Описание** | **Пример** | |
| **floor(a)** | округление **а** до наибольшего целого, но не больше чем **а** | floor(12.4)=12 floor(-2.9)=-3 | |
| **fmod(a, b)** | вычисление остатка от a/b | fmod(4.4, 7.5) = 4.4 fmod( 7.5, 4.4) = 3.1 | |
| **exp(a)** | вычисление экспоненты **еа** | exp(0)=1 | |
| **sin(a)** | **a** задаётся в радианах |  | |
| **cos(a)** | **a** задаётся в радианах |  | |
| **log(a)** | натуральный логарифм **a**(основанием является экспонента) | log(1.0)=0.0 | |
| **log10(a)** | десятичный логарифм **а** | Log10(10)=1 | |
| **asin(a)** | арксинус **a**, где  **-1.0 < а < 1.0** | asin(1)=1.5708 | |

# **10. Использование функций. Указатель на функцию. Указатели на функции. Указатели на методы. Указатель на функцию и динамическое связывание. Таблицы функций, вызов по имени. Указатель на функцию как средство параметризации алгоритма.**

Указатели на функции задаются следующим образом:

1. void f() { }
2. void (\*pf)() = &f;
3. pf();

В этом коде **f** - некоторая функция.

Во второй строчке определяется переменная **pf**, которая является указателем на функцию, которая ничего не возвращает и не принимает ни одного аргумента. В определении **pf** ей присваивается адрес функции **f.**

В третьей строке мы вызываем функцию по указатели **pf**(в данном случае будет вызвана функция **f**).

**Динамическое связывание функций.**

Термин динамическое связывание следует понимать как установление соответствия между именем функции и ее адресом. Можно выделить три этапа связывания:

если определение функции и ее вызов находятся в одном модуле, то транслятору ничего не мешает вычислить относительный адрес функций внутри объектного модуля во время трансляции – **статическое связывание;**

при вызове внешней функции, определение которой находится в другом файле, адрес функции становится известен при компоновке программного файла из объектных модулей. Хотя процедура компоновки выполняется после трансляции, данное связывание также называется **статическим**;

набор внешних функций (объектный модуль целиком) может быть оформлен в виде динамически связываемой библиотеки – DLL (dynamic linking library). Это связывание уже считается динамическим.

Если на уровне среды программирования или непосредственно в Си-программах имеет место элемент динамического связывания, то в том или ином виде используется указатель на функцию:

**DLL**, а также все возможные виды динамической загрузки внутреннего программного кода;

**виртуальные функции в Си++** – при конструировании объекта производного класса в базовый класс помещается указатель на таблицу виртуальных функций для производного класса (массив указателей на функции);

**функции, работающие с произвольными типами данных** – для настройки на конкретный тип данных в качестве «**довеска»** получают указатель на функцию, которая «**знает**», как с ним работать. В результате основной алгоритм изолируется от типа данных, с которым работает.

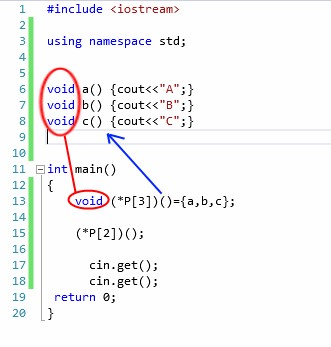
**Таблицы функций, вызов по имени**

Тип данных вида **void (\*pp[])()** – последовательность операций в контексте – массив – указатель – вызов функции. Образно, хотя и не совсем точно этот тип можно назвать **таблицей функций,** вызов которых может производиться по номеру (индексу).

**Указатель на функцию как средство параметризации алгоритма**

если обычный указатель позволяет параметризовать алгоритм обработки данных, то указатель на функцию позволяет параметризовать сам алгоритм. Это значит, что некоторая его часть может быть заранее неизвестна (не определена, произвольна) и будет подключаться к основному алгоритму только в момент его выполнения (динамическое связывание).

# **11. Использование функций. Массив указателей на функции. Шаблоны функций в С++. Основные понятия. Параметры шаблонов функций.**



**P** является массивом из **3 указателей** и при этом каждый из трёх указателей указывает на функцию с пустым списком формальных параметров (пустые скобки), и тип всей этой сущности — **void** обозначает тип каждой ячейки массива, т. е. тип функции, т. е. в ячейках массива можно хранить функции, тип у которых **void**.

**Шаблоны функций—** это инструкции, согласно которым создаются локальные версии шаблонированной функции для определенного набора параметров и типов данных.

**Параметры шаблонов функций.**

1 *#include <iostream>*

2

1. using std::cout;
2. using std::cin;

5

1. template <typename T1, typename T2> *//Шаблон с двумя параметрами*
2. T1 max(T1 a, T2 b){ *//Функция с шаблонными параметрами* 8
3. if (a > b) return a;
4. else if (b > a) return b;
5. else {
6. cout << "a == b";
7. return a;
8. }
9. }

15

1. int main(){
2. int i = 10;
3. double d = 66.999;

19

1. cout << max(i, d); *//Вызов функции*
2. cin.get();
3. }

Параметры шаблонов очень похожи на параметры функций, но в отличие от параметров функций, параметры шаблонов умеют автоматически выводить типы. Иногда имеет смысл помочь автоматике и вручную подсказать правильный тип нужному аргументу. Для этого используются угловые скобки в месте обращения к шаблонной функции, а в угловых скобках пишутся типы, которые будут отдаваться в шаблон. Типы, описываемые в угловых скобках, предназначенные для шаблонных параметров.

# **12. Использование функций. Шаблоны функций. Аргументы по умолчанию. Функции округления. Компиляция программ, состоящих из двух или более функций.**

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

Пример: double heron\_space(double a, const double b = 6.5, const double c =

10.7);

**round, roundf, roundl – округление до ближайшего целого**

**Синтаксис:**

#include < math.h >

double round (double x); float roundf (float x); long double roundl (long double x);

**Аргументы:**

x – число, которое необходимо округлить.

**Возвращаемое значение:** Округленный аргумент.

**Описание:**

Функции округляют аргумент x до ближайшего целого числа. Если округляемый аргумент отстоит от наибольшего и наименьшего целого на одну и ту же величину, то округления произведется до ближайшего большего целого числа. Аргумент и возвращаемое значение функций являются значениями с плавающей точкой.

Причем в функции **roundf** аргумент для расчета и возвращаемое значение задаются числами с плавающей точкой.

В функции **round** аргумент и возвращаемое значение задаются числами с плавающей точкой двойной точности.

В функции **roundl** аргумент и возвращаемое значение задаются числами с плавающей точкой повышенной точности.

**Компиляция программ, состоящих из двух или более функций.**

Простейший способ использования нескольких функций в одной программе заключается в том, чтобы поместить их в один файл, после чего осуществить компиляцию программы, содержащейся в этом файле так, как будто она состояла из одной функции

Второй способ заключается в применении директивы #include Если одна функция содержится в файле с именем filel.c, а вторая в файле file2.c, поместите эту директиву в файл filel.c

# **13? Классы памяти. "Зоопарк" классов памяти. Автоматические переменные. Регистровые переменные. Статические переменные. Внешние переменные. Внешние статические переменные.**

Класс памяти определяет порядок размещения объекта в памяти. В языке Си существует четыре класса памяти:

* Автоматический (automatic);
* Регистровый (register);
* Статический (static);
* Внешний (external).

**1. Автоматический (automatic)**

По умолчанию, локальные переменные имеют класс auto. Такие переменные располагаются на стеке, а их область видимости ограничена своим блоком. Спецификатор auto уведомляет компилятор о том, что локальная переменная, перед именем которой он стоит, создаётся при входе в блок и разрушается при выходе из блока. Пример:

int nVar = 5;

int auto nVar = 5;

Две эти записи абсолютно идентичны.

#### **2. Регистровый (register)**

Когда мы определяем регистровую переменную, то мы просим компилятор, чтобы переменная располагалась в регистре, а не в оперативной памяти. Компилятор может сделать переменную регистровой, если позволяют условия. Регистровые переменные определяются с помощью служебного слово register перед типом.

Так как регистровая переменная не имеет адреса, то к ней не применима операция взятия адреса, это вызовет ошибку во время компиляции.

Применять register можно только к near указателям и целому типу. Использовать register можно и при указании формальных параметров функций. Примеры:

register int nVar;

int func(register int var);

#### **3. Статический (static)**

Спецификатор static указывает компилятору на хранение локальной переменной во время всего жизненного цикла программы вместо ее создания и разрушения при каждом входе в область действия и выходе из неё.

Модификатор static можно также применять к глобальным переменным. В этом случае область видимости такой переменной ограничивается файлом, в котором она объявлены, это означает, что переменная будет иметь внутреннюю привязку.

Пример. Объявим в отдельном модуле две функции: одну статическую, а другую обычную:

static int staticFunc(int nVar) //статическая

{

return ++nVar;

}

int simpleFunc(int nVar) //обычная

{

return ++nVar;

}

Если бы служебное слово static отсутствовало в функции staticFunc, то каждый раз при вызове функции локальная переменная nVar снова создавалась, инициализировалась и уничтожалась после выхода из функции, что делается в простой функции simpleFunc.

#### **4. Внешний (external)**

Внешние переменные вводятся как противоположные автоматическим. Это глобальные переменные и к ним можно обращаться именами из любой функции. Поскольку внешние переменные доступны везде, их можно использовать для связи между функциями, не пренебрегая механизму формальных параметров.

Класс памяти extern в Си используем в двух случаях:

* если переменная объявляется в программе ниже, чем ссылка на неё;
* если переменная объявлена в другом модуле.

Пример:

#include <stdio.h>

int x=145; //Описание внешней переменной

main()

{

extern int x,y;

printf("x=%d y=%d \n",x,y);

}

int y=541; //Описание внешней переменной

Внешние переменные могут определятся вне какой-либо функции, при этом выделяется фактическая память. В любой другой функции, обращающейся к этим переменным, они дол жны описываться с помощью описателя extern.

# **14? Классы памяти. Объявление переменных на внутреннем уровне. Объявление переменных на внешнем уровне.**

#### **1. Объявление переменных на внутреннем уровне**

Любая из четырех спецификаций класса памяти может быть использована для объявления переменной на внутреннем уровне. Если спецификация класса памяти опущена в объявлении переменной на внутреннем уровне, то подразумевается класс памяти **auto**. Как правило, ключевое слово **auto** опускается. Понятия объявления и определения для переменных внутреннего уровня совпадают, если только в объявлении не задана спецификация класса памяти **extern**.

*int i = 1;* *//*определение i

*main() {*

*extern int i;* //объявление i, ссылающееся на данное выше определение

*static int a;* //начальное значение а равно нулю; область действия а — функция main

*register int b = 0;* //b будет (по возможности) помещено в регистр

*int с = 0;* //поумолчанию с будет иметь класс памяти auto

*printf("%d,%d,%d,%dn", i, a, b, c);* //печатаются значения 1, 0, 0, 0

*}*

*other()*

*int i = 16;* //локальное переопределение переменной i

*static int a = 2;* //область действия переменной а — функция other

*a += 2;*

*printf("%d,%dn", i, a);* //печатаются значения 16, 4

*}*

Переменная **i** определяется на внешнем уровне с начальным значением **1**; В функции **main** объявление **i** является ссылкой на определение переменной **i** внешнего уровня.

**2. Объявление переменных на внешнем уровне**

Объявления переменной на внешнем уровне используют спецификации класса памяти **static** и **extern** или вообще опускают их. Спецификации класса памяти **auto** и **register** не допускаются на внешнем уровне.

Определение внешней переменной—это объявление, которое вызывает выделение памяти для этой переменной и инициализирует ее (явно или неявно). Определение на внешнем уровне может задаваться в следующих различных формах:

1) Переменная может быть определена путем ее объявления со спецификацией класса памяти **static**. Таким образом, каждое из объявлений:

*static int k = 16;*

и

*static int k;*

рассматривается как определение.

2) Переменная может быть определена, если спецификация класса памяти в ее объявлении опущена, и переменная явно инициализируется, например,

*int j = 3;*

Область действия переменной, определенной на внешнем уровне, распространяется от точки, где она определена, до конца исходного файла.

Существует одно исключение из правил. Можно опустить в объявлении переменной на внешнем уровне и спецификацию класса памяти, и инициализатор. Например, объявление **int n;** будет вполне корректным внешним объявлением. Это объявление имеет различный смысл в зависимости от контекста:

1) Если в каком-то другом исходном файле программы есть определение на внешнем уровне переменной с таким же именем, то данное объявление является ссылкой на это определение. В этом случае объявление аналогично объявлению со спецификацией класса памяти **extern**.

2) Если же такого определения переменной в программе нет, то данное объявление само считается определением переменной.

# **15? Классы памяти. Переменные класса volatile. Ключевое слово mutable. Классы памяти и область действия. Правила области действия переменной. Операция уточнения области действия в C++. Выбор класса памяти.**

1. **Переменные класса volatile**

квалификатор **volatile** информирует компилятор что переменная может быть изменена не явным способом т.е. без явного использовать оператора присвоения. как правило компиляторы автоматически применяют оптимизацию, предполагая что значение переменной остаётся постоянным, если оно не указано с левой стороны от оператора присваивания.

Пример:

bool exit = true;

while( exit )

{

};

В данном случае значение переменной **exit** будет прочитано только один раз. если содержание переменно **exit** может изменять не только код текущего потока, но допустим другой поток или вообще некое внешние устройство, то мы можем получить бесконечный цикл.

Но если мы перед объявлением переменной поставим квалификатор **volatile**, то переменная будет считываться при каждой проверке.

Пример:

volatile bool exit = true;

while( exit )

{

};

**2. Классы памяти и область действия**

Иногда есть необходимость изменить некий объект внутри класса, гарантируя неприкосновенность остальных элементов. Неприкосновенность можно гарантировать при помощи **const**, однако **const** запрещает изменение всего.

Область действия переменной – это правила, которые устанавливают, какие данные доступны из текущего места программы. Имеются три типа переменных: глобальные, локальные и формальные.

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

Глобальные переменные объявляются вне какой-либо функции. Глобальные переменные могут быть использованы в любом месте программы, но перед их первым использованием они должны быть объявлены и проинициализированы. Область действия глобальных переменных - вся программа с момента их объявления.

В языке **С** каждая переменная принадлежит к одному из четырех классов памяти – автоматическая (**auto**), внешняя (**extern**), статическая (**static**), регистровая (**register**). Тип памяти указывается ключевым словом, стоящим перед спецификацией типа переменной. Например, **register int a**;

Область действия переменной, определенной на внешнем уровне, распространяется от точки, где она определена, до конца исходного файла.

Бинарная операция уточнения области действия позволяет сослаться на данные-элемент или функцию-элемент класса, даже если имеются одноименные переменные или функции, определенные вне класса или в нескольких классах. Она используется также при описании функции-элемента вне класса.

Выбор класса памяти, помимо явных спецификаторов, зависит от размещения определения или объявления в тексте программы. Модуль, функция, блок могут включать соответствующие операторы объявления или определения, причём всякий раз определяемый объект будет размещаться в строго определённых областях памяти.

# **16? Классы памяти. Пространства имен. Функции и классы памяти. Объявления функций на внешнем уровне.**

**1. Пространства имен**

**Пространство имен** — это декларативная область, в рамках которой определяются различные идентификаторы. Пространства имен используются для организации кода в виде логических групп и с целью избежания конфликтов имен, которые могут возникнуть, особенно в таких случаях, когда база кода включает несколько библиотек. Все идентификаторы в пределах пространства имен доступны друг другу без уточнения. Код в файлах заголовков всегда должен содержать полное имя в пространстве имен.

В следующем примере показано объявление пространства имен и продемонстрированы три способа доступа к членам пространства имен из кода за его пределами.

namespace ContosoData

{

class ObjectManager

{

public:

void DoSomething() {}

};

void Func(ObjectManager) {}

}

Использование полного имени:

ContosoData::ObjectManager mgr;

mgr.DoSomething();

ContosoData::Func(mgr);

Чтобы добавить в область видимости один идентификатор, используйте объявление **using**:

using WidgetsUnlimited::ObjectManager;

ObjectManager mgr;

mgr.DoSomething();

Чтобы добавить в область видимости все идентификаторы пространства имен, используйте директиву **using:**

using namespace WidgetsUnlimited;

ObjectManager mgr;

mgr.DoSomething();

Func(mgr);

Функции могут быть объявлены со спецификаторами класса памяти **static** или **extern**. Функции всегда имеют глобальное время жизни.

Правила видимости для функций отличаются от правил видимости для переменных. функции не могут иметь блочной видимости и видимость функций не может быть вложенной. Функция объявленная как **static**, видима только в пределах исходного файла, в котором она определяется. Любая функция в том же самом исходном файле может вызвать функцию **static**, но функции **static** из других файлов нет. Функция **static** с тем же самым именем может быть объявлена в другом исходном файле.

Функции, объявленные как **extern** видимы в пределах всех исходных файлов, которые составляют программу. Любая функция может вызвать функцию **extern**.

Объявления функций, в которых опущен спецификатор класса памяти, считаются по умолчанию **extern**.

Для функций также, как и для переменных, различают объявление и определение. Объявление обычно прячут в заголовочный файл, также как и объявление переменных. Определения находятся в си файле.

# **17. Дополнительные приемы программирования. Совместимость типов. Определение совместимости типов в ANSI С.**

[*Приведение типов в C++*](http://alenacpp.blogspot.com/2005/08/c.html)

есть четыре способа приведения типов

**int** i;

**const** **int** \* pi = &i;

**// \*pi имеет тип const int,**

**// но pi указывает на int, который константным не является**

**int**\* j = **const\_cast**<**int** \*> (pi);

1)**const\_cast**  
Самое простое приведение типов. Убирает так называемые cv спецификаторы (cv qualifiers), то есть const и volatile. volatile встречается не очень часто, так что более известно как приведение типов, предназначенное для убирания const. Если приведение типов не удалось, выдается ошибка на этапе компиляции.

**2) static\_cast** 

Может быть использован для приведения одного типа к другому.   
static\_cast между указателями корректно, только если один из указателей - это указатель на void или если это приведение между объектами классов, где один класс является наследником другого.

**3)dynamic\_cast**  
Безопасное приведение по иерархии наследования, в том числе и для виртуального наследования.

Используется RTTI (*Runtime Type Information*), чтобы привести один указатель на объект класса к другому указателю на объект класса. Классы должны быть полиморфными, то есть в базовом классе должна быть хотя бы одна виртуальная функция. Если эти условие не соблюдено, ошибка возникнет на этапе компиляции. Если приведение невозможно, то об этом станет ясно только на этапе выполнения программы и будет возвращен NULL.

Работа со ссылками происходит почти как с указателями, но в случае ошибки во время исполнения будет выброшено исключение bad\_cast.  
  
**4)reinterpret\_cast**  
Не портируемо, результат может быть некорректным, никаких проверок не делается. Не может быть приведено одно значение к другому значению. Обычно используется, чтобы привести указатель к указателю, указатель к целому, целое к указателю. Умеет также работать со ссылками.

**Примеры**

unsigned\* и int\* никак не связаны между собой. Есть правило приведения между unsigned (int) и int, но не между указателями на них. И привести их с помощью static\_cast не получится, придется использовать reinterpret\_cast. То есть вот так работать не будет:

**unsigned**\* v\_ptr;

**cout** << \***static\_cast**<**int**\*>(v\_ptr) <<endl;

Приведение вниз по иерархии:

**class** Base { **public**: **virtual** ~Base(**void**) { } };

**class** Derived1 : **public** Base { };

**class** Derived2 : **public** Base { };

**class** Unrelated { };

Base\* pD1 = **new** Derived1;

Вот такое приведение корректно: dynamic\_cast<Derived1 \*>(pD1);

А вот такое возвратит NULL: dynamic\_cast<Derived2 \*>(pD1);

Никак не связанные указатели можно приводить с помощью reinterpret\_cast:

Derived1 derived1;

Unrelated\* pUnrelated = **reinterpret\_cast**<Unrelated\*>(&derived1);

**int**\* pi;

**void**\* vp = pi;

**char**\* pch = **static\_cast**<**char**\*>(vp);

Пример использования static\_cast:

Приведение в стиле C можно использовать, чтобы избавиться от значения, возвращаемого функцией. Польза от этого сомнительная, правда...

**string** sHello("Hello");

(**void**)sHello.size();

Также я видела использование приведение типов в стиле С для приведения к приватному базовому классу, но для этого можно использовать и reinterpret\_cast.

# **18. Дополнительные приемы программирования. Идентичный тип. Перечисляемые типы. Типы массивов. Типы функций. Типы структур и объединений. Типы указателей. Совместимость нескольких исходных файлов**

*Перечисляемые типы*

Перечислимый тип вводится ключевым словом **enum** и задает набор значений, определяемый пользователем. Набор значений заключается в фигурные скобки и является набором целых именованных констант, представленных своими идентификаторами. Эти константы называются **перечислимыми константами**.

пример перечислимого типа:

**enum** Months {JAN = 1, FEB, MAR, APR, MAY, YUN, YUL AUG, SEP, OCT, NOV, DEC};

Это объявление создает определенный пользователем тип Months с константами перечисления, представляющими месяцы года. Поскольку первое значение приведенного перечисления установлено равным 1, оставшиеся значения увеличиваются на 1 от 1 до 12.

В объявлении перечислимого типа любой константе перечисления можно присвоить целое значение.

После того, как константа перечисления определена, попытка присвоить ей другое значение является синтаксической ошибкой.

Идентификаторы в **enum** должны быть уникальными, но отдельные константы перечисления могут иметь одинаковые целые значения.

Набор идентификаторов перечислимого типа — собственный уникальный тип, отличающийся от других целочисленных типов.

Перечислимая константа может быть объявлена **анонимно**, то есть без имени типа.

**enum** {FALSE, TRUE};

**enum** {lazy, hazy, crazy} why;

Первое объявление — распространенный способ объявления мнемонических целочисленных констант. Второе объявление объявляет переменную перечислимого типа why, с допустимыми значениями этой переменной lazy, hazy и crazy.

Перечисления могут неявно преобразовываться в обычные целочисленные типы, но не наоборот.

*Типы массивов*

***Массив*** это ***структура данных***, представленная ***в виде группы ячеек одного типа***, объединенных под одним единым именем. Массивы используются для обработки большого количества однотипных данных. ***Имя массива является указателем***. ***Отдельная ячейка*** данных массива называется ***элементом массива***.

***Одномерный массив*** — массив, с одним параметром, характеризующим количество элементов одномерного массива. Фактически одномерный массив — это массив, у которого может быть только одна строка, и n-е количество столбцов.

***Индекс ячейки*** – это целое неотрицательное число, по которому можно обращаться к каждой ячейке массива и выполнять какие-либо действия над ней (ячейкой).

***Двумерные массивы***

двумерный массив — это обычная таблица, со строками и столбцами. Фактически двумерный массив — это одномерный массив одномерных массивов.

В объявлении  двумерного массива, также как и в объявлении одномерного массива, первым делом, нужно указать:

* тип данных;
* имя массива.

инициализация двумерного массива:

int a[5][3] = { {4, 7, 8}, {9, 66, -1}, {5, -5, 0}, {3, -3, 30}, {1, 1, 1} };

*О типах функции:*

 В языке программирования С++ есть два типа функций:

1. Функции, возвращающие значение
2. Функции, которые ничего не возвращают

Для возврата значений в функциях первого типа используется *return.*

***Структуры и объединения***

*Структуры (****struct****)* - это совокупность переменных, объединенных одним именем, предоставляющая общепринятый способ совместного хранения информации.

Пример кода:

*Struct name {*

*int hour;*

*int min;*

*int sec;*

*};*

*Объединения (****union****)* - множество объектов разного типа расположенных в одной области памяти. Другими словами, появляется возможность интерпретировать одни и те же данные по-разному.

Элементы структур и объединений называют ***членами* данных**.

Когда объявлено **объединение**, компилятор автоматически создает переменную достаточного размера для хранения наибольшей переменной, присутствующей в объединении.

***Указатели***

Указатель — это та же переменная, только инициализируется она не значением одного из множества типов данных в C++, а адресом, адресом некоторой переменной, которая была объявлена в коде ранее.

# **Указатели на массивы**

Рассмотрим статичный одномерный массив определенной длинны и инициализируем его элементы:

void main(){

const int size = 7;

int i\_array[size];

for (int i = 0; i != size; i++){

i\_array[i] = i;

}

}

А теперь будем обращаться к элементам массива, используя указатели:

int\* arr\_ptr = i\_array;

for (int i = 0; i != size; i++){

cout << \*(arr\_ptr + i) << endl;

}

Что здесь происходит: мы инициализируем указатель *arr\_ptr* адресом начала массива *i\_array*. Затем, в цикле мы выводим элементы, обращаясь к каждому с помощью начального адреса и смещения. То есть:

\*(arr\_ptr + 0)

это тот же самый нулевой элемент, смещение нулевое (*i* = 0),

\*(arr\_ptr + 1)

— первый (*i* = 1), и так далее.  
  
Однако здесь возникает естественный вопрос — почему присваивая указателю адрес начала массива, мы не используем операцию взятия адреса? Ответ прост — использование идентификатора массива без указания квадратных скобок эквивалентно указанию адреса его первого элемента. Тот же самый пример, только в указатель «явно» занесем адрес первого элемента массива:

int\* arr\_ptr\_null = &i\_array[0];

for (int i = 0; i != size; i++){

cout << \*(arr\_ptr\_null + i) << endl;

}

Пройдем по элементам с конца массива:

int\* arr\_ptr\_end = &i\_array[size - 1];

for (int i = 0; i != size; i++){

cout << \*(arr\_ptr\_end - i) << endl;

}

# **19. Дополнительные приемы программирования. Макроопределения. Определение макросов. Макросы и параметры. Сложности при раскрытии макросов. Создание и использование собственных макросов. Макросы, поставляемые вместе с компилятором. Выбор макроопределения или функции.**

В программировании принято называть идентификатор, следующий после  
#define **макроопределением**. В случае, если этот идентификатор встречается в программе, он называется **макровызовом**, а замена препроцессором макроопределения на строку замещения называется **макрорасширением**.

Если непосредственно за идентификатором следует открывающая скобка,то препроцессор считает, что в программе определено макроопределние с аргументамми.  
Общая форма записи директивы #define:

**#define идентификатор\_макроса(аргументы) замещающий\_текст**

Между идентификатором макроса и открывающейся скобкой не должно быть пробела. Вызов макроса осуществляется выражением:

идентификатор\_макроса(аргументы)

Макрос, определяемый директивой препроцессора **#define**, это символическое имя некоторых операций.

Например, следующий макрос с одним параметром определяет площадь круга, воспринимая передаваемый в него параметр как радиус:

#define CIRC(x) (3.14 \* (х) \* (х))

Везде в тексте файла, где появится идентификатор **CIRC(А)**, значение аргумента А будет использовано для замены х в замещающем тексте и этот расширенный текст макроса будет использован для замещения. Например, оператор с макросом: в тексте программы примет вид:

S=CIRC(4);  
S=(3.14 \* (4) \* (4));

Поскольку это выражение состоит только из констант, его значение будет вычислено во время компиляции и полученный результат будет присвоен переменной S во время выполнения программы. Если вызов имеет вид:

S = CIRC (a + b);

то после расширения макроса текст будет иметь вид:

S = (3.14 \* (а + b) \* (а + b));

В данном случае аргумент макроса является выражением, содержащим переменные а и b. Поэтому вычисления будет осуществляться не во время компиляции, а во время выполнения программы.

Вызов функции сопряжен с накладными расходами и затягивает выполнение программы. С другой стороны, макрос расширяется во всех местах текста, где используется его вызов. Так что функции позволяют сокращать объем выполняемого файла, а макросы - сокращать скорость выполнения. Правда, макросы тоже могут быть связаны с дополнительными накладными расходами.

Недостатком макросов является отсутствие встроенного контроля согласования типов аргументов и формальных параметров. Отсутствие соответствующих предупреждений компилятора может приводить к ошибкам программы, которые трудно отлавливать.

# **20. Дополнительные приемы программирования. Директивы препроцессора. Директива #define. Директива #include. Заголовочные файлы. Правильное использование заголовочных файлов. Более эффективное использование заголовочных файлов. Новый стиль заголовков. Предварительная компиляция заголовочных файлов. Файлы limits.h и float.h.**

**Препроцессор** — это специальная программа, являющаяся частью компилятора языка Си. Она предназначена для предварительной обработки текста программы. Препроцессор позволяет включать в текст программы файлы и вводить макроопределения. Работа препроцессора осуществляется с помощью специальных директив (указаний). Они отмечаются знаком решетка #.

Директива #include вставляет код из указанного файла в текущий файл, то есть, просто подключив другой файл, мы можем пользоваться его функциями, классами, переменными.

Подключение заголовочных файлов выполняется во время компиляции, либо как файл, который является частью вашего проекта.

Если подключаемый файл не найден, процесс компиляции завершается с ошибкой.

#### **Директива #define**

Директива #define принимает две формы:

* определение констант;
* определение макросов.

Определение констант

|  |  |
| --- | --- |
|  | #define nameToken value |

При использовании имени константы — nameToken, оно будет заменено значением value. Смотрим пример использования константы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #include <iostream>    #define TEXT "Марс" // определение константы    int main()  {      std::cout <<  TEXT;      return 0;  } |

для доступа к значению константы, просто используем её имя.

**заголовочные файлы** — основной способ подключить к программе типы данных, структуры, прототипы функций, перечислимые типы и макросы, используемые в другом модуле. По умолчанию используется расширение .h. Чтобы избежать повторного включения одного и того же кода, используются директивы #ifndef, #define, #endif.

***limits.h*** — заголовочный файл **стандартной библиотеки общего назначения** языка программирования Си, который включает определения характеристик общих типов переменных. Значения зависят от специфики реализации, но могут быть не ниже диапазона, определенного для конечного значения в удобной реализации Си.

Большинство реализаций будут иметь больший диапазон по крайней мере для некоторых из этих чисел. например:

* Реализации [дополнительного кода](https://ru.wikipedia.org/wiki/Дополнительный_код_(представление_числа)) имеют SCHAR\_MIN равным −128
* [32-битные](https://ru.wikipedia.org/w/index.php?title=32-битность&action=edit&redlink=1) реализации устанавливают INT\_MAX равным +2,147,483,647
* Реализации с поддержкой [Юникода](https://ru.wikipedia.org/wiki/Юникод) устанавливают MB\_LEN\_MAX 4 или более.

**float.h** — заголовочный файл стандартной библиотеки языка программирования С, который содержит макрос, определяющий различные ограничения и параметры типов с плавающей точкой.

# **21. Дополнительные приемы программирования. Директивы #ifdef и #endif. Директива #undef. Директива #ifndef. Директива #if. Директива #else. Директива #elif.**

#### **Директива #ifdef**

|  |  |
| --- | --- |
| 1  2  3  4  5 | #ifdef nameToken  // код, который выполнится, если nameToken определен  #else  // код, который выполнится, если nameToken не определен  #endif |

Директива #ifdef проверяет, был ли ранее определен макрос или символическая константа как #define. Если — да, компилятор включает в программу код, который находится между директивами #ifdef и #else, если nameToken ранее определен не был, то выполняется код между #else и #endif, или, если нет директивы #else, компилятор сразу переходит к #endif.

#### **Директива #undef**

Директива #undef переопределяет константу или препроцессорный макрос, ранее определенный с помощью директивы #define.

|  |  |
| --- | --- |
| 1 | #undef nameToken |

Давайте посмотрим пример использования директивы #undef:

|  |  |
| --- | --- |
| 1  2  3 | #define E 2.71828 // раннее определенный макрос  int sumE = E + E; // обращение к макросу  #undef E // теперь E - не макрос |

Как правило, директива #undef используются для снятия, ранее определенной константы или макроса, в небольшой области программы. Это делается для того, чтобы для всей программы, макроc или константа оставались, а для некоторой области, эти же макрос или константа могут быть переопределены.

#### **Директива #ifndef**

|  |  |
| --- | --- |
| 1  2  3  4  5 | #ifndef nameToken  // код, который выполнится, если nameToken не определен  #else  // код, который выполнится, если nameToken определен  #endif |

Директива #ifndef проверяет, был ли ранее определен макрос или символическая константа как #define. Если — да, компилятор включает в программу код, который находится между директивами  #else и #endif, если nameToken ранее определен не был, то выполняется код между #ifndef и #else, или, если нет директивы #else, компилятор сразу переходит к #endif.  Директива #ifndef может быть использована для подключения заголовочных файлов. если они не подключены, для этого использовать символическую константу, как индикатор подключенного к проекту функционала.

#### **Директива #if**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #if value  // код, который выполнится, в случае, если value - истина  #elsif value1  // этот код выполнится, в случае, если value1 - истина  #else  // код, который выполнится в противном случае  #endif |

Директива #if проверяет, является ли значение value истиной и, если это так, то выполняется код, который стоит до закрывающей директивы #endif. В противном случае, код внутри #if не будет компилироваться, он будет удален компилятором, но это не влияет на исходный код в исходнике.

# **22. Дополнительные приемы программирования. Директива #line. Директива #error. Директива #pragma.**

Директива препроцессора **#error** имеет следующий формат:

#error errmsg

Директива печатает в процессе компиляции сообщение об ошибке вида:

Error: filename line# : Error directive

e errmsg

где **errmsg** – сообщение, заданное директивой **#error**. После печати этого сообщения компиляция прекращается. Директива используется в сочетании с директивами условной компиляции и срабатывает при возникновении условий, не позволяющих продолжить работу. Например:

#ifndef File1#error Не найден файл File1

Директива препроцессора **#line** задаёт целочисленное константное начальное значение номера строки для нумерации следующих за директивой строк исходного текста программы. Возможны две формы директивы:

#line номер\_строки

#line номер\_строки «имя\_файла»

Элемент директивы номер\_строки задаёт начальное значение номера строки. Все последующие строки исходного текста программы будут нумероваться, начиная с этого номера. Директива #line обычно используется для того, чтобы сделать сообщение о синтаксических ошибках и предупреждениях компилятора более удобными для понимания. Номера строк не добавляются в исходный файл.

Директива #pragma имеет следующий синтаксис:

#pragma имя опции

и вызывает действия, зависящие от указанной опции. Пример:

1) #pragma startup имя\_функции <приоритет>

Указывает на функцию, которая должна вызываться в самом начале программы (перед вызовом функции **main**).

2) #pragma hdrstop

Эта опция связана с особенностью работы препроцессора, производительность которого существенно повышается, если учитывается, что некоторое количество заголовочных файлов общие для всех модулей. Директива **#pragma hdrstop** указывает компилятору конец списка таких общих файлов.

3) #pragma message("сообщение")

просто выдает сообщение при компиляции.

# **23. Дополнительные приемы программирования. Оператор defined. Условная компиляция**

Специальный оператор **defined** используется в ‘**#if’** и **‘#elif’** выражения, чтобы проверить, является ли определенное имя определяется как макрос. **defined *name*** и **defined (*name*)** не оба выражения, значение которого равно **1**, Если ***name***определяется как макрос на текущая точка в программе, и **0** В противном случае. Таким образом, **#if defined MACRO** в точности эквивалентно **#ifdef MACRO**.

**defined** полезно, когда вы хотите протестировать более одного макроса для существование сразу.

Для условной  
компиляции используются директивы препроцессора #if, #ifdef, #ifndef, #else,  
#elif, #endif

Директива **#else** и следующие за ней операторы могут отсутствовать.

Если **\_константноевыражение\_**, входящее в состав директивы **#if** истинно (**отлично от 0**),  
компилируются операторы и выполняются директивы препроцессора, заключенные  
между директивами **#if** и **#else**, а операторы и директивы препроцессора между  
**#else** и **#endif** игнорируются. Если директива **#else** отсутствует, то компилируются  
операторы, заключенные между директивами **#if** и **#endif**.

Если \_**константное выражение\_**, входящее в состав директивы **#if** ложно  
(**равно 0**), то компилируются операторы и выполняются директивы препроцессора,  
заключенные между директивами **#else** и **#endif**, а операторы и директивы  
препроцессора между **#if** и **#else** игнорируются. Если директива **#else** отсутствует,  
операторы между директивами **#if** и **#endif** игнорируются.

Вместо директивы **#if** может быть записана директива **#ifdef**  
или директива **#ifndef**. Общая форма записи директив **#ifdef** и **#ifndef**:

|  |
| --- |
| #ifdef \_идентификатор\_#ifndef \_идентификатор\_ |

Если **\_идентификатор**\_, указанный в директиве **#ifdef** определен ранее с помощью директивы **#define**, то компилируются операторы и выполняются директивы препроцессора, заключенные между директивами **#ifdef** и **#else**, а операторы и директивы препроцессора между **#else** и **#endif** игнорируются. Если директива **#else** отсутствует, компилируются операторы заключенные между директивами **#ifdef** и **#endif**. Если \_идентификатор\_ не определен, операторы, заключенные между директивами **#ifdef** и **#else** игнорируются, а операторы, заключенные между директивами **#else** и **#endif** компилируются . Директива **#ifndef** работает в  
точности наоборот: если \_идентификатор\_, указанный в директиве **#ifndef**, не определен ранее с помощью директивы **#define**, компилируются операторы, следующие за **#ifndef**, а если определен - операторы, следующие за **#else** (если директива **#else** присутствует).

# **24. Дополнительные приемы программирования. Дополнительные операции препроцессора. Операция подстановки строки (#). Операция конкатенации (##). Операция подстановки символа (#@).**

С предоставляет два оператора препроцессора: **#** и **##**. Данные операторы используются в совокупности c **#define**. Оператор **#** помещает аргумент, перед которым он стоит, в двойные кавычки. Например, рассмотрим программу:

#include "stdio.h"

#define mkstr(s) # s

int main(void)

{

printf(mkstr(I like C) );

return 0;

}

Препроцессор превратит строку

printf(mkstr(I like C));

в

строку printf ("I like С");

Оператор **##** используется для конкатенации двух частей. Например:

#include "stdio.h"

#define concat(a, b) a ## b i

nt main(void)

{

int xy = 10;

printf("%d", concat(x, y));

return 0;

}

Препроцессор преобразует

printf("%d", concat(x, y) );

в

printf("%d", xy);

# **25.Дополнительные приемы программирования. Обработка ошибок: perror(). Модели памяти. Модель tiny. Модель small. Модель medium. Модель compact. Модель large. Модель huge.**

Функция **perror**.

Прототип функции **perror:**

**void perror( const char \* string );**

Описание

Функция perror интерпретирует значение глобальной переменной ERRNO в строку и выводит эту строку на стандартный поток вывода с сообщением, указанным в параметре string.

Макрос ERRNO является глобальной переменной, которая содержит значение об ошибке 02, полученной при вызове библиотечных функций. Значение ошибки переводится в строковое сообщение функцией perror, содержание сообщения зависит от платформы и компилятора.

Если параметр string не является нулевым указателем, то перед выводом ошибки, сначала печатается сообщение содержащееся в string, после — двоеточие **(:)**  и символ пробела. Когда string — нулевой указатель, тогда, генерируется сообщение об ошибке и выводится на экран. В конце сообщения следует символ новой строки '\n'.

Функция perror должна быть вызвана сразу после того, как возникла ошибка, в противном случае, ошибка может быть перезаписана при вызове других функций.

Параметры:

* **string**  
  Строка содержащая дополнительное сообщение, которое выводится перед сообщением об ошибке. Если параметр — нулевой указатель, то никакого дополнительного сообщения не печатается, но сообщение об ошибке печатается. Чаще всего в качестве этого параметра используется имя приложения.

Возвращаемое значение

нет

Пример: исходный код программы

|  |  |
| --- | --- |
|  | #include <cstdio>   int main()  {    FILE \* ptrFile = fopen("nonexist.ent", "rb");     if (ptrFile == NULL)      perror("Произошла ошибка: ");    else      fclose (ptrFile);    return 0;  } |

Модели памяти

C++ поддерживает **6** моделей памяти:

1. Крошечная модель **tiny**  
2.Малая модель **small**  
3.Средняя модель **medium**  
4.Компактная модель **Compact**  
5. Большая модель **large**  
6.Гигантская модель **huge**

Модель **TINY**.

Общий объём памяти для кода, данных и стека – **64Kбайта**. Все указатели – **ближние**.\

Модель **SMALL**.

Общий объём памяти для кода – **64Kбайта**, для данных и стека – **64Kбайта**. Все указатели по умолчанию – ближние, но для данных могут использоваться и дальние указатели. Начиная с этой модели в программе появляется возможность использования дальней кучи (**far heap**) для динамического использования памяти. К дальней куче могут обращаться только дальние указатели.

Модель **MEDIUM**.

Рекомендуется для больших программ с малым количеством статических данных. Общий объём памяти для кода каждого модуля – **64Kбайта**, для данных и стека – **64Kбайта**. Указатели данных по умолчанию – **ближние**, указатели функций по умолчанию – **дальние**.

Модель **COMPACT**.

Рекомендуется в случае с малым объёмом кода, но большим объёмом данных. Общий объём памяти для кода – **64Kбайта**, для данных – **64Kбайта**, для стека – **64Kбайта**. Указатели данных по умолчанию – **дальние**, указатели функций по умолчанию – **ближние.** Начиная с этой модели отсутствует ближняя куча.

Модель **LARGE**.

Общий объём памяти для кода каждого модуля – **64K**, для данных – **64K**, для стека – **64K**. Все указатели – **дальние**.

Модель **HUGE**.

Общий объём памяти для кода каждого модуля – **64K**, для данных каждого модуля – **64K**, для стека – **64K**. Все указатели – **дальние**.

Для «**малых**» моделей все указатели типа **near**, для больших — **far**. Для указателей на функции **near** для **tiny, small,compact, far** в остальных. **DS** для **near** указывает на данные, **CS** -для **near** указывает на функции.  
Размер кода тоже не может превышать **64кб**. Для любого модуля заводится свой сегмент кода. Общий размер памяти выделяется для хранения кода не более **1 Мб** для **medium, large, huge**.  
Если все функции в одном файле, то все указатели типа **near**. Если несколько модулей, но они не обращаются друг к другу, тоже самое. Но если есть обращения функций одного модуля к функциям другого, они должны быть описаны как **far** функции.

# **26. Дополнительные приемы программирования. Модификаторы функций. Модификаторы cdecl и pascal.**

При компиляции все глобальные идентификаторы программы, т. е. имена функций и глобальных переменных, сохраняются в объектном коде и используются компоновщиком в процессе работы. По умолчанию эти идентификаторы сохраняются в своем первоначальном виде . Кроме того, в качестве первого символа каждого идентификатора компилятор языка Си добавляет символ подчеркивания.

Компоновщик по умолчанию различает прописные и строчные буквы, поэтому идентификаторы, используемые в различных исходных файлах программы для именования одного и того же объекта, должны полностью совпадать с точки зрения, как орфографии, так и регистров клавиатуры. Для обеспечения совпадения идентификаторов, используемых в разноязычных исходных файлах, применяются модификаторы **pascal** и **cdecl**.

**Модификатор pascal**

Применение модификатора **pascal** к идентификатору приводит к тому, что идентификатор преобразуется к верхнему регистру и к нему не добавляется символ подчеркивания.

Если модификатор **pascal** применяется к идентификатору функции, то он оказывает влияние также и на передачу аргументов. Засылка аргументов в стек производится в этом случае не в обратном порядке, а в прямом—первым засылается в стек первый аргумент(слева направо).

Функции типа **pascal** не могут иметь переменное число аргументов, как, например, функция **printf**. Поэтому нельзя использовать завершающее многоточие в списке параметров функции типа **pascal**.

*Пример:* char far pascal initlist[INITSIZE];

char far nextchar, far \*prevchar, far \*currentchar;

В примере также представлены два объявления. Первое объявляет массив типа **char** с именем **initiist** и модификаторами **far** и **pascal**. Модификатор **pascal** указывает на то, что имя данного массива используется не только в программе на языке Си, но и в программе на языке Паскаль. Модификатор **far** указывает на то, что для доступа к элементам массива долины использоваться 32-битовые адреса.

Второе объявление объявляет три указателя на **far** значения типа **char** с именами **nextchar**, **prevchar** и **currentchar**. Эти указатели могут быть, в частности, использованы для хранения адресов элементов массива **initlist**. Обратите внимание на то, что специальное ключевое слово **far** должно быть повторено перед каждым описателем.

**Модификатор** cdecl

можно указать, что некоторые функции и указатели на функции используют вызывающую последовательность, принятую в языке Си, а их идентификаторы имеют традиционный вид для идентификаторов языка Си. Для этого их объявления должны содержать модификатор **cdecl**.

**Cdecl – запрещает или разрешает распознавание заглавных и прописных букв.**

**Замечание: модификатор cdecl может быть реализован напрямую за счет использования опций настройки среды программирования.**

*Примечание.* Все функции в стандартных включаемых файлах объявлены с модификатором **cdecl**. Это позволяет использовать библиотеки стандартных функций даже в тех программах, которые компилируются с упомянутой выше опцией компиляции.

*Примечание.* Главная функция программы (**main**) должна быть всегда объявлена с модификатором **cdecl**, поскольку модуль поддержки выполнения передает ей управление, используя вызывающую последовательность языка Си.

*Пример:* double near cdecl calc(double, double);

double cdecl near calc(double, double);

В примере показано два эквивалентных объявления. В них объявляется **calc** как функция с модификаторами near и **cdecl**.

# **27. Файлы в C. Файлы и потоки. Связь с файлами. Закрытие потоков. Обработка ошибок в C и C++. Переменная errno и коды ошибок.**

Связь с файлами

функция **fopen()**, которая открывает файл, затем применяются специальные функции ввода-вывода для чтения файла или записи в этот файл и далее используется функция **fclose()** для закрытия файла.

**Файл** – это набор данных, размещенный на внешнем носителе и рассматриваемый в процессе обработки как единое целое. В файлах размещаются данные, предназначенные для длительного хранения.

Когда файл открывается, то создается **объект** и с этим объектом связывается файловый поток.

**Файловый поток** – это файл вместе с предоставленными средствами буферизации.

Потоки можно:

* открывать и закрывать;
* вводить и выводить строки, символы, форматированные данные, порции данных произвольной длины;
* анализировать ошибки ввода/вывода и достижения конца файла;
* управлять буферизацией потока и размером буфера;
* получать и устанавливать указатель текущей позиции в файле.

Прежде чем начать работать с потоком, его надо инициировать (**открыть**).

При этом поток связывается со структурой **FILE**, определение которой находится в библиотечном файле <**stdio.h**>.

В структуре **FILE** находится указатель на буфер, указатель на текущую позицию файла и т.п.

Для открытия файла(потока) используется функция

**fopen**(<имя\_файла>, <режим\_открытия>)

При открытии потока возвращается указатель на поток, т.е. на объект типа **FILE**:

**#include <stdio.h>**

**int main()**

**{ …………**

**FILE \*fp;** //дескриптор

**………**

**fp = fopen(“t.txt”, ”r”);**

**………**

Здесь файл находится в папке примера.

Если файл расположен в другом месте, то надо указывать полный путь к нему:

**“F:\p1\p2\a.txt”**

Если открытия файла не произошло, то функция **fopen** вернет **NULL**.

После работы с файлом его надо **закрыть** с помощью функции(закрывает поток):

fclose ( <указатель на поток> );

Она разрывает связь структурированной переменной с файлом и записывает в память оставшееся содержимое буфера.

Пример:

**#include <stdio.h>**

**……………….**

**FILE \*fd;**

**fd = fopen("aaa.txt", "r");**

**if (fd == NULL)**

**printf("Файл не открыт\n");**

**else**

**{ fscanf(fd, ....);**

**. . . . . . . . . . . . . .**

**fclose(fd);**

**}**

**Обработка ошибок**

В процессе открытия или создания файла мы можем столкнуться с рядом ошибок, например, при открытии в режиме чтения не окажется подобного файла, недостаточно памяти и т.д. И в случае возникновения ошибки функция fopen() возвращает значение NULL. Мы можем обработать возникновение ошибки с помощью проверки результата функции:

|  |  |
| --- | --- |
|  | #include <stdio.h>  #include <stdlib.h>   int main(void)  {      FILE \* fp;      if((fp= fopen("D:\data28.txt", "r"))==NULL)      {          perror("Error occured while opening data28.txt");          exit(0);      }        fclose(fp);      return 0;  } |

Для вывода ошибки на консоль применяется встроенная функция **perror()**. И так как дальнейшие действия в программе в случае ошибки при открытии файла смысла не имеют, то с помощью вызова **exit(0)** завершаем работу приложения.

**Переменная errno и коды ошибок**

[**<errno.h>**](https://en.wikipedia.org/wiki/Errno.h)

**errno** – переменная, хранящая целочисленный код последней ошибки. Обычно **errno** реализуется в виде макроса, разворачивающегося в вызов функции, возвращающей указатель на целочисленный буфер. При запуске программы значение **errno** равно нулю.

Все коды ошибок имеют положительные значения, и могут использоваться в директивах препроцессора **#if**. В целях удобства и переносимости заголовочный файл <errno.h> определяет макросы, соответствующие кодам ошибок.

Стандарт ISO C определяет следующие коды:

* **EDOM** – (**E**rror **dom**ain) ошибка области определения.
* **EILSEQ** – (**E**rror **i**nva**l**id **seq**uence) ошибочная последовательность байтов.
* **ERANGE** – (**E**rror **range**) результат слишком велик.

.

Если вызов функции завершился ошибкой, то она устанавливает переменную **errno** в ненулевое значение. Если же вызов прошёл успешно, функция обычно не проверяет и не меняет переменную **errno**. Поэтому перед вызовом функции её нужно установить в 0.

# **28.Файлы в C. Переключение и работа с файлами. Переключение вывода. Переключение ввода. Комбинированное переключение. Конвейерная пересылка. Текстовые и бинарные (двоичные) файлы.**

Файл – именованная область внешней памяти, выделенная для хранения массива данных. При открытии файла с ним связывается поток ввода-вывода. Выводимая информация записывается в поток, вводимая информация считывается из потока. Когда поток открывается для ввода-вывода, он связывается со стандартной структурой типа FILE, которая определена в stdio.h. Структура FILE содержит необходимую информацию о файле. Открытие файла осуществляется с помощью функции fopen(), которая возвращает указатель на структуру типа FILE, который можно использовать для последующих операций с файлом. FILE \*fopen(name, type); name– имя открываемого файла (включая путь), type— указатель на строку символов, определяющих способ доступа к файлу.

Возвращаемое значение — указатель на открытый поток. Если обнаружена ошибка, то возвращается значение NULL. Функция fclose() закрывает поток или потоки, связанные с открытыми при помощи функции fopen() файлами. Если достигнут конец файла или возникла ошибка, возвращается константа EOF. Запись символа в файл:

fputc(символ,поток);

Функции fscanf()и fprintf() работают с файлами данных, и имеют первый аргумент — указатель на файл. fscanf(поток,«ФорматВвода»,аргументы);

fprintf(поток, «ФорматВывода», аргументы);

Функции fgets() и fputs() предназначены для ввода-вывода строк. fgets(УказательНаСтроку,КоличествоСимволов,поток);

Символы читаются из потока до тех пор, пока не будет прочитан символ новой строки ‘\n’, который включается в строку, или пока не наступит конец потока EOF или не будет прочитано максимальное символов. Результат помещается в указатель на строку и заканчивается нуль-символом ‘\0’. Функция возвращает адрес строки.

fputs(УказательНаСтроку,поток);

Переключение ввода

чтобы слова вводимые с клавиатуры попадали в файл с именем mywords нужно ввести команду

getput4 > mywords

и начать ввод символов Символ > служит обозначением еще одной операции переключения , используемой в ОС. Ее выполнение приводит к тому что создается новый файл с именем UNIX. , mywords, а затем результат работы программы ввод вывод - 4, представляющий собой копию вводимых символов направляется в данный файл Если файл с именем , . mywords уже существует он обычно уничтожается и вместо него создается новый В некоторых реализациях , , . в данном случае появятся лишь вводимые вами символы их же копии будут направлены в ; указанный файл Чтобы закончить работу программы введите признак . , EOF; UNIX в системе это обычно символ Попробуйте воспользоваться описанной здесь операцией Если вам [CTRL/d]. . ничего другого не придет в голову просто воспроизведите на своей машине пример , , приведенный ниже Знак приглашения выводимый на экран интерпретатором команд . , SHELL, обозначается здесь символом %. Не забывайте оканчивать каждую введенную строку символом [ ], . возврат чтобы содержимое буфера пересылалось в программу

% getput4 > mywords

у вас не должно быть трудностей с запоминанием того какая операция переключения для чего , предназначена Необходимо помнить только что знак каждой операции указывает на . , направление информационного потока Вы можете по ассоциации представлять себе этот знак в . виде воронки . [CTRL/d]

После того как введен символ [CTRL/d], программа заканчивает свою работу и возвращает управление операционной системе на что указывает повторное появление знака UNIX, приглашения Как убедиться в том что наша программа вообще работала В ОС . , ? UNIX существует команда Is, ; которая выводит на экран имена файлов обращение к ней должно продемонстрировать вам что файл с именем , mywords теперь существует Чтобы проверить его . содержимое вы можете воспользоваться командой , cat или запустить заново программу ввод- вывод4, . направляя в нее на этот раз содержимое входного файла

% getput4 < mywords

**Комбинированное переключение**

Предположим, что мы хотим создать копию файла my\_words и назвать его my\_words2. Нужно ввести для этого команду

get\_put < my\_words > my\_words2

и требуемое задание будет выполнено.

Команда

get\_put > my\_words2 < my\_words

приведет к такому же результату, поскольку порядок указания операций переключения не имеет значения. Нельзя в одной команде использовать один и тот же файл и для ввода, и для вывода.

**Бинарные файлы**

Текстовые файлы хранят данные в виде текста . Это значит, что если, например, мы записываем целое число 12345678 в файл, то записывается 8 символов, а это 8 байт данных, несмотря на то, что число помещается в целый тип. Кроме того, вывод и ввод данных является форматированным, то есть каждый раз, когда мы считываем число из файла или записываем в файл происходит трансформация числа в строку или обратно. Это затратные операции, которых можно избежать. Текстовые файлы позволяют хранить информацию в виде, понятном для человека. Можно, однако, хранить данные непосредственно в бинарном виде. Для этих целей используются бинарные файлы.

Число, которое ввёл пользователь записывается в файл непосредственно в бинарном виде. Запись в файл осуществляется с помощью функции

size\_t fwrite ( const void \* ptr, size\_t size, size\_t count, FILE \* stream );

Функция возвращает число удачно записанных элементов. В качестве аргументов принимает указатель на массив, размер одного элемента, число элементов и указатель на файловый поток. Для считывания используется функция fread

size\_t fread ( void \* ptr, size\_t size, size\_t count, FILE \* stream );

Функция возвращает число удачно прочитанных элементов, которые помещаются по адресу ptr. Всего считывается count элементов по size байт. Одной из важных функций для работы с бинарными файлами является функция fseek

int fseek ( FILE \* stream, long int offset, int origin );

Эта функция устанавливает указатель позиции, ассоциированный с потоком, на новое положение. Индикатор позиции указывает, на каком месте в файле мы остановились. Когда мы открываем файл, позиция равна 0. Каждый раз, записывая байт данных, указатель позиции сдвигается на единицу вперёд. fseek принимает в качестве аргументов указатель на поток и сдвиг в offset байт относительно origin. origin может принимать три значения

•SEEK\_SET - начало файла

•SEEK\_CUR - текущее положение файла

•SEEK\_END - конец файла.

# **29. Файлы в C. Потоковый ввод-вывод. Соединение и отсоединение потока от файла. Функция fopen. Функция fclose. Функция freopen.**

Выводимая информация записывается в поток, вводимая информация считывается из потока.

Когда поток открывается для ввода-вывода, он связывается со стандартной структурой типа FILE, которая определена в stdio.h. Структура FILE содержит необходимую информацию о файле.

Открытие файла осуществляется с помощью функции fopen(), которая возвращает указатель на структуру типа FILE, который можно использовать для последующих операций с файлом.

**FILE \*fopen(name, type);**

**name** –имя открываемого файла (включая путь), type — указатель на строку символов, определяющих способ доступа к файлу:

* **«r»** — открыть файл для чтения (файл должен существовать);
* **«w»** — открыть пустой файл для записи; если файл существует, то его содержимое теряется;
* **«a»** — открыть файл для записи в конец (для добавления); файл создается, если он не существует;
* **«r+»** — открыть файл для чтения и записи (файл должен существовать);
* **«w+»** — открыть пустой файл для чтения и записи; если файл существует, то его содержимое теряется;
* **«a+»** — открыть файл для чтения и дополнения, если файл не существует, то он создаётся.

Возвращаемое значение — указатель на открытый поток. Если обнаружена ошибка, то возвращается значение NULL.

Функция **fclose()** закрывает поток или потоки, связанные с открытыми при помощи функции fopen() файлами. Закрываемый поток определяется аргументом функции fclose().

Возвращаемое значение: значение 0, если поток успешно закрыт; константа EOF, если произошла ошибка.

*#include <stdio.h>   
int main() {*

*FILE \*fp;*

*char name[]=«my.txt»;*

*if(fp = fopen(name, «r»)!=NULL) { // открыть файл удалось?  
…                 // требуемые действия над данными*

*} else printf(«Не удалось открыть файл»);*

*fclose(fp);*

*return 0;  
}*

Для закрытия **нескольких** файлов введена функция:

**void fcloseall(void);**

Для этого используют станд. функцию:

**FILE \*freopen(char\***имя\_файла**, char \***режим**,  FILE \***дескриптор**\_**файла**);**

Эта функция переназначает указатель потока: сначала закрывает файл, объявленный дескриптором (как это делает функция **fopen**), а затем открывает файл с именем\_файла и правами доступа.

# **30. Файлы в C. Работа с индикаторами ошибки, позиции и конца файла (ferror, clearerr, feof, rewind, fseek, fsetpos, ftell, fgetpos). Блочный ввод-вывод (fwrite, fread).**

1. **ferror (FILE \*stream); //** тестирует поток на ошибки чтения. Если в файле была обнаружена ошибка, то функция возвращает ненулевое значение.

**stream** - указатель на управляющую таблицу открытого потока данных.

1. **clearerr (FILE \*stream);** // Сбрасывает индикатор ошибок потока и устанавливает в ноль индикатор конца файла (end-of-file)

**stream** - указатель на управляющую таблицу открытого потока данных.

1. **feof (FILE \*stream );** // проверяет наличие установленного признака конца файла. Используется для определения достижения конца файла при чтении данных из файла.

**stream** – указатель на управляющую таблицу потока данных, к которому привязан проверяемый файл.

**Позиционирование в файле**

**Текущая позиция** – это номер байта, начиная с которого производится очередная операция чтения/записи.

При **открытии** файла текущая позиция устанавливается на **начало** файла, после чтения-записи порции данных перемещается вперед на **размерность** этих данных.

Текущая позиция представляется в программе переменной типа **long**.

Для работы с ней в станд. библиотеке имеются две функции:

1. **long ftell(FILE \*fp);**

возвращает текущую позицию в файле.

Вторая функция устанавливает текущую позицию в файле на байт с номером **pos**:

1. **int  fseek(FILE \*fp, long pos, int mode);**

Возвращает 0 при успешном позиционировании и **-1 (EOF)** - в случае неудачи.

Параметр **mode** определяет, относительно чего отсчитывается текущая позиция в файле и имеет следующие значения (установленные в **stdio.h**):

**#define SEEK\_SET 0** // Относительно начала файла

// начало файла - позиция 0

**#define SEEK\_CUR 1**  // Относит. текущей позиции,

// >0 - вперед, <0 - назад

**#define SEEK\_END 2** // Относительно конца файла

// (знач. pos – отрицательное)

Получить текущую **длину** **файла** можно позиционированием:

*long fsize;*

*fseek(f1, 0L, SEEK\_END); //устан. на конец файла*

*fsize = ftell(f1); //прочитать знач. текущей позиции*

Еще одна функция позиционирования в файле на начало потока: **rewind (fp);**

Функция эквивалентна функции **fseek** за исключением того, что сбрасывается индикатор конца файла и индикатора ошибок. После функции **rewind** может выполнять операции обновления файлов.

Пример. Определить размер файла **a.txt** в байтах.

*#include <stdio.h>*

*void main()*

*{ FILE \*fd;*

*long int pos;*

*fopen\_s(&fd, "a.txt", "r");*

*fseek (fd, 0, SEEK\_END); //0 байт от конца*

*pos = ftell (fd); //Текущая позиция в файле*

*if (pos < 0) puts ("\nError");*

*else if (!pos) puts ("\nFile is empty");*

*else printf ("\nSize= %ld", pos);*

*}*

1. **rewind ( FILE \* filestream );**

|  |  |
| --- | --- |
|  |  |

Устанавливает внутренний указатель положения файла в начальное .

Эквивалентный вызов функции fseek для возвращения указателя на начало:

fseek( stream , 0L , SEEK\_SET );

|  |  |
| --- | --- |
|  |  |

**filestream -** Указатель на объект типа FILE, идентифицируемый поток.

1. **fsetpos( FILE \* filestream, const fpos\_t \* pos );**

Функция fsetpos перемещает внутренний указатель положения в файле, связанный с потоком, на новую позицию. Параметр pos является указателем на объект типа fpos\_t, значение которого предварительно должно быть получено с помощью вызова функции fgetpos.  
Внутренний индикатор конца файла EOF очищается после вызова этой функции, и все эффекты от предыдущих вызовов функции ungetc удаляются. Для потоков, открытых в режиме обновления (чтения + записи), вызов функции fsetpos позволяет переключаться между режимами чтения и записи.

* **filestream -** Указатель на объект типа FILE, идентифицируемый поток.
* **position -** Указатель на объект типа fpos\_t содержащий позицию указателя, ранее полученную с помощью функции fgetpos.

1. **fgetpos( FILE \* filestream, fpos\_t \* position );**

Функция fgetpos возвращает текущую позицию в файле. Через параметры filestream и position функция получает информацию, необходимую для идентификации текущего положения внутреннего указателя потока.  
Параметр position должен указывать на уже существующий объект типа fpos\_t, который предназначен только для использования, в качестве параметра функции fsetpos.  
Чтобы получить целочисленное значение типа int внутреннего индикатора позиции файла,  используйте функцию ftell.

**filestream-**Указатель на объект типа FILE, идентифицируемый поток.

**position -** Указатель на объект типа fpos\_t.

1. **int fread( void \*ptr, int size, int n, FILE \*fp); //**считывает неформатированные данные из потока

**void \*ptr** – указатель на область памяти, в которой размещаются считываемые из файла данные;

**int size** – размер одного считываемого элемента;

**int n** – количество считываемых элементов;

**FILE \*fp** – указатель на файл, из которого производится считывание.

1. **int fwrite( void \*ptr, int size, int n, FILE \*fp); //**записывает неформатированные данные в поток

**void \*ptr** – указатель на область памяти, в которой размещаются записываемые в файл данные;

**int size** – размер одного записываемого элемента;

**int n** – количество элементов;

**FILE \*fp** – указатель на файл, в который производится запись.

# **31. Файлы в C. Символьный ввод-вывод. Функции fputc, putc, fgetc, getc, ungetc. Функции fputs и fgets. Функции fprintf, fscanf.**

Две функции **getc()** и **putc()** работают аналогично функциям **getchar()** и **putchar()**. Разница заключается в том, что вы должны сообщить, какой файл следует использовать. Таким образом,

**ch = getchar()**;

предназначен для получения символа от стандартного ввода, а

**ch = getc(in);**

— для получения символа от файла, на который указывает **in**. Аналогично функция

**putc(ch, out);**

предназначена для записи символа **ch** в файл, на который ссылается указатель **out** типа **FILE**. В списке аргументов функции **putc()** этот символ стоит первым, а затем указатель файла.

* 1. **Прототип функции putc:**

|  |  |
| --- | --- |
|  | **int putc( int character, FILE \* filestream );** |

Функция putc помещает символ character в поток и перемещает позицию индикатора текущего положения.

* 1. **Прототип функции getc:**

|  |  |
| --- | --- |
|  | **int getc ( FILE \* filestream );** |

Функция возвращает символ из потока filestream, на который ссылается внутренний индикатор позиции файла.

* 1. **Прототип функции ungetc:**

|  |  |
| --- | --- |
|  | **int ungetc( int character, FILE \* filestream );** |

Функция ungetc возвращает только что прочитанный символ обратно в поток ввода filestream, через параметр character.

Параметр character  может содержать любой символ, например, последний символ прочитанный из потока в предыдущей операции или любой другой. В обоих случаях, значение, полученное по следующей операции чтения является значением функции ungetc, независимо от символа character.

* 1. **Чтение символа из файла**:

**char fgetc(поток);**

Аргументом функции является указатель на поток типа FILE. Функция возвращает код считанного символа. Если достигнут конец файла или возникла ошибка, возвращается константа EOF.

* 1. **Запись символа в файл**:

**fputc(символ,поток);**

Аргументами функции являются символ и указатель на поток типа FILE. Функция возвращает код считанного символа.

* 1. **fgets(УказательНаСтроку,КоличествоСимволов,поток);**

Возвращает строку из потока

* 1. **fputs(УказательНаСтроку,поток);**

Записывает строку в поток

* 1. **fscanf(поток, «ФорматВвода»,аргументы);**

Считывает форматированные данные из потока

* 1. **fprintf(поток, «ФорматВывода», аргументы);**

Выполняет форматированную запись в поток

# **32. Файлы в C. Работа с буферами (setvbuf, setbuf, fflush). Стандартные потоки (putchar, getchar, puts, gets, perror). Служебные функции для работы с файлами (remove, rename, tmpfile, tmpnam).**

**Работа с буферами (setvbuf, setbuf, fflush).**

Иногда, перед критическими операциями имеет смысл сбросить данные, хранящиеся в буферах обмена потоков на диск.

**int fflush(указатель\_на\_поток);**- записывает информацию хранящуюся в буфере в файл, поток при этом не закрывается.. **fflush(stdin)**– очистить входной поток.

Для управления размерами буфера можно использовать следующие функции:

**void setbuf(FILE \*stream, char \*buf) - устанавливает буферизацию потока**

**int setvbuf(FILE \*stream, char \*buf, int type, size\_t size) - назначает буферизацию потока**

* **stream**Указатель на объект типа FILE, который соединён с открытым потоком.
* **buf**Выделенный пользователем буфер..
* **type**

Когда программа начинает выполняться, автоматически открываются пять потоков:

−стандартный поток ввода (на него ссылаются, используя предопределенный указатель на поток **stdin);**

−стандартный поток вывода **(stdout);**

−стандартный поток вывода сообщений об ошибках **(stderr).**

По умолчанию стандартному потоку ввода **stdin**ставится в соответствие клавиатура, а потокам **stdout и stderr**соответствует экран дисплея.

Для ввода-вывода данных с помощью стандартных потоков в библиотеке языка Си определены следующие функции:

−**getchar( )/putchar(**) -ввод-вывод отдельного символа;

−**gets( )/puts()**-ввод-вывод строки;

−**scanf( )/printf(**) -ввод-вывод в режиме форматирования данных.

Ввод-вывод одного символа осуществляется с помощью библиотечных функций getchar() и putchar(). **Прототипы функций:**

**int getchar(void);**

**int putchar(int с);**

Функция **getchar()** осуществляет ввод одного символа.

Функция **putchar( )** выводит в стандартный поток один символ, при этом также возвращает в вызвавшую ее функцию только что выведенный символ.

Функция **getchar( )** вводит очередной байт информации (символ) в виде значения типа **int** ,что гарантирует успешность распознавания ситуации "**достигнут конец файла**".

Функция **putchar(**) служит для вывода на устройство стандартного вывода одного символа, заданного в качестве параметра.

**#include <stdio.h>**

**int main() {**

**int c;**

**while {(c = getchar()) != EOF)**

**putchar(c);**

**return 0;**

**}**

В библиотеку Си для обмена данными через стандартные потоки ввода-вывода включены функции ввода-вывода строк **gets() и puts(**), прототипы:

|  |  |  |  |
| --- | --- | --- | --- |
| **char \* gets (char \* s);** | **/\*** | **Функция** | **ввода \*/** |
| **int puts (char \* s);** | **/\*** | **Функция** | **вывода\*/** |

Функция **puts()**в случае успешного завершения возвращает последний выведенный символ, который всегда является символом '\n'. Если произошла ошибка, то возвращается **EOF.**

Пример :

**#include <stdio.h>**

**int main() {**

**char name[80];**

**char strl[ ] = "Введите фамилию сотрудника:";**

**puts(strl);**

**gets(name);**

**return 0; }**

Функция **gets( )** начинает обработку информации от клавиатуры только после нажатия клавиши **<Enter>.** Таким образом, она **"ожидает",** пока не будет набрана нужная информация и нажата клавиша **<Enter>.** Только после этого начинается ввод данных в программу.

**Функция**perror(**)**

Для вывода на экран дисплея сообщения об ошибке при открытии потока используется стандартная библиотечная функция **perror(),** прототип:

**void perror (const char \* s);**

Функция **perror()** выводит строку символов, адресуемую указателем **s**, за которой размещаются: **двоеточие, пробел и сообщение об ошибке**.

**int remove( const char \* fname );**

Функция **remove** удаляет файл, имя которого указано в параметре **fname**.

**int rename( const char \* oldfilename, const char \* newfilename );**

Функция **rename**изменяет имя файла или каталога, указанного в **oldfilename** на новое имя, указанное в **newfilename**.

Функция:

**FILE\* tmpfile (void);**

которая создает на диске временный файл с правами доступа **w+b**.

**char \* tmpnam( char \* string );**

Функция **tmpnam** создает имя для файла.   
Эта строка может быть использована для создания временных файлов без перезаписи любого другого существующего файла.

# **33.Файлы в C. Низкоуровневый ввод и вывод в С. Форматированный вывод. Использование функций printf() и fprintf().**

Низкоуровневый ввод-вывод (read и write) Ввод-вывод основан на системных вызовах read и write, к которым программа на Си обращается с помощью функций с именами read и write. Для обеих первым аргументом является дескриптор файла. Во втором аргументе указывается массив символов вашей программы, куда посылаются или откуда берутся данные. Третий аргумент — это количество пересылаемых байтов.

int n\_read = read(int fd, char \*buf, int n);

int n\_written = write(int fd, char \*buf, int n);

Обе функции возвращают число переданных байтов.

Функция форматированного вывода printf получает в качестве аргументов строку формат и аргументы, которые необходимо вывести в соответствии с форматом, и возвращает число выведенных символов. В случае ошибки возвращает отрицательное значение и устанавливает значение ferror. Если произошло несколько ошибок, errno равно EILSEQ. int printf (const char \* format, ...);

#include <stdio.h>

#include <conio.h>

void main() {

//функция не получает никаких аргументов, кроме строки

printf("Hello world"); getch();

}

Функция проходит по строке и заменяет первое вхождение % на первый аргумент, второе вхождение % на второй аргумент и т.д. Далее мы будем просто рассматривать список флагов и примеры использования. Спецификатор – это самый важный компонент. Он определяет тип переменной и способ её вывода.

Рассмотрим форматированный ввод функцией scanf. int scanf(const char\*, ...) Функция принимает строку формата ввода (она похожа на строку формата printf) и адреса, по которым необходимо записать считанные данные. Возвращает количество успешно проинициализированных аргументов. Формат спецификатора ввода

%[\*][ширина][длинна] [спецификатор]

#include <stdio.h>

#include <conio.h>

void main()

{

int year, month, day;

char buffer[128];

int count;

//Требует форматированного ввода, например 2013:12:12

printf("Enter data like x:x:x = ");

scanf("%d:%d:%d", &year, &month, &day);

printf("year = %d\nmonth = %d, day = %d\n", year, month, day);

//Считываем строку, не более 127 символов. При считывании в массив писать & не надо, //так как массив подменяется указателем

printf("Enter string = ");

scanf("%127s", buffer);

printf("%s", buffer);

getch();

}

Помимо основных функций ввода/вывода, система буферизированного ввода/вывода содержит fprintf() и fscanf(). Данные функции ведут себя так же, как и printf() и scanf(), за тем исключением, что работают с дисковыми файлами. Они имеют следующие прототипы:

int fprintf(FILE \*fp, const char \*форматная\_строка, ...);

int fscanf(FILE \*fp, const char \* форматная\_строка, ...);

где fp - это указатель на файл, возвращаемый fopen().

# **34. Файлы в C++. Структура стандартной библиотеки ввода-вывода. Предопределенные потоки. От файла STREAM.H к файлу IOSTREAM.H. Список классов iostream.**

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

Для работы с файлами необходимо подключить заголовочный файл <fstream> (файл fstream уже включает iostream, так что включать оба файла необязательно). В <fstream> определены несколько классов и подключены заголовочные файлы <ifstream> — файловый ввод и <ofstream>  — файловый вывод.

В языке C++ объекты для работы с файлами называются **потоками** (streams). В данном случае слово "поток" означает то же самое, что и "файл" в языке C.

Классы для работы с файлами в языке C++ называются std::istream и std::ostream для ввода и вывода соответственно.

Файловый ввод/вывод аналогичен стандартному вводу/выводу, единственное отличие – это то, что ввод/вывод выполнятся не на экран, а в файл.

iostream содержит два основных класса:

* ***cin*** — для обработки ввода с клавиатуры;
* ***cout*** — для вывода в консоль переменных или просто текста;

Есть еще классы cerr и clog, но их в целом, используют не так часто. Если кому интересно — эти классы используют для вывода ошибок при операциях и для логирования действий.

*Например*, необходимо создать текстовый файл и записать в него строку *Работа с файлами в С++*. Для этого необходимо проделать следующие шаги:

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

fout.open("file.txt"); // связываем объект с файлом

fout << "Работа с файлами в С++"; // запись строки в файл

fout.close(); // закрываем файл

// создаём объект для записи в файл

ofstream /\*имя объекта\*/; // объект класса ofstream

# **35. Файлы в C++. Простота ввода/вывода в С ++. Потоки cin, cout и сеrr. Операции выделения ( >> ) и вставки ( << ). Опции ввода/вывода в С/С++.**

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

Для работы с файлами необходимо подключить заголовочный файл <fstream> (файл fstream уже включает iostream, так что включать оба файла необязательно). В <fstream> определены несколько классов и подключены заголовочные файлы <ifstream> — файловый ввод и <ofstream>  — файловый вывод.

Файловый ввод/вывод аналогичен стандартному вводу/выводу, единственное отличие – это то, что ввод/вывод выполнятся не на экран, а в файл.

*cin* является потоком, ассоциированным со стандартным вводом, а *cout* представляет собой поток, ассоциированный со стандартным выводом. Потоки *cerr* и *clog* используются для вывода сообщений об ошибках. Разница между *cerr* и *clog* заключается в том, что, хотя они оба привяза­ны к стандартному выводу, *cerr* не буферизирован, поэтому все посланные в него данные выво­дятся немедленно. В противоположность этому, *clog* буферизирован, так что данные выводятся только тогда, когда буфер оказывается полным.

Основными классами ввода-вывода C++ являются istream и ostream. Первый из них перегружает операцию правого сдвига (**>>**), которая служит в нем для ввода данных и называется операцией извлечения из потока. Класс ostream перегружает соответственно операцию левого сдвига (**<<**); она применяется для вывода и называется операцией передачи в поток.

Операции извлечения и передачи в поток можно перегрузить таким образом, чтобы можно было применять их для ввода или вывода объектов класса, определенного пользователем.

std::ostream &operator << (std::ostream &os, int N);

* &os – поток, в который мы будем выводить;
* N – переменная, которую мы будем выводить;

Оператор возвращает ссылку на поток, в результате чего можно писать так: std::cout << N << M;

FILE \***fopen**(**const** **char** \*filename, **const** **char** \*mode) *// открытие файла*

**int** **fclose**(FILE \*stream) *// закрытие файла*

**int** **printf**(**const** **char** \*format, ...) *// форматированный консольный вывод*

**int** **fprintf**(FILE \*stream, **const** **char** \*formatб, ...) *// форматированный ввод из файла*

**int** **sprintf**(**char** \*s, **const** **char** \*format, ...) *// форматированный вывод в буфер (строку)*

**int** **scanf**(**const** **char** \*format, ...) *// форматированный консольный ввод*

**int** **fscanf**(FILE \*stream, **const** **char** \*format, ...) *// форматированный ввод*

**int** **sscanf** (**const** **char** \*s, **const** **char** \*format, ...) *// форматированный ввод из буфера (строки)*

**int** **fgetc**(FILE \*stream) *// читает символ из файла*

**char** \***fgets**(**char** \*s, **int** n, FILE \*stream) *// читает строку из файла*

**int** **fputc**(**int** с, FILE \*stream) *// записывает символ в файл*

**int** **fputs**(**const** **char** \*s, FILE \*stream) *// записывает строку в файл*

**int** **getchar**(**void**) *// читает символ из stdin*

**char** \***gets**(**char** \*s) *// читает строку из stdin*

**int** **putchar**(**int** с) *// записывает символ в stdout*

**int** **puts**(**const** **char** \*s) *// записывает строку в stdout*

**int** **ungetc**(**int** с, FILE \*stream) *// возвращает символ обратно в файл для последующего чтения*

# **36. Файлы в C++. Операции и методы классов. Вывод символов в C++. Преобразование системы счисления в C++. Форматирование строк в C++. Форматирование чисел в C++. Файловый ввод и вывод в C++.**

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

Для работы с файлами необходимо подключить заголовочный файл <fstream> (файл fstream уже включает iostream, так что включать оба файла необязательно). В <fstream> определены несколько классов и подключены заголовочные файлы <ifstream> — файловый ввод и <ofstream>  — файловый вывод.

Файловый ввод/вывод аналогичен стандартному вводу/выводу, единственное отличие – это то, что ввод/вывод выполнятся не на экран, а в файл.

|  |  |  |
| --- | --- | --- |
| Имя | Класс | Описание |
| cin | istream | Ассоциируется со стандартным вводом (клавиатурой). (для ввода с клавиатуры) |
| cout | ostream | Ассоциируется со стандартным выводом (экраном). (для буферизованного вывода на монитор) |
| cerr | ostream | Ассоциируется со стандартным устройством ошибок (экраном) без буферизации. (для небуферизованного вывода на монитор сообщений об ошибках) |
| clog | ostream | Ассоциируется со стандартным устройством ошибок (экраном)с буферизацией. (буферизованный аналог cerr) |

|  |  |  |
| --- | --- | --- |
| cin | get() | Позволяет ввести символ или строку. Метод **get()** ждет нажатия клавиши Enter. Только потом обрабатывает считываемое. При вводе символа в числовую переменную, метод возвращает код этого символа:  int c = cin.get();  // Получаем код символа  char c = cin.get(); // Или сам символ  **cin.get()** часто ставят в конце программы, чтобы задержать консоль с |
| Класс | Метод | Описание |
| cin | getline() | Аналогичен методу **get()**. Помимо всего, что «умеет» **get()**, переопределен для строк типа **string**. Так же как и **get()** умеет считывать до символа, указанного в качестве разделителя, так же первым параметром указывается массив символов, а вторым количество символов для считывания.  string s;  getline(cin,s); |
| cout | put() | Метод**put()**выводит символ в консоль:  cout.put('В');  Выведет один символ. |
| write() | Метод **write()**выведет блок символов из массива символов, переданный ему в качестве указателя:  #include <iostream>  using namespace std;    int main() {  char \*s = "Hello";  cout.write(s, strlen(s));  cout << endl;  cout.write(s, 2);  cout << endl;  } |

|  |  |  |
| --- | --- | --- |
| Класс | Метод | Описание |
| cout | width() | Метод**width()** задает ширину выводимого, если необходимо выровнять до определенного количества символов |
| precision() | Метод**precision()** указывает сколько цифр будет в дробной части, если выводится вещественная переменная. |
| setf() | Метод**setf()** определяет, как будет выравниваться (влево, вправо, по центру) выводимое, и в каком формате оно будет. |

*Ввод и вывод символов (Аналогично и для C++).*

В точности то же самое выведет команда printf("%x", N);  
Чтобы не писать перед кажой переменной ее формат, можно использовать функцию:  
  
Она установит формат вывода в стандартный поток вывода на экран. Этот подход настолько же мощный, как и использование форматной печати с помощью printf.

std::cout << std::ios::hex << N;

std::cout.setf(std::ios::hex);

cout << **hex** << n << endl; //вывод в шестнадцатеричной системе счисления  
  
cout << **dec** << n << endl; //вывод в десятичной системе счисления  
  
cout << **oct** << n << endl; //вывод в восьмеричной системе счисления

*Строковое форматирование* — это операция, позволяющая получить результирующую строку из строки-шаблона и набора аргументов. Строка-шаблон содержит текст, в который включены местозаполнители (placeholders), вместо которых подставляются аргументы.

Строковое форматирование в C осуществляется с помощью семейства функций Xprintf. С тем же успехом, мы можем воспользоваться этими функциями и в C++:

неплохой способ форматирования:

* это самый быстрый способ строкового форматирования
* этот способ работает практически на всех версиях компиляторов, не требуя поддержки новых стандартов

недостатки:

* нужно знать заранее сколько памяти потребуется для результирующей строки, что не всегда возможно определить
* соответствие количества и типа аргументов и местозаполнителей не проверяется при передаче параметров извне (как в обертке над vsnprintf, реализованной ниже), что может привести к ошибкам при выполнении программы

# **37. Файлы в C++. Классы ios\_base и ios. Потоки вывода. Функции open, close и is\_open. Функция flush. Функция put. Функция seekp и tellp. Функция write. Функция str. Классы потокового вывода.**

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

Для работы с файлами необходимо подключить заголовочный файл <fstream> (файл fstream уже включает iostream, так что включать оба файла необязательно). В <fstream> определены несколько классов и подключены заголовочные файлы <ifstream> — файловый ввод и <ofstream>  — файловый вывод.

Файловый ввод/вывод аналогичен стандартному вводу/выводу, единственное отличие – это то, что ввод/вывод выполнятся не на экран, а в файл.

*cout* представляет собой поток, ассоциированный со стандартным выводом. Потоки *cerr* и *clog* используются для вывода сообщений об ошибках. Разница между *cerr* и *clog* заключается в том, что, хотя они оба привяза­ны к стандартному выводу, *cerr* не буферизирован, поэтому все посланные в него данные выво­дятся немедленно. В противоположность этому, *clog* буферизирован, так что данные выводятся только тогда, когда буфер оказывается полным.

|  |  |
| --- | --- |
| ios\_base | *«InputOutputStream\_Base»*, базовый класс всей иерархии классов потоков. Содержит общие функции, типы и классы, в основном представляющие собой флаги (индикаторы). Эти флаги используются функциями подклассов fstream и могут быть определены с помощью функций ios\_base. |
| ios | *«InputOutputStream»*, основной подкласс, вместе с ios\_base, определяющий все остальные подклассы библиотеки потоков. Содержит функции-флаги форматирования и обработки ошибок, а также некоторые функции, унаследованные от ios\_base. |

В системе ввода/вывода С++ каждый поток связан с набором флагов форматирования, управляющих процессом форматирования информации.

*Примеры:*

|  |  |
| --- | --- |
| skipws | Если флаг skipws установлен, то при потоковом вводе данных ведущие «пробельные» символы, или символы пропуска (пробел, табуляция, новая строка), отбрасываются. |
| left  right | Если флаг left установлен, то выводимые данные устанавливаются по левому краю,  если установлен right – по правому. |
| internal | Числовое значение дополняется пробелами, которыми заполняется поле между ним и знаком числа или символом основания системы счисления. (если ни один из флагов left, right или internal не установлен, то по умолчанию по правому краю) |
| showbase | Приводит к отображению обозначения основания системы счисления, в которой представляются числовые значения. Например, если используется 16-ричное представление, то значение 1F будет отображено как 0x1F |
| showpoint | Приводит к отображению десятичной точки и хвостовых нулей для всех чисел с плавающей точкой – нужны они или нет. |

Эти значения используются для установки или очистки флагов форматирования.

Для установки любого флага используется функция setf(), которая является членом класса ios. В самом общем виде её формат выглядит так:

fmtflags setf (fmtflags flags);

Эта функция возвращает значение предыдущих установок флагов форматирования и устанавливает их в соответствии со значением, заданным параметром flags. Например, чтобы установить флаг showbase, можно использовать такую инструкцию:

stream.setf (ios::showbase);

* open(). Этим методом можно открыть заданный файл, сопоставив его с одним из объектов потока..
* is\_open(). Функция, определяющая, открыт ли в данный момент файл, которому сопоставлен определенный объект потока.
* close(). Функция закрывает файл, то есть прекращает доступ к нему, таким образом освобождая его для других функций или программ.

ostream & flush (); - сброс буфера выходного потока

Синхронизирует связанный [поток](http://www.cplusplus.com/streambuf) с его управляемой выходной последовательностью.   
Для объектов [буфера потока,](http://www.cplusplus.com/streambuf) которые реализуют промежуточные буферы, эта функция требует, чтобы все символы записывались в контролируемую последовательность.

Функция put() выводит одиночный символ в стандартное устройство вывода .

с помощью функций seekg() и seekp() выполняется произвольный доступ, т.е. система ввода-вывода C++ управляет двумя указателями, связанными с файлом. Один из них — это указатель ввода(get pointer), который указывает, в каком месте файла будет выполняться следующая операция ввода. А другой — указатель вывода(put pointer), который указывает, в каком месте файла будет выполняться следующая операция вывода.

Версия функции seekp() с двумя параметрами перемещает указатель вывода на offset байт от позиции, заданной параметром origin.

Параметр origin имеет тип ios::seek\_dir и представляет собой перечисление, которое имеет следующие значения.

|  |  |
| --- | --- |
| ios::beg | Смещение от начала |
| ios::cur | Смещение от текущей позиции |
| ios::end | Смещение от конца |

[tellp](http://ic.asf.ru/~/docs/cpp/cppf_iocppold_tellp.htm) — Определение текужего указателя вывода.

ostream & write (const char \* s, streamsize n); - Запись блока данных

Классы потокового ввода:

|  |  |
| --- | --- |
| istream | Универсальный класс ввода, или родительский класс для других производных потоковых классов |
| ifstream | Ввод из файлов. |
| istream\_withassign | Ввод из потока cin. |
| istrstream | Ввод из строки. |

# **38. Файлы в C++. Потоки ввода. Функция gcount. Функция get. Функция getline. Функция ignore. Функция peek. Функция putback. Функция read. Функция seekg. Функция sync. Функция unget. Функция str. Классы потокового ввода.**

Потоки ввода (istream)

Для ввода используется оператор >>. Он также определен для всех встроенных типов и некоторых классов стандартной библиотеки.

istream**&** **operator>>**(istream**&** s)

{

...

}

Функция gount

Синтаксис:

streamsize istream::gcount();

Функция **gcount()** используется с потоками ввода и возвращает количество символов, прочтенных при последней операции неформатированного ввода.

Функция get

Синтаксис:

* int istream::get();
* istream& istream::get( char& ch );
* istream& istream::get( char\* buffer, streamsize num );

Функция **get()** используется с потоками ввода, и производит следующие действия:

* считывает символ и возвращает его значение,
* считывает символ и располагает его в ch,
* считывает символы в буфер до тех пор, пока не считает num - 1 символов, либо не достигнет конца файла, либо не достигнет конца строки, либо не достигнет символа delim
* считывает символы в буфер, пока не достигнет конца строки, или конца файла, или символа delim .

Функция getline

Синтаксис:

istream& istream::getline( char\* buffer, streamsize num );

Функция **getline()** используется с потоками ввода. Функция считывает символы в буфер, пока не произойдет хотя бы одно из событий:

* считано num - 1 символов,
* достигнут конец файла EOF,
* достигнут символ delim. Этот символ не будет считан.

Функция ignore

Синтаксис:

istream& istream::ignore( streamsize num=1, int delim=[EOF](http://ru.cppreference.com/w/cpp/io/c) );

Функция **ignore()** используется с потоками ввода.

Функция peek

Синтаксис:

int istream::peek();

Функция **peek()** используется с потоками ввода и возвращает следующий **символ потока** или **EOF**, если достигнут конец файла. **peek()** не удаляет символ из потока.

Функция read

Синтаксис:

istream& istream::read( char\* buffer, streamsize num );

Функция **read()** используется с потоками ввода. Она считывает **num** байтов из потока до расположения их в буфер.

Функция seekg

Перемещает указатель ввода. Функция является членом класса **istream.**

В системе ввода-вывода **C++** с помощью функций **seekg()** и **seekp()** выполняется произвольный доступ, т.е. система ввода-вывода **C++** управляет двумя указателями, связанными с файлом. Один из них — это указатель **ввода**(**get pointer**), который указывает, в каком месте файла будет выполняться следующая операция ввода. А другой — указатель **вывода**(**put pointer**), который указывает, в каком месте файла будет выполняться следующая операция вывода.

istream &seekg(pos\_type position);

Версия функции **seekg()** с одним параметром перемещают указатели файлов в позиции, заданные параметром position.

Функция sync

Она просто очищает поток от имеющихся в нем символов. То есть, очищает буфер.

Функции putback и unget

istream& istream::unget ()

istream& istream::putback (**char** c)

Обе функции возвращают в поток данных последний считанный символ, чтобы он был считан следующей операцией чтения.

Различия между функциями **unget** и **pushback**() заключаются в том, что **putback()** проверяет, был ли передаваемый символ с последним считанным символом.

Классы потокового ввода

|  |  |
| --- | --- |
| Классы потокового ввода | |
| Istream | Универсальный класс ввода, или родительский класс для других производных потоковых классов ввода. |
| Ifstream | Ввод из файлов. |
| istream\_withassign | Ввод из потока *cin.* |
| Istrstream | Ввод из строки. |

# **39. Файлы в C++. Потоки ввода-вывода. Классы буферизированных потоков. Класс строковых потоков. Форматирование потока. Резидентные в памяти потоки. Буферы и синхронизация.**

Потоки ввода-вывода

|  |  |
| --- | --- |
| Классы потокового ввода/вывода | |
| Iostream | Универсальный класс ввода/вывода, или родительский класс для других производных потоковых классов ввода/вывода. |
| Fstream | Ввод/вывод в файлы. |
| stdiostream | Стандартный поток ввода/вывода. |
| Strstream | Ввод/вывод в строку. |

Форматирование потока

Непосредственное применение операций ввода **<<** и вывода **>>** к стандартным потокам **cout**, **cin**, **cerr, clog** для данных базовых типов приводит к использованию “**умалчиваемых”** форматов внешнего представления пересылаемых значений.

Форматы представления выводимой информации и правила восприятия данных при вводе могут быть изменены программистом с помощью флагов форматирования.

Кроме флагов форматирования используются следующие **protected** компонентные данные класса **ios**:

**int x\_width** – минимальная ширина поля вывода.

**int x\_precision** – точность представления вещественных чисел (количество цифр дробной части) при выводе;

**int x\_fill** – символ-заполнитель при выводе, пробел – по умолчанию.

Для получения (установки) значений этих полей используются следующие компонентные функции:

int width();

int width(int);

int precision();

int precision(int);

char fill();

char fill(char);

Классы строковых потоков

Строковые потоки позволяют считывать и записывать информацию из областей оперативной памяти так же, как и из файла, с консоли или на дисплей. три класса строковых потоков:

**istringstream** – входные строковые потоки;

**ostringstream** – выходные строковые потоки;

**stringstream** – двунаправленные строковые потоки.

Классы буферизированных потоков

|  |  |
| --- | --- |
| Классы буферизованных потоков | |
| Streambuf | Родительский класс для производных буферизованных классов. |
| Filebuf | Буферизованный поток для файлов. |
| Stdiobuf | Буферизованный поток для стандартного файлового ввода/вывода. |
| Strstreambuf | Буферизованный поток для строк. |

Резидентные в памяти потоки

**Резидентные потоки применяются для выполнения форматных операций ввода/вывода для данных, находящихся в памяти. В файле strstream.h определены три класса резидентных потоков: istrstream, ostrstream и strstream. Создавая резидентный поток, нужно указать буфер и его размер. Если буфер ввода**

Буферы и синхронизация

Каждый буфер содержит в зависимости от режима доступа один или два символьных массива: один для ввода, а второй для вывода данных. Каждый поток поддерживает внутренний указатель на буфер. Ввод-вывод данных из буфера в файл или в строку называется ***синхронизацией буфера с внешним устройством ввода-вывода***. Синхронизация буфера с внешним устройством может выполняться неявно или явно.

# **40. Файлы в C++. Условные признаки файлов в C++. Опрос и установка состояния потока. Ошибки потоков. Часто применяемые функции. Двоичные файлы. Объединение программ на С и С++. Использование спецификатора extern "С".**

**Состояния потока**

Пользователей библиотеки iostream, разумеется, интересует, находится ли поток в ошибочном состоянии. Например, если мы пишем

**int ival;**

**cin ival;**

и вводим слово "*Borges*", то cin переводится в состояние ошибки после неудачной попытки присвоить строковый литерал целому числу. Если бы мы ввели число 1024, то чтение прошло бы успешно и поток остался бы в нормальном состоянии.

**if ( !cin )**

**// операция чтения не прошла или встретился конец файла**

**while ( cin word )**

**// операция чтения завершилась успешно ...**

Условие в цикле while будет равно false, если достигнут конец файла или произошла ошибка при чтении. В большинстве случаев такой проверки потокового объекта достаточно.

**if ( inOut.eof() )**

**// отлично: все прочитано ...**

\* bad() возвращает true при попытке выполнения некорректной операции, например при установке позиции за концом файла.

\* fail() возвращает true, если операция завершилась неудачно, например не удалось открыть файл или передан некорректный формат ввода:

**ifstream iFile( filename, ios\_base::in );**

**if ( iFile.fail() ) // не удалось открыть**

**error\_message( ... );**

\* good() возвращает true, если все вышеперечисленные условия ложны:

**if ( inOut.good() )**

Существует два способа явно изменить состояние потока iostream. С помощью функции-члена clear() ему явно присваивается указанное значение. Функция setstate() не сбрасывает состояние, а устанавливает один из флагов, не меняя значения остальных

**if ((ch = is.get()) != ' )**

**{**

**is.setstate( ios\_base::failbit );**

**return is;**

**}**

Имеются следующие значения флагов состояния:

**ios\_base::badbit**

**ios\_base::eofbit**

**ios\_base::failbit**

**ios\_base::goodbit**

Для установки сразу нескольких флагов используется побитовый оператор ИЛИ:

**is.setstate( ios\_base::badbit | ios\_base::failbit );**

**cin.clear(); // сброс ошибок**

В более общем случае clear() используется для сброса текущего состояния и установки одного или нескольких флагов нового. Например:

**cin.clear( ios\_base::goodbit );**

восстанавливает нормальное состояние потока.

Функция-член rdstate() позволяет получить текущее состояние объекта:

**ios\_base::iostate old\_state = cin.rdstate();**

**cin.clear();**

**process\_input();**

**// перевести поток cin в прежнее состояние**

**cin.clear( old\_state );**

**Модель двоичного файла**

Двоичный файл отличается от текстового тем, что данные в нем представлены во внутренней форме. А поскольку при внутреннем представлении используется двоичная система счисления, то «в честь ее» файлы и называются двоичными

.

* физическая память имеет байтную структуру – единицей адресации является байт;
* любая переменная занимает фиксированное количество байтов, определяемое ее типом. Операция sizeof возвращает эту размерность;
* указатель  на переменную интерпретируется как ее адрес в памяти.

Исходя из этих принципов, функции двоичного ввода-вывода fread и fwrite переносят содержимое памяти в двоичный файл «прозрачно», т.е. байт в байт без каких либо преобразований.

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

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

Особенностью этих функций является то, что для них безразличен характер структуры данных в той области памяти, в которую осуществляется ввод-вывод .Функци fread читает, а функция fwrite пишет в файл, начиная с текущей позиции, массив из nrec элементов размерностью size байтов каждый, возвращая количество успешно прочитанных (записанных) элементов.

В Си для представления адресов используются переменные типа long, на которые можно распространить известное понятие указатель - указатель в файле.

**extern**

Если при объявлении выделяется память под переменную, то процесс называется определением. Использование extern приводит к объявлению, но не к определению. Оно просто говорит компилятору, что определение происходит где-то в другом месте программы.

Имеется другой вариант использования extern. Когда используется глобальная переменная внутри функции, находящейся в том же файле, где происходит объявление глобальной переменной, то можно объявлять ее как extern, хотя делать это не обязательно.

# **41. Файлы в C++. Манипуляторы. Манипуляторы без параметров. Манипуляторы с одним параметром. Манипуляторы с несколькими параметрами.**

**Манипуляторами** называют специальные функции, позволяющие программисту изменять состояния и флаги потока. Особенность манипуляторов и их отличие от обычных функций состоит в том, что **их имена и вызовы можно использовать в качестве правого операнда для операции обмена** (**<<** или **>>**). В качестве левого операнда в этом выражении, как обычно, используется поток, и именно на этот поток оказывает влияние манипулятор.

   манипулятор **hex** позволяет устанавливать шестнадцатеричное основание счисления выводимых в поток **cout** числовых значений:

cout << "\nДесятичное число: " << 15 << hex;

cout << "\nШестнадцатеричное представление: << 15;

получим на экране:

Десятичное число: 15

Шестнадцатеричное представление: 0xF

    В качестве параметра каждый манипулятор получает ссылку на тот поток, с которым он используется в выражении. После выполнения манипулятора он возвращает ссылку на тот же поток.

    Манипуляторы библиотеки классов ввода-вывода языка **С++** делятся на две группы: **манипуляторы с параметрами**и **манипуляторы без параметров**.

    Манипуляторы без параметров:

* **dec** - при вводе и выводе устанавливает флаг десятичной системы счисления;
* **hex** - при вводе и выводе устанавливает флаг шестнадцатеричной системы счисления;
* **oct** - при вводе и выводе устанавливает флаг восьмеричной системы счисления;
* **ws**
* **endl** - действует только при выводе, обеспечивает включение в выходной поток символа новой строки и сбрасывает буфер (выгружает содержимое) этого потока;
* **ends** ;
* **flush**

   Манипулятор **ws** действует только при вводе; **endl, ends, flush** - только при выводе.

    Манипуляторы **dec, hex, oct**, задающие основание системы счисления, изменяют состояние потока, и это изменение остается в силе до следующего явного изменения.

    Манипулятор **endl** рекомендуется использовать при каждом выводе, который должен быть незамедлительно воспринят пользователем.

cout << "Ждите! Идет набор статистики." << endl;

При отсутствии **endl** здесь нельзя гарантировать, что сообщение пользователю не останется в буфере потока **cout** до окончания набора статистики.

    Манипуляторы с параметрами определены в файле **iomanip.h**. Перечислим их:

**setbase(int n)**

устанавливает основание (**n**) системы счисления.

**resetiosflags(long L)**

сбрасывает отдельные флаги состояния потоков ввода и вывода на основе битового представления значения параметра ;

**setiosflags(long L)**

устанавливает отдельные флаги состояния потоков ввода-вывода на основе битового представления значения параметра **L** ;

**setfill(int n)**

значение параметра **n** в дальнейшем используется в качестве кода символа-заполнителя, который помещается в незанятых позициях поля при вводе значения ;

**setprecision(int n)**

определяет с помощью значения параметра **n** точность представления вещественных чисел;

**setw(int n)**

значение параметра **n** задает минимальную ширину поля вывода .

   для управления форматом вывода класс **ios** имеет следующие компоненты:

* компонент  [**x\_flags**](http://it.kgsu.ru/C_STREAM/oop_0006.html#1);
* функции доступа к атрибуту **x\_flags**: [**flags()** и **setf()**](http://it.kgsu.ru/C_STREAM/oop_0006.html#1);
* атрибуты управления форматом: [**x\_width, x\_precision, x\_fill**](http://it.kgsu.ru/C_STREAM/oop_0006.html#1);
* принадлежащие классу функции: [**width(), precision(), fill()**](http://it.kgsu.ru/C_STREAM/oop_0006.html#1);
* манипуляторы .

# 42. Функции для работы с файлами. Полезные функции для работы с файлами. clearerr(). fclose(). fcloseall(). fdopen(). feof(). ferror(). fflush(). fgetc(). fgetchar(). fgetpos().

**void clearerr(FILE \*file)** - устанавливается в нуль состояние признака ошибки в указанном файле.

**fclose()** используется для закрытия потока, ранее открытого с помощью fopen(). Она сохраняет в файл данные, находящиеся в дисковом буфере, и выполняет операцию системного уровня по закрытию файла.

Функция fclose() имеет прототип:

**int fclose(FILE \*fp);**

где fp - это указатель на файл, возвращенный fopen(). Если возвращен 0, то это означает, что операция закрытия выполнена успешно, а если EOF, то, значит, была ошибка.

**int fcloseall(void)**

закрывает все открытые потоки ввода-вывода. за исключением stdin, stdout, stderr, stdaux, stdprn. Возвращает общее число потоков, закрытых функцией, либо EOF.

**fdopen(int handle, char \* type);**

связывает поток с дескриптором, полученным функциями creat, dup, dup2 или open. Тип потока должен совпадать с режимом, в котором был открыт handle.

**int feof(FILE \* указатель\_на \_файл);**

возвращает отличное от нуля значение (true) если достигнут конец файла, и ноль (false), если конец файла не достигнут. Функция может работать с любыми типами файлов.

**int ferror(FILE \* stream);**

является макрокомандой, которая проверяет данный поток stream на ошибочную операцию записи или чтения. Если установлен признак ошибки потока stream, он сохраняет его до вызова функций clearerr или rewind или до момента закрытия потока.

**int fflush(FILE \* stream);**

записывает в файл содержимое буфера, связанного с потоком stream, если он был открыт на вывод. Функция fflush не оказывает влияния на не буферизованные потоки.

**int fgetc(FILE \*stream);**

Возвращает символ из открытого файла, описываемого переменной типа FILE , на которую указывает stream .

**fgetchar()**

Считывает символ из потока stdin.

**int fgetpos(FILE \* stream, fpos\_t \*pos);**

сохраняет позицию указателя файла, связанного с потоком stream, в месте, указываемом pos.

# 43. Функции для работы с файлами. fgets(). filelength(). fileno(). flushall() fflush(). fopen(). fprintf(). fputc(). fputchar(). fputs(). fread().

**fgets() -** Возвращает строку из потока.

**long filelength(int handle);**

возвращает длину в байтах файла, соответствующего дескриптору handle.

При успешном завершении функция filelength возвращает значение типа long- длину файла в байтах. При ошибке функция возвращает значение -1L, и глобальной переменной errno присваивается: EBADF - Неверный номер файла.

**int fileno(FILE \* stream);**

представляет собой макрокоманду, которая возвращает логический номер файла для заданного потока stream. Если поток stream имеет более одного номера, функция fileno возвращает номер, назначенный данному потоку при первом открытии.

Функция fileno возвращает целое число, обозначающее значение дескриптор файла, соответствующий потоку stream.

**int flushall(void);**

сбрасывает все буфера, связанные с открытыми входными потоками, и записывает в соответствующие файлы все буфера, относящиеся к открытым выходным потокам. Любые операции чтения, следующие за функцией flushall, будут читать новые данные для входных файлов в буферы. Все потоки остаются открытыми.

Функция flushall возвращает число открытых входных и выходных потоков.

**int fflush(FILE \* stream);**

записывает в файл содержимое буфера, связанного с потоком stream, если он был открыт на вывод. Функция fflush не оказывает влияния на не буферизованные потоки.

В случае успешного завершения возвращает 0. При ошибке возвращает EOF.

**FILE\* fopen(const char\*filename,const char\*mode);**

где const char\*filename – строка, которая содержит имя файла, связанного с потоком,

const char\*mode – строка режимов открытия файла.

**fprintf()** - Выполняет форматированную запись в поток

**int fputc(int ch, FILE \* stream);**

записывает символ ch в заданный поток stream в текущую позицию файла, а затем передвигает индикатор позиции файла. Возвращает значение записанного символа, а в случае ошибки значение EOF.

**fputchar()** - Записывает символ в stdout

**fputs()** - Записывает строку в поток

**fread()** - Считывает неформатированные данные из потока

# 44. Функции для работы с файлами. freopen(). fscanf(). fseek(). fsetpos(). fstat(). ftell(). fwrite(). getc(). getchar(). gets(). getw().

**FILE\* freopen(const char\*filename, const char\*mode, FILE\*stream)**

закрывает поток, на описание которого указывает stream, а затем открывает файл, на спецификацию которого указывает ASCIIZ-строка filename. Режим открытия файла задает ASCIIZ- строка mode. В случае успеха функция возвращает указатель на описание открытого потока, которое будет совпадать с stream. В противном случае функция возвращает NULL.

**int fscanf(FILE \* stream, char \* format[, adress,...]);**

сканирует посимвольно набор вводимых полей, считывая их из потока. Затем каждое поле из потока форматируется в соответствии со спецификацией формата, которая передается fscanf в виде указателя на строку format. Полученное в результате этого поле fscanf запоминает в аргументах, передаваемых функции fscanf после параметра format. Количество аргументов должно совпадать с количеством спецификаций формата.

**int fseek(FILE \*fp, long число\_байт, int начало);**

Можно выполнять операции произвольного чтения и записи, используя систему буферизированного ввода/вывода, с помощью fseek(), устанавливающей текущую файловую позицию.

**int fsetpos(FILE\* stream, const fpos\_t\* pos);**

Устанавливает индикатор позиции файла в позицию, на которую указывает pos. Индикатор конца файла сбрасывается. Код возврата: удача – 0, неудача – не 0.

**fstat()** - Возвращает информацию об открытом файле

**long int ftell(FILE \*stream);**

возвращает положение указателя текущей позиции файла, связанного с потоком stream. Значение выдается в виде смещения в байтах относительно начала файла.

Значение, возвращаемое ftell, можно в дальнейшем использовать при вызове функции fseek.

Функция ftell возвращает положение указателя текущей позиции при успешном завершении. При ошибке возвращает-1L, и присваивает переменной errno положительное значение.

**fwrite()** - Записывает неформатированные данные в поток

**getc()** - Этот макрос считывает символ из потока

**getchar()** - Этот макрос считывает символ из stdin

**gets()** - Возвращает строку из stdin

**getw()** - Считывает целое число из потока

# 45. Функции для работы с файлами. perror(). printf(). putc(). putchar(). puts(). putw(). remove(). rename(). rewind(). scanf(). setbuf().

**perror()** - Выдает системную ошибку на stderr.

**printf()** - Записывает форматированные данные в stdout.

**putc()** - Этот макрос записывает символ в поток.

**putchar()** - Этот макрос записывает символ в stdout.

**puts()** - Записывает строку в stdout.

**putw()** - Записывает целое число в поток.

**remove()** - Удаляет файл.

**rename()** - Переименует файл.

**rewind()** - Позиционирует указатель файла на начало потока.

**scanf()** - Сканирует и вводит форматированные данные из stdln.

**setbuf()** - Отменяет автоматическую буферизацию, позволяя приложению определить собственный буфер для потока

# 46. Функции для работы с файлами. setvbuf(). sprintf(). sscanf(). tmpfile(). tmpnam(). ungetc(). vfprintf(). vfscanf(). vprintf(). vsprintf(). vsscanf().

**setvbuf()** - То же самое, что и setbuf(); позволяет также определить размер буфера

**sprintf()** - Записывает форматированные данные в строку

**sscanf()** - Сканирует и вводит форматированные данные из строки

**tmpnam()** - Создает в заданном подкаталоге файл с уникальным именем

**ungetch()** - Засылает символ в буфер клавиатуры

**vfprintf()** - Записывает форматированные данные в поток, используя указатель на форматирующую строку

**vfscanf()** - Сканирует и форматирует ввод из потока, используя указатель на форматирующую строку

**vprintf()** - Записывает форматированные данные в stdout, используя указатель на форматирующую строку

**vscanf()** - Сканирует и форматирует ввод из stdin, используя указатель на форматирующую строку

**vsprintf()** - Записывает форматированные данные в строку, используя указатель на форматирующую строку

**vsscanf()** - Сканирует и форматирует ввод строки, используя указатель на форматирующую строку

# 47. Функции для работы с файлами. Низкоуровневый ввод и вывод. close(). lseek(). open(). read(). unlink() (\_unlink()). write().

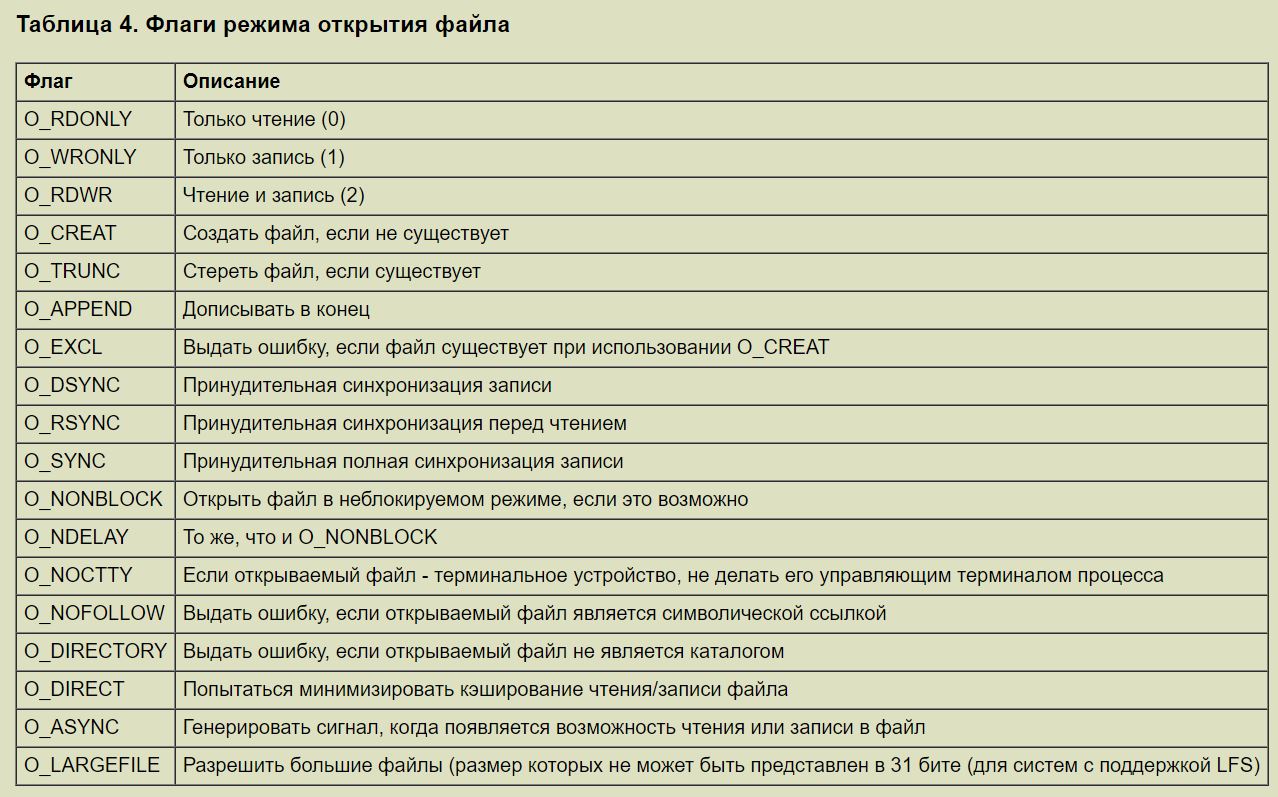
Чтобы получить возможность прочитать что-то из файла или записать что-то в файл, его нужно открыть. Это делает системный вызов open(). Этот системный вызов не имеет постоянного списка аргументов. Существуют две "разновидности" open().

Адаптированный прототип open().

**int open (const char \* filename, int flags, ...);**

Начнем по порядку. Первый аргумент - имя файла в файловой системе в обычной форме: полный путь к файлу (если файл не находится в текущем каталоге) или сокращенное имя (если файл в текущем каталоге).

Второй аргумент - это режим открытия файла, представляющий собой один или несколько флагов открытия, объединенных оператором побитового ИЛИ. Список доступных флагов приведен в таблице 1.



Третий аргумент используется в том случае, если open() создает новый файл. В этом случае файлу нужно задать права доступа (режим), с которыми он появится в файловой системе. Права доступа задаются перечислением флагов, объединенных побитовым ИЛИ. Вместо флагов можно использовать число (как правило восьмиричное), однако первый способ нагляднее и предпочтительнее. Список флагов приведен в [Таблице](https://www.opennet.ru/docs/RUS/zlp/appendix02.html#1) 2.

Если файл был успешно открыт, open() возвращает файловый дескриптор(это неотрицательное целое число. Когда создается новый поток ввода-вывода, ядро возвращает процессу, создавшему поток ввода-вывода, его файловый дескриптор), по которому мы будем обращаться к файлу.

Чтобы открыть существующий файл на чтение, можно написать

int fd;

fd = open(name, 0\_RDONLY, 0);



Системный вызов close() закрывает файл. Если файлы не закрывать самостоятельно, то соответствующие дескрипторы не освобождаются, что может привести к превышению лимита открытых файлов.

Адаптированный прототип.

**int close (int fd);**

Очевидно, что единственный аргумент - это файловый дескриптор. Возвращаемое значение - ноль в случае успеха, и -1 - в случае ошибки.

Системный вызов read(), позволяет читать данные из файла. Read() читает данные в "сыром" виде, то есть как последовательность байт, без какой-либо интерпретации.

Адаптированный прототип read().

**ssize\_t read (int fd, void \* buffer, size\_t count);**

Первый аргумент - это файловый дескриптор. Второй аргумент - это указатель на область памяти, куда будут помещаться данные. Третий аргумент - количество байт, которые функция read() будет **пытаться** прочитать из файла. Возвращаемое значение - количество прочитанных байт, если чтение состоялось и -1, если произошла ошибка.

Пример:

**/\* myread1.c \*/**

**#include <stdlib.h>**

**#include <stdio.h>**

**#include <unistd.h>**

**#include <fcntl.h>**

**#include <sys/stat.h>**

**#include <sys/types.h>**

**#define BUFFER\_SIZE 64**

**int main (int argc, char \*\* argv)**

**{**

**int fd;**

**ssize\_t read\_bytes;**

**char buffer[BUFFER\_SIZE+1];**

**if (argc < 2)**

**{**

**fprintf (stderr, "Too few arguments\n");**

**exit (1);**

**}**

**fd = open (argv[1], O\_RDONLY);**

**if (fd < 0)**

**{**

**fprintf (stderr, "Cannot open file\n");**

**exit (1);**

**}**

**while ((read\_bytes = read (fd, buffer, BUFFER\_SIZE)) > 0)**

**{**

**buffer[read\_bytes] = 0; /\* Null-terminator for C-string \*/**

**fputs (buffer, stdout);**

**}**

**if (read\_bytes < 0)**

**{**

**fprintf (stderr, "myread: Cannot read file\n");**

**exit (1);**

**}**

**close (fd);**

**exit (0);**

**}**

Для записи данных в файл используется системный вызов write(). Ниже представлен его прототип.

**ssize\_t write (int fd, const void \* buffer, size\_t count);**

Эффективнее обмениваться большим числом байтов, поскольку при этом требуется меньше системных вызовов.

#include "syscalls.h"

main() /\* копирование ввода на вывод \*/

{

char buf[BUFSIZ];

int n;

while ((n = read(0, buf, BUFSIZ)) › 0)

write(i, buf, n);

return 0;

}

Параметр BUFSIZ также определен в ‹syscalls.h›: в каждой конкретной системе он имеет свое значение. Если размер файла не кратен BUFSIZ, то какая-то операция чтения вернет значение меньшее, чем BUFSIZ, а следующее обращение к *read* даст в качестве результата нуль.

Полезно рассмотреть, как используются *read* и *write* при написании программ более высокого уровня - таких как *getchar*, *putchar* и т. д. Вот, к примеру, версия программы *getchar*, которая осуществляет небуферизованный ввод, читая по одному символу из стандартного входного потока.

#include "syscalls.h"

/\* getchar: небуферизованный ввод одного символа \*/

int getchar(void) {

char с;

return (read(0, &c, 1) == 1) ? (unsigned char) с: EOF;

}

Переменная *c* должна быть типа *char*, поскольку *read* требует указателя на *char*. Приведение *c* к *unsigned char* перед тем, как вернуть ее в качестве результата, исключает какие-либо проблемы, связанные с распространением знака.

Для изменения текущей позиции чтения-записи используется системный вызов lseek(). Ниже представлен его прототип.

**off\_t lseek (int fd, ott\_t offset, int against);**

Первый аргумент, как всегда, - файловый дескриптор. Второй аргумент - смещение, как положительное (вперед), так и отрицательное (назад). Третий аргумент обычно передается в виде одной из трех констант SEEK\_SET, SEEK\_CUR и SEEK\_END, которые показывают, от какого места отсчитывается смещение. SEEK\_SET - означает начало файла, SEEK\_CUR - текущая позиция, SEEK\_END - конец файла. Рассмотрим следующие вызовы:

**lseek (fd, 0, SEEK\_SET);**

**lseek (fd, 20, SEEK\_CUR);**

**lseek (fd, -10, SEEK\_END);**

В случае удачного завершения, lseek() возвращает значение установленной "новой" позиции относительно начала файла. В случае ошибки возвращается -1.

Следует обратить внимание на аргумент 0L: вместо 0L можно было бы написать (*long*)0 или, если функция *lseek* должным образом объявлена, просто 0. Cледующая функция читает любое число байтов из любого места файла. Она возвращает число прочитанных байтов или -1 в случае ошибки.

#include "syscalls.h"

/\* get: читает n байт из позиции pos \*/

int get(int fd, long pos, char \*buf, int n)

{

if (lseek(fd, pos, 0) ›= 0) /\* установка позиции \*/

return read(fd, buf, n);

else

return -1;

}

Функция unlink(char \*name) удаляет имя файла из файловой системы. Функция unlink возвращает 0, если файл успешно удален. Возвращаемое значение -1 свидетельствует об ошибке.

#include <io.h>

#include <stdio.h>

int result;

result=unlink("tmpfile");

if (result == -1)

perror("couldn't delete tmpfile");

## 48. Структуры С и C++: синтаксис и правила. Структуры C++: расширения синтаксиса и правил. Доступ к элементам структуры. Инициализация структуры. Передача структур в функции.

Структура – сложный тип данных, позволяющий одновременно хранить простые пользовательские и сложные типы данных. В отличие от массива, который является однородным объектом, структура может быть неоднородной. Объявление структуры всегда должно начинаться с ключевого слова struct. Необязательно, чтобы структура имела имя, но тогда такая структура обязательно должна иметь структурные переменные. Обязательно в объявлении структуры должны присутствовать фигурные скобочки, они обрамляют тело структуры, в котором объявляются её атрибуты (элементы). Структурные переменные, при объявлении структуры, указывать необязательно.

struct Name {тип\_элемента элемент1; тип\_элемента элемент2; … } structVariable;

! Компилятор для шаблона память не выделяет, только для переменных, которые соответствуют шаблону.

struct book {

char title[MAXTIT];

char author[MAXAUTH];

float value;

} libty;

Для элементов структуры выделяется память, равная сумме типов (int 4б, char 1б, float 4б).

Для структуры допустима инициализация

struct MyStruct {

inti Value;

long iValue;

char str[10];

} ;

MyStruct one = {10, 300L, “Hello”};

Таким образом, структуры имеют следующие свойства:

поля структуры могут иметь разный тип;

полями структуры могут быть функции (С++);

структура может быть аргументом и результатом функции:

struct man \*create()

{ ...

};

void f(struct man \*q)

{ ... }

Доступ к элементам структуры

1 способ) используя опции (.) т.е. (strname.elname)

struct Organization

{

char name[64];

int age;

float salary;

} worker;

cout << worker.name << endl;

2 способ) используя указатель (-> )

3 способ) оператор расширенного доступа (::)

Передача структур в функции

При передаче функции члена структуры передается его значение, притом не играет роли то, что значение берется из члена структуры.

struct fred

{

char x;

int y;

char s[10];

} mike;

func(mike.x); /\* передается символьное значение x \*/

func2(mike.y); /\* передается целое значение y \*/

func(mike.s[2]); /\* передается символьное значение s[2] \*/

В каждом из этих случаев функции передается значение определенного элемента, и здесь не имеет значения то, что этот элемент является частью какой-либо большей совокупности.

При использовании структуры в качестве аргумента надо помнить, что тип аргумента должен соответствовать типу параметра. Например, в следующей программе и аргумент arg, и параметр parm объявляются с одним и тем же типом структуры.

#include <stdio.h>

/\* Определение типа структуры. \*/

struct struct\_type {

int a, b;

char ch;

} ;

void f1(struct struct\_type parm);

int main(void)

{

struct struct\_type arg;

arg.a = 1000;

f1(arg);

return 0;

}

void f1(struct struct\_type parm)

{

printf("%d", parm.a);

}

## 49. Массив структур. Описание массива структур. Определение элементов массива структур. Вложенные структуры.

С помощью шаблона и переменной можно описать только 1 элемент структуры – запись. Если необходимо хранить много записей, то требуется использовать массивы структур. Возможно, наиболее часто структуры используются в виде массивов структур. Для объявления массива структур следует сначала определить структуру, а затем объявить массив переменных данного типа.   
  
struct MyStruct

{

int A;

char C[10];

};

MyStruct obj[10]; //Десять элементов типа MyStruct

//При этом каждый из этих десяти элементов состоит из двух элементов определенных внутри структуры

int main()

{

obj[0].A=100; //Первый элемент массива переменная int A

strcpy(obj[0].C,"12345"); //Первый элемент массива переменная char C[10]

}  
  
Для доступа к отдельным структурам массива obj следует проиндексировать имя массива. Например, для вывода содержимого поля A третьей структуры, следует написать:  
  
printf("%ld", obj[2].A);

Вложенные структуры

Т.к структура является типом данных, то она может стать элементом структуры. Член структуры может быть как простым, так и составным. Простой член - это обычный базовый тип данных, например целочисленный или символьный. Другие составные типы данных являются одномерными или многомерными массивами различных типов данных и структур.

Когда структура является членом другой структуры, то она называется вложенной структурой. Например, ниже структура addr вложена в emp:

struct emp {

struct addr address;

float wage;

} worker;

Здесь структура emp имеет два члена. Первый - это структура типа addr, содержащая адреса служащих. Другой - это wage, содержащий заработную плату служащих. Следующий фрагмент кода присваивает 35000$ элементу wage структуры worker и 98765 полю zip структуры address:

worker.wage = 35000.00;

worker.address.zip = 98765;

В данном примере обращение к членам каждой структуры происходит снаружи вовнутрь (слева направо).

## 50. Использование указателей на структуры. Описание и инициализация указателя на структуру. Доступ к элементу структуры при помощи указателя.

Разработчик ПО на C++ имеет возможность обращаться к элементам структуры через указатели. Для этого должна быть ооъявлена соответствующая переменная типа указателя на структуру, синтаксис которой может быть представлен в виде:

тип\_структуры\* идентификатор\_указателя;

Доступ к элементам структуры через указатель осуществляется с использованием не точки, а символа стрелки (->). Например, создадим указатель на структуру HOUSE:

HOUSE \*pHouse; pHouse = &MyHouse;

Теперь запишем в структуру информацию о наличии автостоянки около дома посредством объявленного указателя и выведем записанную информацию:

pHouse->Parking = true;

cout « MyHouse.Parking;

//или так: cout « pHouse->Parking;

Для получения адреса структурной переменной следует поместить оператор & перед именем структуры. Пусть имеется следующий фрагмент

struct bal {

float balance;

char name[80];

} person;

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

p = &регson; /\*тогда помещает адрес структуры person в указатель р\*/

Для доступа к членам структуры с помощью указателя на структуру следует использовать оператор "стрелка". Например, для доступа к члену balance с помощью p следует написать:

p -> balance

Кроме указателей на простые типы можно использовать указатели на структуры. А для доступа к полям структуры, на которую указывает указатель, используется операция ->:

class Program

{

static void Main(string[] args)

{

unsafe

{

Person person;

person.age = 29;

person.height = 176;

Person\* p = &person;

p->age = 30;

Console.WriteLine(p->age);

(\*p).height = 180; // разыменовывание указателя

Console.WriteLine((\*p).height);

}

}

}

public struct Person

{

public int age;

public int height;

}

Обращаясь к указателю p->age = 30; мы можем получить или установить значение свойства структуры, на которую указывает указатель. Обратите внимание, что просто написать p.age=30 мы не можем, так как p - это не структура Person, а указатель на структуру. Альтернативой служит операция разыменования.

Помимо использования указателей возможно применение ссылок на структуры. Объявление такой ссылки имеет следующий синтаксис:

тип\_\_структуры &имя\_ссылки =>имя\_переменной ;

Как и ссылка на обычную переменную, ссылка на структуру должна быть инициализирована именем объекта, на который она указывает (в данном случае это имя\_переменной)

Ссылки и указатели на структуры данных могут быть переданы в качестве аргументов в тело функции. При этом значительно снижается время (в сравнении с передачей по значению), за которое данный параметр передается в функцию.

// Пример передачи указателя и ссылки на целочисленную переменную:

bool Func(int\* ptr, ints ref);

// Передача указателя и ссылки на структуру типа HOUSE:

char Func2(HOUSE\* pMh, HOUSE& rMh) ;

Таким образом, в функцию Func2 будут переданы не сами значения структур, а соответствующие адреса, что в значительной степени экономит стековую память.

## 51. Структуры и функции. Передача структур по значению. Передача структур посредством указателей. Передча структур посредством ссылок. Передача элементов структуры. Передача массива структур.

По значению

struct ALLNUMB

{

int nVar;

long lVar;

short shVar;

unsigned int uiVar;

}

прототип функции:

void Punc (ALLNUMB);

посредством указателей

HOUSE \*pHouse;

pHouse = &MyHouse;

pHouse->Parking = true;

cout << MyHouse.Parking;

// или так: cout << pHouse->Parking;

Посредством ссылок

// Пример передачи указателя и ссылки на

// целочисленную переменную:

bool Func(int\* ptr, int& ref);

// Передача указателя и ссылки на

// структуру типа HOUSE:

char Func2 (HOUSE\* pMh, HOUSES rMh);

Элементов структуры

#include<stdio.h>

struct funds {

char \*bank;

float bankfund;

char \*save;

float savefund; } stan = {

"Senior tomatoes Bank",

1023.43,

"Savings and lendings of Snoopy",

4239.87

};

float sum(float x, float y);

void main()

{

float total;

extern struct funds stan; /\* необязательное описание \*/

total=sum(stan.bankfund, stan.savefund);

printf ("Stan has only %.2f dollars.\n", total);

}

/\* складывает два числа типа float \*/

float sum(float x, float y)

{

return x+y;

}

Массива структур

/\* передача массива структур в функцию \*/

#include<stdio.h>

struct funds {

char \*bank;

float bankfund;

char \*save;

float savefund; } jones[2] ={

{

"Senior tomatoes Bank",

1023.43,

"Savings and lendings of Snoopy",

4239.87

},

{

"Honest Jack's Bank",

976.57,

"Accumulation by prior plan",

1760.13

}

};

float sum(struct funds \*);

void main()

{

float total;

total=sum(jones);

printf ("Jonsons have only %.2f dollars.\n" , total);

}

float sum (struct funds \*money)

{

float total;

int i;

for( i = 0, total = 0; i < 2; i++ , money++)

total+= money->bankfund + money->savefund;

return(total);

}

Программа выдает

## 52. Функция-член стуктуры. Особенность возврата const. Возврат структуры из функции.

Функция - член

struct **\_3d {**

double **x, y, z;**

} vect;

struct **\_3d {**

double **x, y, z;**

double **mod ();**

} vect;

Особенности возврата

int\* - pointer на int

int const \* - pointer на const int

int \* const - const pointer на int

int const \* const - const pointer на const int

Теперь первый const может быть по обе стороны от типа:

const int \* == int const \*

const int \* const == int const \* const

Если вы хотите сойти с ума, вы можете делать такое:

int \*\* - pointer на pointer на int

int \*\* const - const pointer на a pointer на int

int \* const \* - pointer на const pointer на int

int const \*\* - pointer на pointer на const int

int \* const \* const - const pointer на const pointer на int

...

constint\* foo;

int **\***const **bar;**

**foo** - переменный указатель на константу int.

**bar** является константой или фиксированным указателем на значение, которое можно изменить.

1. **Постоянная ссылка:**

Ссылка на переменную (здесь int), которая является постоянной. Мы передаем переменную как ссылку главным образом, потому что *ссылки меньше по размеру, чем фактическое значение*.

int **var0 = 0;**

constint **&ptr1 = var0;**

ptr1 = 8; // Error

var0 = 6; // OK

1. **Константные указатели**

int **var1 = 1;**

int **var2 = 0;**

int **\***const **ptr2 = &var1;**

ptr2 = &var2; // Error

1. **Указатель на константу**

intconst **\* ptr3 = &var2;**

\*ptr3 = 4; // Error

1. **Постоянный указатель на константу**

Постоянным указателем на константу является указатель, который не может ни изменить адрес, на который он указывает, ни изменить значение, сохраненное на этом адресе, и не изменять его.

int **var3 = 0;**

int **var4 = 0;**

constint **\*** const **ptr4 = &vВar3;**

\*ptr4 = 1; // Error

**ptr4 = &var4;** // Error

Возврат структуры

#include<stdio.h>

#include<string.h>

#include<iostream>

struct **Object {**

public:

int **a;**

int **b;**

int **c;**

char **string[50];**

};

Object func() {

**Object o;**

**o.a = 1;**

**o.b = 2;**

**o.c = 3;**

**strcpy(o.string,** "Hello world!");

return **o;**

}

int **main() {**

**Object o = func();**

**printf(**"%d\n",o.a);

**printf(**"%d\n",o.b);

**printf(**"%d\n",o.c);

**printf(**"%s\n",o.string);

**system(**"PAUSE");

return **0;**

}

В качестве функции возвращаемой структурой func типа Object

## 53. Структуры и битовые поля. Структуры. Их дальнейшее использование. Объединения. Объединения: синтаксис и правила. Создание простого объединения.

Структуры и битовые поля

**Битовые** **поля** - это специальный тип структуры, в котором определено, из скольких **бит** состоит каждое поле. Они позволяют рационально использовать память с помощью хранения данных в минимально требуемом количестве битов, позволяют работать с отдельными битами и их группами (полями) внутри переменных. Элементы битового поля должны быть объявлены как тип **int**  или **unsigned**. Вне структур битовые поля объявлять нельзя. Нельзя также организовывать массивы битовых полей и нельзя применять к полям операцию определения адреса **&**.

**struct**

**{ unsigned**идентификатор 1 **:**  длина\_поля\_1**;**

**unsigned**идентификатор 2 **:**длина\_поля\_2**;**

**}**

Длина-поля задается целым выражением или константой. Эта константа определяет число битов, отведенное соответствующему полю. Битовое поле рассматривается как **целое** число, максимальное значение которого определяется длиной поля.

**struct number**

**{**

**unsigned group: 4;** // 4 бита – числа от 0 до 15

**unsigned department: 3;**  // 3 бита – числа от 0 до 7

**unsigned course: 3;** // 3 бита – числа от 0 до 7

**}**

Это описание включает три битовых поля типа **unsigned: group,  department** и **course**, используемых для предст. номера. При объявл. битового поля вслед за **unsigned** или **int** ставится двоеточие (:) и пишется целочисл. константа, задающая ширину поля (т.е. число битов, в кот. хранится этот член структуры).

Битовые поля удобно использовать для решения задач, связанных с **булевой** **логикой**. Напр., представление цветов, шифрование данных, перевод чисел в двоичную систему счисления и т.п.

Структуры. Их дальнейшее использование

Структура может считаться прерходным типом данных от структурного программирования к ООП. Развитие структуры – тип данных класс. Struct и class позволяют решать множество типовых задач. Для решения бользователь может создать любой собственный тип данных. Наиболее распространённые новые типы: очереди, двоичные деревья, графы и т.д. Эти типы эффективны для поиска и сортировки данных.

Объединения. Объединения: синтаксис и правила. Создание простого объединения.

Объединения - это объект, позволяющий нескольким переменным различных типов занимать один участок памяти.  Объединение подобно структуре, однако в каждый момент времени может использоваться только один из элементов объединения.

Вместо того, чтобы впустую растрачивать память на объекты, которые используются не постоянно, можно поместить их в объединение, где они будут делить между собой одну и ту же область памяти.

union идентификатор\_объединения

{ описание элемента 1;

………………….

описание элемента n

};

Когда используется элемент меньшей длины, то переменная. типа объединения может содержать неиспользуемую память.

union Info

{ char name[30];

int age;

int telephon;

} Inf ;

using namespace std;

void main()

{ Inf.telephon = 1234567;

Inf.age = 20;

cout<<Inf.age<<endl;

cout<<Inf.telephon<<endl;

}

При использовании объекта Inf типа union можно обрабатывать только тот элемент, который получил значение, т.е. после присвоения значения элементу Information.age не имеет смысла обращаться к другим элементам. Надо сначала присвоить им значения.

Inf.telephon = 1234567;

Допустимые операции:

* операция присваивания значения одного объединения другому объединению того же типа;
* операция вычисления адреса объединения (&);
* доступ к элементу объединения при помощи операций . и –>

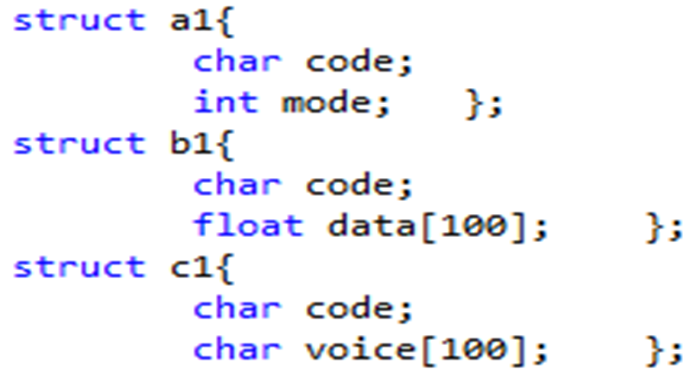
Над объединениями нельзя выполнять операции сравнения.

Свойства объединений:

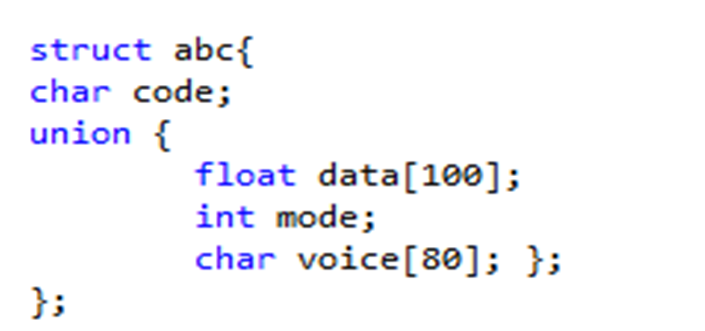
* размещение всех членов объединения начинается с одного и того же адреса;
* в любой момент времени можно присвоить значение лишь одному члену;
* не бывает статических членов;
* не бывает членов, являющихся ссылками.

Типичной ошибкой является инициализация объединения при его объявлении значением или выражением, тип которого отличается от типа первого элемента объединения.

Можно вместо трех структур:



создать одну



Объединения могут входить в структуры и массивы, и наоборот. Запись доступа к элементу объединения, находящегося в структуре (как и структуры, находящейся в объединении), такая же, как и для вложенных структур. Например, в массиве структур

struct {

char \*name;

int flags;

int utype;

union {

int ival;

float fval;

char \*sval;

} u;

} symtab[NSYM];

к *ival* обращаются следующим образом:

symtab[i].u.ival

а к первому символу строки *sval* можно обратиться любым из следующих двух способов:

\*symtab[i].u.sval

symtab[i].u.sval[0]

Операции, применимые к структурам, годятся и для объединений, т. е. законны присваивание объединения и копирование его как единого целого, взятие адреса от объединения и доступ к отдельным его элементам. Инициализировать объединение можно только значением, имеющим тип его первого элемента; таким образом, упомянутую выше переменную *u* можно инициализировать лишь значением типа *int*.

## 54. Вспомогательные средства. Использование typedef. Использование enum. Сложные формы данных. Функции работы с датой и временем.

**Использование typedef**

Язык С позволяет определять имена новых типов данных с помощью ключевого слова typedef. На самом деле здесь не создается новый тип данных, а определяется новое имя существующему типу. Он позволяет облегчить создание машинно-независимых программ. Единственное, что потребуется при переходе на другую платформу, - это изменить оператор typedef. Он также может помочь документировать код, позволяя назначать содержательные имена стандартным типам данных. Стандартный вид оператора typedef следующий:

*typedef тип имя;*

где тип — это любой существующий тип данных, а имя - это новое имя для данного типа. Новое имя определяется в дополнение к существующему имени типа, а не замещает его. Например, можно создать новое имя для float, используя

*typedef float balance;*

Данный оператор сообщает компилятору о необходимости распознавания balance как другого имени для float. Далее можно создать вещественную переменную, используя balance:

*balance past\_due;*

Здесь past\_due - это вещественная переменная типа balance, другими словами - типа float. Можно использовать typedef для создания имен для более сложных типов. Например:

*typedef struct {*

*float due;*

*int over\_due;*

*char name[40];*

*} client; /\* здесь client - это имя нового типа \*/*

*client clist[NUM\_CLIENTS]; /\* определение массива структур типа client \*/*

Использование typedef может помочь при создании более легкого для чтения и более переносимого кода. Но надо помнить, что на самом деле не создаются никакие новые типы данных.

**Перечисления в С++ (enum)**

Используется для еречисления констант

*#include <iostream>*

*using namespace std;*

*enum weddingAnn {chintz = 1, paper, leather, linen, wooden} year;//определяем перечисление и объявляем переменную*

*int main()*

*{*

*setlocale(LC\_ALL, "rus");*

*cout << "Олег\t" << chintz << "выполняет";*

*cout << "\n";*

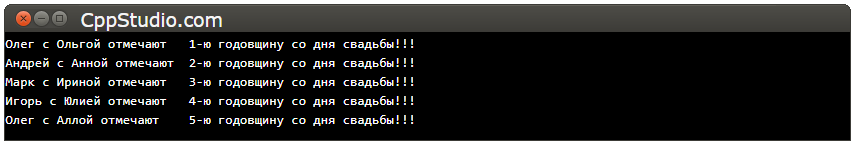
*cout << "Андрей\t" << paper << "выполняет";*

*return 0;*

*}*

Определяем перечисление — **строка 4** . Для этого используется зарезервированное слово enum, далее даем название типу — в нашем случае weddingAnn, в фигурных скобках объявляем целочисленные константы (элементы перечисления)  {chintz = 1, paper, leather, linen, wooden}, объявляем переменную (декларатор) year    в конце ставим ;. Объявление типа  и переменной  необязательно  при определении перечисления. В нашем примере мы могли обойтись и без них — enum {chintz = 1, paper, leather, linen, wooden};, но лучше сразу показать полную версию. Вы, скорее всего, обратили внимание на то, что первому элементу chintz мы присвоили значение 1. Дело в том, что по умолчанию, при определении перечисления, первой перечисляемой константе присваивается значение 0, а для остальных на единицу больше, чем у предыдущей. Поэтому нам достаточно было присвоить значение только первой константе. Остальные значения присвоились автоматически. При выводе данных на экран , **строки 10 — 18**, мы указываем название годовщин свадеб, а компилятор уже обращается к элементам перечисления и предоставляет нам целое число, которое хранит этот элемент (идентификатор).

Что мы увидим в консоли:

**

Так — с этим, надеюсь, разобрались. Теперь немного теории — то что важно запомнить.

Первое, то что мы уже знаем: всем идентификаторам (целочисленным константам) перечисления присваивается значение. По умолчанию первому – 0, остальным на единицу больше предыдущего значения. А если мы сами задаем значение — то это может быть любое целое число. При этом, ВАЖНО, имена целочисленных констант должны быть уникальны. При определении одинаковых имен компилятор выдаст ошибку. А вот значения, которые мы присваиваем константам могут совпадать. Например: enum year{vadim = 2, vanya = 2, sonya, yula = 0, denis = sonya + 20,}; Когда выведем эти значения на экран, увидим 2, 2, 3, 0, 23.

Перечисление — это отдельный тип данных. Так типом идентификаторов vadim, vanya, sonya и т.д. выступает тип year.

Значение константам можно присвоить только в фигурных скобках при определении перечисления. Изменить его ниже в коде уже нет возможности.

Определяя перечисление, объявление типа и переменной необязательно.

Переменная (декларатор) типа созданного нами перечисления может иметь значение одной из определённой в перечислении константы. Чтобы было понятней, если в функции main мы объявляем переменную типа перечисления, это будет выглядеть так — year child = sonya;.

используя перечисления, мы можем сделать наши коды программ более понятными. В первую очередь потому, что благодаря им возможно использовать в программе не «магические числа» инициализированных где-то в коде переменных, а читаемые названия элементов перечисления. Но, как всегда, не увлекайтесь. Всего в меру.

Рассмотрим еще один код программы. Здесь мы создадим перечисление level (этаж) и предложим пользователю проехаться на виртуальном лифте по виртуальному торговому центру.

# 55.Динамические структуры данных C++. Линейные однонаправленные списки. Общие сведения. Однонаправленные списки без заглавного звена. Построение списка с заглавным звеном. Удаление списка из памяти.

## Линейные однонаправленные списки.

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

Каждый элемент списка имеет адрес на следующий элемент

Каждый элемент списка представим структурой языка C++ с двумя полями:

* **информационное поле**, которое в общем случае может содержать произвольное количество полей разных типов.
* **ссылка на следующий элемент списка**.

struct **node**

**{**

int **elem;** //Информационный элемент звена списка

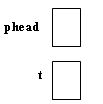
**node \*sled;** // Указатель на следующее звено списка

**};**

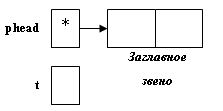
## Однонаправленные списки без заглавного звена.

 Алгоритм построения списка:

1. struct node \*phead;
2. struct node \*t;

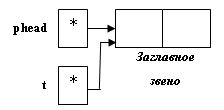
  
Рис.1. Резервирование места в памяти

1. В куче зарезервируем место для динамического объекта, на который указывает **phead**:
2. phead = new(node);

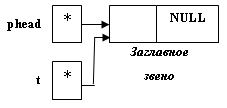
  
Рис.2. Выделили место для элемента списка

    Построено ***заглавное звено*** будущего однонаправленного списка.

1. Выполним еще ряд подготовительных действий:
2. t = phead;

  
Рис.3. Присвоим значение **t**

(\*t).sled = NULL;

  
Рис.4. Присвоим указателю заглавного звена **NULL**

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **value;**

**node \*next;**

};

void **main ()**

{

int **i;**

**node \*phead, \*t;**

**setlocale(LC\_ALL,**"Rus");

**phead =** new **(node);**

**t = phead;**

**(\*t).value = 1;**

**(\*t).next =** new **(node);**

**t = (\*t).next;**

**(\*t).value = 2;**

**(\*t).next =** new **(node);**

**t = (\*t).next;**

**(\*t).value = 6;**

**(\*t).next =** new **(node);**

**t = (\*t).next;**

**(\*t).value = 17;**

**(\*t).next =** new **(node);**

**(\*t).next = NULL;**

// Вывод содержимого информационных полей списка

for **(t = phead; t != NULL; t = (\*t).next)**

**cout << (\*t).value <<** " ";

**cout <<** "\n";

**system(**"PAUSE");

}

## Построение списка без заглавного звена.

    Линейный список имеет доп звено, чащ всего такого же типа, как и все элементы

1) Вначале отведем место для указателей в статической памяти:

struct **node \*phead;**

struct **node \*t;**

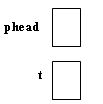


Рис.1. Резервирование места в памяти

2) В куче зарезервируем место для динамического объекта, на который указывает **phead**:

**phead =** new(node);

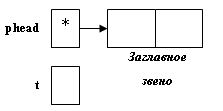


Рис.2. Выделили место для элемента списка

Построено **заглавное звено** будущего однонаправленного списка.

Выполним еще ряд подготовительных действий:

**t = phead;**

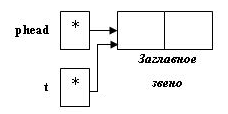


Рис.3. Присвоим значение t

**(\*t).sled = NULL;**

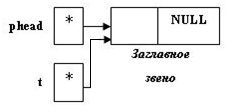


Рис.4. Присвоим указателю заглавного звена NULL

Теперь можно приступить к циклическому процессу построения списка. Идентификатор **Число** обозначает объект языка C++, вводимый с клавиатуры.

**cin >> Число;**

while **(Число != Числу, определяющему окончание ввода)**

**{**

**(\*t).sled =** new **(node);** //Резервируем место для нового объекта.

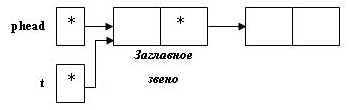


Рис.5. Создаем новый объект

**t = (\*t).sled;** //Указатель t содержит адрес

//расположения созданного объекта.

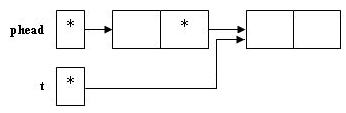


Рис.6. Определяем t

**(\*t).elem = Число;** //Заполняем поля объекта.

**(\*t).sled = NULL;**

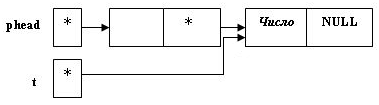


Рис.7. Результат заполнения полей

**cin >> Число;** //Запрос на ввод следующего значения.

**}**

Алгоритм построения оформим в виде функции на языке C++:

void **POSTROENIE (node \*\*phead)**

// Построение списка с заглавным звеном.

// \*phead - указатель на заглавное звено.

{

**node \*t;**

int **el;**

// Вначале создадим заглавное звено

**\*phead =** new **(node);**

**t = \*phead; (\*t).sled = NULL;**

**cout <<** "Вводите элементы звеньев списка: ";

**cin >> el;**

while **(el!=0)**

**{**

**(\*t).sled =** new **(node); t = (\*t).sled;**

**(\*t).elem = el; (\*t).sled = NULL;**

**cin >> el;**

**}**

}

## Алгоритм удаления списка из памяти.

Алгоритм удаления можно сформулировать следующим образом:

* заводим два указателя **q** и **q1**, один из которых будет "опережать" другой (пусть **q1** опережает **q**);
* начальное значение **q** - это адрес расположения в памяти заглавного звена, а **q1** - адрес расположения первого элемента списка;
* организуем цикл, пока **q1 != NULL**, то есть пока **q1** "не доберется" до указателя последнего элемента списка;
* в теле цикла переместим указатели на следующую пару элементов списка (то есть для первого случая **q** будет содержать адрес первого звена списка, а **q1** - второго), и удалим элемент, который адресуется значением **q**;
* после выполнения цикла у нас останется только заглавное звено, адресуемое указателем **phead**. Его также нужно удалить.

Приведем текст функции, реализующей указанный алгоритм.

void **OCHISTKA (node \*\*phead)**

//Удаление однонаправленного списка из памяти.

// \*phead - указатель на заглавное звено списка.

{

struct **node \*q, \*q1;** // Рабочие указатели.

**q = \*phead;**

**q1 = (\*q).sled;** // Указатель q1 "опережает" указатель q.

while **(q1!=NULL)**

**{ q = q1; q1 = (\*q1).sled;** delete **q;}**

delete **\*phead;** //Удаление заглавного звена.

}

Отметим, что одной из распространенных ошибок при составлении программ является неаккуратное обращение с динамически распределяемой памятью, в частности, "забывание" освободить ее после использования, что в конце концов может привести к аварийному завершению программы из-за нехватки оперативной памяти. Поиск же мест "засорения" оперативной памяти в отлаживаемых программах, занимающих значительный объем, может занять много места, поскольку к моменту комплексной отладки, когда эти ошибки попадаются на глаза, естественным образом забываются многие детали создания программных компонентов. Поэтому целесообразно с самого начала отладки программ отслеживать процесс возврата динамической памяти в "кучу".

# 56.Операции над списками с заглавным звеном. Поиск звена. Включение звена после звена (1-й случай). Включение звена с заданным информационным полем перед звеном (2-й случай). Удаление звена после звена (1-й случай). Удаление звена на которое указывает ссылка (2-й случай).

## Поиск звена

***алгоритм поиска звена***.

    Приведем алгоритм последовательного поиска звена с заданным значением информационного поля в однонаправленном списке, записанный в виде функции языка **C++**:

void POISK (node \*\*phead,int el, node \*\*Res)

// Поиск звена с элементом el в списке, заданном указателем \*phead.

// В случае успешного поиска в \*Res находится адрес

// звена списка, содержащего элемент el, в случае неуспеха в \*Res помещается NULL.

{

node \*t;

\*Res = NULL;

t = \*phead; t = (\*t).sled;

while (t!=NULL && \*Res==NULL)

if ((\*t).elem==el) \*Res = t; else t = (\*t).sled;

}

## Первый случай включения звена в список.

    На этом шаге мы рассмотрим ***алгоритм включения звена в список после звена, на которое указывает заданная ссылка***.

    Предположим, что имеется однонаправленный список с заглавным звеном, и необходимо вставить звено с заданным информационным полем после звена, на которое указывает ссылка **Res**. Вставка звена осуществляется по следующему алгоритму.

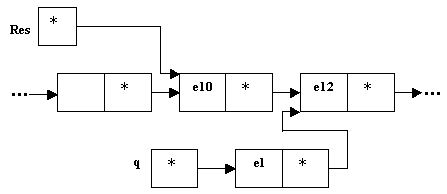
1. В куче резервируется место для динамического объекта:
2. q = new (node);

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris6_1.jpg  
Рис.1. Зарезервировали место под элемент списка

1. В информационное поле этого объекта помещается значение элемента, который необходимо вставить:
2. (\*q).elem = el;

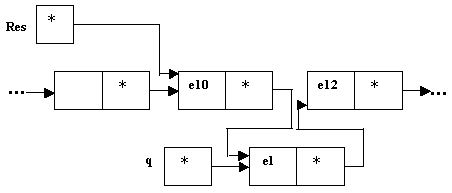
http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris6_2.jpg  
Рис.2. Заполнение информационного поля

1. В поле указателя помещается адрес элемента, следующего за звеном, на которое указывает **Res**:
2. (\*q).sled = (\*Res).sled;

  
Рис.3. Заполнение поля указателя вставляемого элемента

1. И, наконец, после выполнения оператора
2. (\*Res).sled = q;

получаем результат, изображенный на следующей схеме:

  
Рис.4. Изменение указателя у предыдущего элемента

    Алгоритм оформим в виде функции.

void VSTAV (node \*\*Res, int el)

// Включение звена с информационным полем el

//после звена, на которое указывает ссылка \*Res.

{

node \*q;

q = new (node);

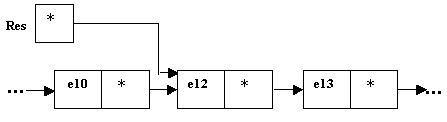
(\*q).elem = el; (\*q).sled = (\*\*Res).sled;

(\*\*Res).sled = q;

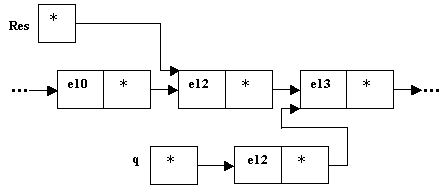
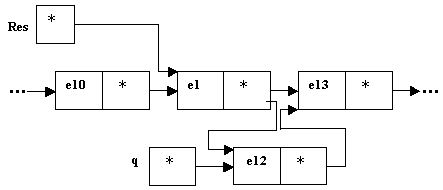
}

## Второй случай включения звена

    На этом шаге мы рассмотрим ***алгоритм включения в однонаправленный список звена с заданным информационным полем перед звеном, на которое указывает заданная ссылка* Res**.

  
Рис.1. Начальное положение

    Опишем алгоритм при помощи схем Д.Кнута "до и после":

1. q = new (node);
2. http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris7_2.jpg  
   Рис.2. Выделение места в памяти для нового звена
3. (\*q).elem = (\*Res).elem;
4. (\*q).sled = (\*Res).sled;
5.   
   Рис.3. Заполнение полей нового элемента
6. (\*Res).elem = el;
7. (\*Res).sled = q;
8.   
   Рис.4. Включение элемента в список

    Приведем функцию, реализующую описанное включение:

void VSTAV1 (node \*\*Res, int el)

//Включение звена с информационным полем el

//перед звеном, на которое указывает \*Res.

{

node \*q;

q = new (node);

(\*q).elem = (\*\*Res).elem; (\*q).sled = (\*\*Res).sled;

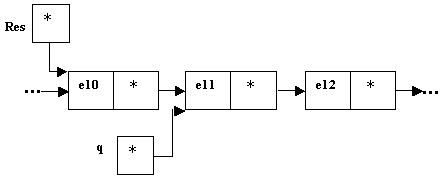
(\*\*Res).elem = el; (\*\*Res).sled = q;

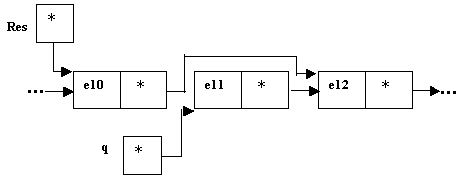
}

## Алгоритм удаления звена(1).

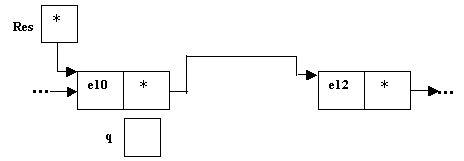
    На этом шаге мы рассмотрим ***алгоритм удаления звена, расположенного после звена, на которое указывает ссылка* Res**.

    Воспользуемся схемами "до и после":

1. q = (\*Res).sled;
2.   
   Рис.1. Определение местоположения удаляемого звена.
3. Проверяем, не является ли звено, после которого нужно удалять, последним. В этом случае удалять нечего.
4. if (q!=NULL) //Если звено, после которого нужно удалять,
5. // не является последним, то...
6. {
7. (\*Res).sled = (\*(\*Res).sled).sled;

  
Рис.2. Исключение удаляемого элемента из списка

1. Последняя тонкость: присоединение "кусочка" **heap**-области к списку свободной памяти:
2. delete q;

  
Рис.3. Возврат памяти в кучу

}

    Приведем текст функции удаления:

void YDALE (node \*\*Res)

// Удаление звена, расположенного после

// звена, на которое указывает ссылка \*Res.

{

node \*q;

q = (\*\*Res).sled;

if (q!=NULL)

// Если звено, после которого нужно удалять,

// не является последним, то...

{ (\*\*Res).sled = (\*(\*\*Res).sled).sled; delete q; }

else

cout<<"Звено с заданным элементом - последнее!\n";

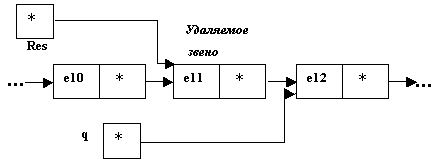
}

## Алгоритм удаления звена(2)

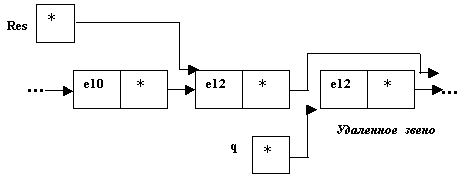
На которое указывает ссылка Res.

Воспользуемся схемами "до и после":

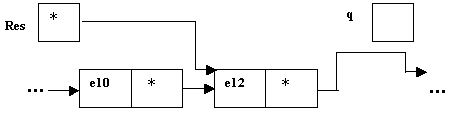
1. Определяем местоположение следующего за удаляемым элемента.
2. q = (\*Res).sled;

  
Рис.1. Сохранение адреса следующего элемента

1. Определяем, не является ли удаляемое звено последним. В зависимости от этого реализация алгоритма будет различной.
2. if (q!=NULL)
3. {//Если удаляемое звено не является последним, то ...
4. (\*Res).elem = (\*q).elem;
5. (\*Res).sled = (\*q).sled;

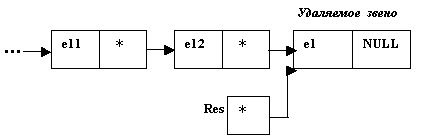
  
Рис.2. "Переписываем" следующий элемент в удаляемый

1. Последняя тонкость: присоединение "кусочка" **heap**-области к списку свободной памяти:
2. delete q;

  
Рис.3. Возврат памяти в кучу

}

1. Теперь рассмотрим ситуацию, когда удаляемое звено является ***последним звеном*** списка:

  
Рис.4. Удаляемый элемент - последний

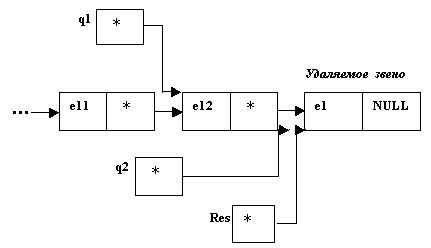
    Найдем указатель на предпоследнее звено линейного списка с помощью двух вспомогательных указателей **q1** и **q2**, перемещающихся по списку "параллельно друг другу", причем указатель **q2** "опережает" указатель **q1** на один шаг.

q1 = phead; q2 = (\*q1).sled; //Инициализация указателей.

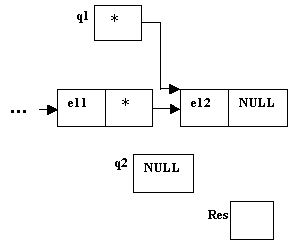
while (q2!=Res)

{ q1 = q2; q2 = (\*q2).sled;}

    После выполнения цикла **while**, мы получим следующую ситуацию:

  
Рис.5. Положение указателей

1. Приступим к удалению:
2. (\*q1).sled = NULL; q2 = NULL; delete (Res);

  
Рис.6. После удаления

    Приведем текст функции удаления:

void YDALE1 (node \*\*phead, node \*\*Res)

//Удаление звена, на которое указывает ссылка \*Res

//из однонаправленного списка с заглавным звеном.

//\*phead - указатель на заглавное звено списка.

{

node \*q,\*q1,\*q2;

q = (\*\*Res).sled;

if (q!=NULL)

{ (\*\*Res).elem = (\*q).elem; (\*\*Res).sled = (\*q).sled;delete q; }

else

{ q1 = \*phead; q2 = (\*q1).sled; //Инициализация указателей.

while (q2!=\*Res)

{ q1 = q2; q2 = (\*q2).sled; }

(\*q1).sled = NULL; q2 = NULL; delete \*Res;

}

}

# 57.Динамические структуры данных C++. Ортогональные списки. Реализация операций над ортогональными списками.

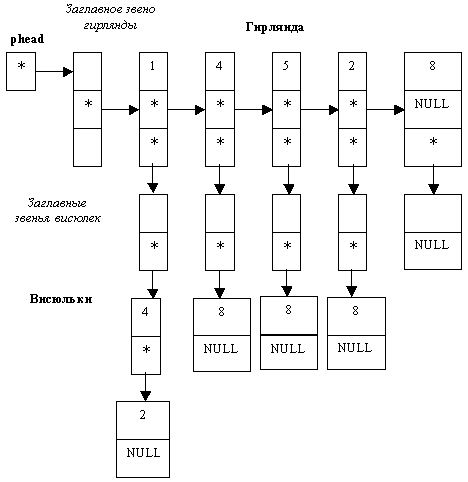
## Ортогональные списки.

    До сих пор мы рассматривали ***линейные структуры динамических переменных***. Добавление к динамической переменной двух и более полей указателей создает возможность построения ***нелинейных структур***. Дело в том, что при решении практических задач обычно не удается обойтись только линейными динамическими структурами данных (***список, очередь, стек, дек*** и т.д.): приходится создавать структуры данных, максимально отражающие существо выполняемой исполнителем задачи.

    Мы рассмотрим только простейшие нелинейные динамические структуры, которые называются ***ортогональными списочными структурами (ортогональными списками, многосвязными списками)***.

    Более точно, ***ортогональными списками*** называется списочная структура данных, в которой узлы могут принадлежать более чем одному списку и содержать более одного указателя [1, с.241].

    На рисунке изображено графическое представление ортогональных списков:

  
Рис.1. "Гирлянды" и "висюльки"

    Горизонтальный линейный однонаправленный список с заглавным звеном мы будем называть ***гирляндой***. Каждое звено этого списка содержит три поля, причем, если указатель **P** указывает на звено гирлянды, то:

* поле **(\*P).Key** является ***информационным полем*** узла гирлянды;
* поле **(\*P).Next** содержит ***указатель на следующее звено*** гирлянды;
* поле **(\*P).Trail** содержит ***указатель на линейный однонаправленный список с заглавным звеном***, который называется ***висюлькой*** (английское слово **Trail** переводится как "тащиться, свисать, волочиться").

    Звено каждой висюльки содержит два поля: **Id** и **Next**, причем, если указатель **Q** указывает на элемент висюльки, то:

* поле **(\*Q).Id** является ***информационным полем*** звена висюльки;
* поле **(\*Q).Next** ***указывает на следующее звено*** данной висюльки.

## Основные операции над ортогональными списками.

    Приведем реализацию на языке **C++** простейших операций над ортогональными списками ("гирляндо-висюлечной" структурой).

    Сначала опишем типы данных:

// Описание типа звена гирлянды.

struct nodeGir

{

int elem; // Информационное поле звена гирлянды.

nodeVis \*vniz; // Указатель на звено висюльки.

nodeGir \*sled; // Указатель на звено гирлянды.

};

// Описание типа звена висюльки.

struct nodeVis

{

int elem; // Информационное поле звена висюльки.

nodeVis \*vniz; // Указатель на звено висюльки.

};

    Приведем программу, демонстрирующую работу с "гирляндо-висюлечной" структурой.

#include<iostream.h>

struct nodeVis

{

int elem; //Информационное поле звена висюльки.

nodeVis \*vniz;//Указатель на звено висюльки.

};

struct nodeGir

{

int elem;//Информационное поле звена гирлянды.

nodeVis \*vniz;//Указатель на звено висюльки.

nodeGir \*sled;//Указатель на звено гирлянды.

};

class GirVis {

private:

nodeGir \*phead;//Голова гирлянды.

nodeVis \*pheadVis;//Голова висюльки.

void VisVyvod ();

public:

GirVis() {phead = new (nodeGir); }

~GirVis() {delete phead;}

nodeVis \*VisPostr ();

nodeVis\* VisPoisk (int);

void SetpheadVis (nodeVis \*r) {pheadVis=r;} //Определение головы висюльки.

void VisVstav (nodeVis \*,int);

void Vis1Vstav (nodeVis \*,int);

void VisUdale (nodeVis \*);

void Vis1Udale (nodeVis \*);

void GirPostr ();

void GirVyvod ();

nodeGir \*GirPoisk (int);

void OCHISTKA();

void OCHISTKA1();

};

void main ()

{

GirVis A;

int el,elGir,elVis;

nodeGir \*Res; //Рабочий указатель.

nodeVis \*ResVis; //Указатель на звено висюльки.

A.GirPostr ();

A.GirVyvod ();

cout<<"\nВведите элемент звена гирлянды, ";

cout<<"чьи висюльки будем изменять:\n";

cin>>elGir;

cout<<"\nВведите элемент звена висюльки, после которого ";

cout<<"осуществляется вставка:\n";

cin>>elVis;

cout<<"\nВведите вставляемый элемент:\n";

cin>>el;

//Поиск элемента elGir в гирлянде.

Res=A.GirPoisk (elGir);

if (Res!=NULL)

{

//Поиск элемента elVis в висюльке.

A.SetpheadVis((\*Res).vniz);

ResVis=A.VisPoisk (elVis);

if (ResVis!=NULL)

A.VisVstav (ResVis,el);

else cout<<"Элемента в висюльке нет!\n";

}

else cout<<"Элемента в гирлянде нет\n";

A.GirVyvod ();

cout<<"\nВведите элемент гирлянды, чью висюльку будем изменять:\n";

cin>>elGir;

cout<<"Введите элемент висюльки, перед которым ";

cout<<"осуществляется вставка:\n";

cin>>elVis;

cout<<"Введите вставляемый элемент:\n";

cin>>el;

//Поиск элемента elGir в гирлянде.

Res=A.GirPoisk (elGir);

if (Res!=NULL)

{

//Поиск элемента elVis в висюльке.

A.SetpheadVis((\*Res).vniz);

ResVis=A.VisPoisk (elVis);

if (ResVis!=NULL)

A.Vis1Vstav (ResVis,el);

else cout<<"Элемента в висюльке нет!\n";

}

else cout<<"Элемента в гирлянде нет!\n";

A.GirVyvod ();

cout<<"\nВведите элемент гирлянды, чью висюльку будем изменять:\n";

cin>>elGir;

cout<<"Введите элемент висюльки, после которого нужно удалить:\n";

cin>>elVis;

//Поиск элемента elGir в гирлянде.

Res=A.GirPoisk (elGir);

if (Res!=NULL)

{

//Поиск элемента elVis в висюльке.

A.SetpheadVis((\*Res).vniz);

ResVis=A.VisPoisk (elVis);

if ((ResVis!=NULL) && ((\*ResVis).vniz!=NULL))

A.VisUdale (ResVis);

else cout<<"Элемента в висюльке нет!\n";

}

else cout<<"Элемента в гирлянде нет!\n";

A.GirVyvod ();

cout<<"\nВведите элемент гирлянды, чью висюльку будем изменять:\n";

cin>>elGir;

cout<<"Введите элемент висюльки, который удаляется:\n";

cin>>elVis;

//Поиск элемента elGir в гирлянде.

Res=A.GirPoisk (elGir);

if (Res!=NULL)

{

//Поиск элемента elVis в висюльке.

A.SetpheadVis((\*Res).vniz);

ResVis=A.VisPoisk (elVis);

if ((ResVis!=NULL) && ((\*ResVis).vniz!=NULL))

A.Vis1Udale (ResVis);

else cout<<"Элемента в висюльке нет или он последний!\n";

}

else cout<<"Элемента в гирлянде нет!\n";

A.GirVyvod ();

A.OCHISTKA();

}

void GirVis::OCHISTKA()

{

nodeGir \*q,\*q1;//Рабочие указатели.

q = phead;

q1 = (\*q).sled; //Указатель q1 "опережает" указатель q.

while (q1!=NULL)

{ q = q1; q1 = (\*q1).sled;

pheadVis=(\*q).vniz;

OCHISTKA1(); //Очистка висюльки.

delete q;}

}

void GirVis::OCHISTKA1()

{

nodeVis \*q,\*q1;

q=pheadVis;

q1 = (\*q).vniz;

while (q1!=NULL)

{ q = q1; q1 = (\*q1).vniz;

delete q;}

}

void GirVis::GirPostr ()

//Построение однонаправленного списка с заглавным звеном,

//заданного указателем phead (построение гирлянды).

{

nodeGir \*t;

int el;

t = phead; (\*t).sled = NULL;

cout<<"Вводите элемент гирлянды: \n";

cin>>el;

while (el!=0)

{

(\*t).sled = new (nodeGir);

t = (\*t).sled; (\*t).elem = el; (\*t).sled = NULL;

(\*t).vniz=VisPostr();

cout<<" Вводите элемент гирлянды: \n";

cin>>el;

}

}

nodeVis \*GirVis::VisPostr ()

//Построение однонаправленного списка с заглавным звеном

//(построение висюльки). pheadVis - указатель на висюльку.

{

nodeVis \*t;

int el;

//Создадим заглавное звено списка.

pheadVis = new (nodeVis);

t = pheadVis; (\*t).vniz = NULL;

cout<<"Вводите элементы звеньев висюльки: \n";

cin>>el;

while (el!=0)

{

(\*t).vniz = new (nodeVis);

t = (\*t).vniz; (\*t).elem = el; (\*t).vniz = NULL;

cin>>el;

}

return pheadVis;

}

void GirVis::GirVyvod ()

//Вывод содержимого однонаправленного списка, заданного

//указателем phead (вывод содержимого гирлянды).

{

nodeGir \*t;

t = phead; t = (\*t).sled;

cout<<"Гирлянда: ";

while (t!=NULL)

{

cout<<(\*t).elem<<" ";

pheadVis=(\*t).vniz;

VisVyvod ();

t = (\*t).sled;

}

}

nodeGir \*GirVis::GirPoisk (int el)

//Поиск элемента el в списке, заданном указателем phead.

//В случае успешного поиска возвращается адрес звена списка,

//содержащего элемент el. В противном случае - NULL.

{

nodeGir \*t,\*r;

r = NULL; t = phead; t = (\*t).sled;

while (t!=NULL && r==NULL)

if ((\*t).elem==el) r = t;

else t = (\*t).sled;

return r;

}

void GirVis::VisVyvod ()

//Вывод содержимого однонаправленного списка с заглавным звеном,

//заданного указателем pheadVis (вывод содержимого висюльки).

{

nodeVis \*t;

t = pheadVis; t = (\*t).vniz;

cout<<"(";

while (t!=NULL)

{

cout<<(\*t).elem<<" "; t = (\*t).vniz;

}

cout<<")";

}

nodeVis \*GirVis::VisPoisk (int el)

//Поиск элемента el в списке, заданном указателем pheadVis.

//В случае успешного поиска возвращается адрес звена списка,

//содержащего элемент el. В противном случае - NULL.

{

nodeVis \*t,\*r;

r = NULL; t = pheadVis; t = (\*t).vniz;

while (t!=NULL && r==NULL)

if ((\*t).elem==el) r = t;

else t = (\*t).vniz;

return r;

}

void GirVis::VisVstav (nodeVis \*r,int el)

//Включение звена с информационным полем el

//после звена, на которое указывает r

//(включение звена в висюльку).

{

nodeVis \*q;

q = new (nodeVis);

(\*q).elem = el; (\*q).vniz = (\*r).vniz; (\*r).vniz = q;

}

void GirVis::Vis1Vstav (nodeVis \*r,int el)

//Включение звена с информационным полем el

//перед звеном, на которое указывает r

//(включение звена в висюльку).

{

nodeVis \*q;

q = new (nodeVis);

(\*q).elem = (\*r).elem; (\*q).vniz = (\*r).vniz;

(\*r).elem = el; (\*r).vniz = q;

}

void GirVis::VisUdale (nodeVis \*r)

//Удаление звена, расположенного после звена,

//на которое указывает ссылка r

//(удаление звена висюльки).

{

nodeVis \*q;

q = (\*r).vniz;

if ((\*r).vniz!=NULL)

{

(\*r).vniz = (\*(\*r).vniz).vniz; delete q;

}

else cout<<"Звено с заданным элементом - последнее!\n";

}

void GirVis::Vis1Udale (nodeVis \*r)

//Удаление звена, на которое указывает ссылка r

//(удаление звена висюльки).

{

nodeVis \*g;

if ((\*r).vniz!=NULL)

{

g = (\*r).vniz;

(\*r).elem = (\*(\*r).vniz).elem;

(\*r).vniz = (\*(\*r).vniz).vniz;

delete g;

}

else cout<<"Не умею удалять последнее звено!\n";

}

# 58.Динамические структуры данных C++. Кольцевые списки. Построение и вывод кольца. Основные операции.

## построение и вывод кольца.

    Хотя структура в виде линейного списка является весьма полезной, у нее имеется ряд недостатков. Сейчас мы рассмотрим другие методы организации списков и использование их с целью устранения этих недостатков.

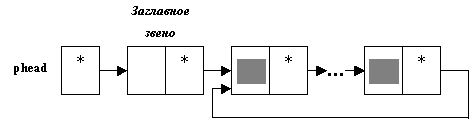
    Один из недостатков линейных списков заключается в том, что, зная указатель **p** на звено списка, мы не имеем доступа к ***предшествующим*** ему звеньям. Если производится просмотр списка, то для повторного обращения к нему исходный указатель на начало списка должен быть сохранен.

    Предположим теперь, что в структуре линейного списка было сделано изменение, при котором поле **sled** последнего элемента содержит указатель ***"назад"*** или на заглавное звено, или на элемент, следующий за заглавным звеном.

    Под ***кольцевым (циклическим) списком*** понимается список, в котором указатель из некоторой ячейки направлен на такое место в списке, откуда данная ячейка может быть достигнута снова [1]. Очевидно, что теперь мы можем из любого звена списка, "перемещаясь" по указателям достичь любого другого звена.

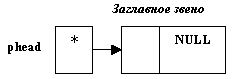
***Кольцевым списком (кольцом)*** на базе линейного однонаправленного списка называется линейный список, в котором указатель из некоторого звена направлен на такое звено в списке, из которого данное звено может быть достигнуто вновь.

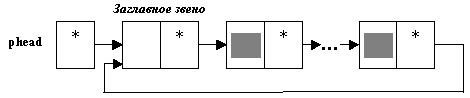
    Опишем два способа представления однонаправленного кольцевого списка с заглавным звеном:

  
Рис.1. Кольцо с удаленным заглавным звеном

    Такой кольцевой список будем называть ***кольцевым списком с удаленным заглавным звеном***.

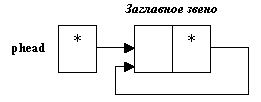
***Пустой кольцевой список с удаленным заглавным звеном*** представим так:

  
Рис.2. Пустое кольцо с удаленным заглавным звеном

  
Рис.3. Кольцо с включенным заглавным звеном

    А этот кольцевой список назовем ***кольцевым списком с включенным заглавным звеном***.

***Пустой кольцевой список с включенным заглавным звеном*** представим так:

  
Рис.4. Пустое кольцо с включенным заглавным звеном

    Рассмотрим способ построения кольцевого списка. Построение выполняется так же, как и в случае линейного однонаправленного списка с заглавным звеном, только после окончания ввода элементов кольцевого списка, в поле указателя последнего звена списка помещается адрес звена, следующего за заглавным.

    Сказанное легко формализуется:

void POSTROENIE (node \*\*phead)

//Построение кольцевого списка с заглавным звеном.

//\*phead - указатель на заглавное звено.

{

int el;

struct node \*t;

// Вначале сформируем заглавное звено.

\*phead = new (node);

t = \*phead; (\*t).sled = NULL;

cout<<"Вводите элементы кольца: "; cin>>el;

while (el!=0)

{ (\*t).sled = new (node); t = (\*t).sled; (\*t).elem = el;

cin>>el;

}

(\*t).sled = (\*(\*phead)).sled;

}

Вывод на экран дисплея содержимого информационных полей кольцевого списка производится до тех пор, пока рабочий указатель, перемещающийся по кольцу, не совпадет с указателем на звено, расположенное после заглавного:

void VYVOD (node \*\*phead)

//Вывод содержимого кольцевого списка с удаленным

// заглавным звеном.

//\*phead - указатель на заглавное звено.

{

struct node \*t;

t = (\*\*phead).sled; cout<< "Кольцо: ";

if (t!=NULL)

{ cout<<(\*t).elem; t = (\*t).sled;

while (t!=(\*\*phead).sled)

{

cout<<(\*t).elem;

t = (\*t).sled; }

}

else cout<<"пусто!\n";

}

## использование основных операций над кольцевыми списками.

**построение кольцевого списка**

**вывод списка, поиск элемента(по данным)**

**вставка звена после заданного**

**вставка звена перед заданным**

**удаление заданного звена**

**удаление звена после заданного**

**очистка кольцевого списка**

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **elem;**

**node \*sled;**

};

class **Spisok {**

private:

**node \*phead,\*Res;**

public:

**Spisok () {phead=**new(node);Res=NULL;}

**~Spisok() {**delete **phead;}**

void **POSTROENIE ();**

void **VYVOD ();**

**node \*POISK (**int);

void **InsAfter (**int);

void **InsBefore (**int);

void **Delete ();**

void **DelAfter ();**

void **OCHISTKA();**

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

int **el,el1;**

**node \*Res\_Zn;**

**A.POSTROENIE();**

**A.VYVOD();**

**cout<<**"\nВведите элемент звена, после которого ";

**cout<<**"осуществляется вставка: ";

**cin>>el;**

**cout<<**"\nВведите элемент вставляемого звена: ";

**cin>>el1;**

if **(A.POISK(el)!=NULL)**

**{ A.InsAfter (el1); A.VYVOD ();}**

else **cout<<**"Звена с заданным элементом в кольце нет!";

**cout<<**"\nВведите элемент звена, перед которым ";

**cout<<**"осуществляется вставка: ";

**cin>>el;**

**cout<<**"Введите элемент вставляемого звена: ";

**cin>>el1;**

if **(A.POISK(el)!=NULL)**

**{**

**A.InsBefore(el1); A.VYVOD ();**

**}**

else **cout<<**"Звена с заданным элементом в кольце нет!";

**cout<<**"\nВведите элемент удаляемого звена: ";

**cin>>el;**

if **(A.POISK(el)!=NULL)**

**{**

**A.Delete (); A.VYVOD ();**

**}**

else **cout<<**"Звена с заданным элементом в кольце нет!";

**cout<<**"\nВведите элемент звена, ";

**cout<<**"после которого нужно удалять: ";

**cin>>el;**

if **(A.POISK(el)!=NULL)**

**{**

**A.DelAfter (); A.VYVOD ();**

**}**

else **cout<<**" Звена с заданным элементом в кольце нет!";

**A.OCHISTKA();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **Spisok::POSTROENIE ()**

//Построение кольцевого списка с удаленным заглавным звеном.

//phead - указатель на заглавное звено.

{

**node \*t;**

int **el;**

**t = phead; (\*t).sled = NULL;**

**cout<<**"Вводите элементы кольца: ";

**cin>>el;**

while **(el!=0)**

**{**

**(\*t).sled =** new **(node);**

**t = (\*t).sled; (\*t).elem = el;**

**cin>>el;}**

**(\*t).sled = (\*phead).sled;**

}

void **Spisok::VYVOD ()**

//Вывод содержимого кольцевого списка с удаленным заглавным звеном.

//phead - указатель на заглавное звено.

{

**node \*t;**

**t = (\*phead).sled;**

**cout<<**"Кольцо: ";

if **(t!=NULL)**

**{**

**cout<<(\*t).elem<<**" "; t = (\*t).sled;

while **(t!=(\*phead).sled)**

**{ cout<<(\*t).elem <<** " "; t = (\*t).sled; }

**}**

else **cout<<**"пусто!\n";

}

node \*Spisok:: POISK (int **el)**

//Поиск элемента el в кольцевом списке phead.

//Если элемент найден, то Res содержит указатель на звено,

//содержащее элемент el. В противном случае - NULL.

{

**node \*t;**

**Res = NULL; t =(\*phead).sled;**

while **((\*t).sled!=(\*phead).sled && Res==NULL)**

if **((\*t).elem==el) Res = t;**

else **t = (\*t).sled;**

if **(Res==NULL && (\*t).elem==el)**

**Res = t;**

return **Res;**

}

void **Spisok::InsAfter (**int **el)**

//Включение звена с информационным полем el в кольцо

//после звена, на которое указывает ссылка Res.

{

**node \*q;**

**q =** new **(node);**

**(\*q).elem = el; (\*q).sled = (\*Res).sled;**

**(\*Res).sled = q;**

}

void **Spisok::InsBefore (**int **el)**

//Включение звена с информационным полем el в кольцо

//перед звеном, на которое указывает ссылка Res.

{

**node \*q;**

**q =** new **(node);**

**(\*q).elem = (\*Res).elem; (\*q).sled = (\*Res).sled;**

**(\*Res).elem = el; (\*Res).sled = q;**

}

void **Spisok::Delete ()**

//Удаление звена, на которое указывает ссылка Res,

//из кольцевого списка с удаленным заглавным звеном,

//заданного указателем phead.

{

**node \*z,\*q;**

if **((\*Res).sled!=(\*phead).sled)**

**{**

**q = (\*Res).sled;**

**(\*Res).elem = (\*((\*Res).sled)).elem;**

**(\*Res).sled = (\*((\*Res).sled)).sled;**

delete **q;**

**}**

elseif **((\*Res).sled==Res)**

**{**

//В кольце единственное звено.

**q = (\*phead).sled; (\*phead).sled = NULL;**

delete **q; cout<<**"Кольцо пусто!";

**}**

else

**{**

//Удаляется "последнее" звено кольца.

**z = phead; q = (\*phead).sled;**

while **(q!=Res)**

**{ z = q; q = (\*q).sled; }**

**(\*z).sled = (\*((\*z).sled)).sled;**

delete **q;**

**}**

}

void **Spisok::DelAfter ()**

//Удаление звена, расположенного после звена,

//на которое указывает ссылка Res,

//из кольцевого списка с удаленным заглавным звеном,

//заданного указателем phead.

{

**node \*q;**

if **((\*Res).sled!=(\*phead).sled)**

**{**

//Ссылка Res не указывает на последнее звено.

**q = (\*Res).sled;**

**(\*Res).sled = (\*((\*Res).sled)).sled;**

delete **q;**

**}**

elseif **((\*Res).sled==Res)**

**{**

//Удаляемое звено - единственное в кольце.

**q = (\*phead).sled; (\*phead).sled = NULL;**

delete **q; cout<<**"Кольцо пусто!";

**}**

else

**{**

//Удаляемое звено - первое в кольце и не единственное.

**q = (\*phead).sled;**

**(\*Res).sled = (\*((\*Res).sled)).sled;**

**(\*phead).sled = (\*Res).sled;** delete **q;**

**}**

}

void **Spisok::OCHISTKA()**

{

**node \*q,\*q1;**// Рабочие указатели.

**q = phead;**

**q1 = (\*q).sled;** // Указатель q1 "опережает" указатель q.

do **{**

**q = q1;**

**q1 = (\*q1).sled;**

delete **q;**

**}**

while **(q1!=(\*phead).sled);**

}

# 59.Динамические структуры данных C++. Списки магазинного типа. Списки магазинного типа. Очереди. Формирование очереди. Добавление звена к очереди. Удаление звена из очереди.

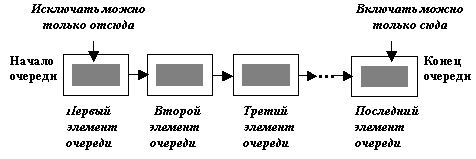
## Общая характеристика списков магазинного типа .

    Назовем ***списком магазинного типа*** линейный список, все звенья которого вставляются и удаляются только с одного или обоих концов списка. Списки магазинного типа подразделяются на ***очереди, стеки*** и ***деки***.

***Очередь*** - список магазинного типа, в котором все включения производятся на одном конце списка, а все исключения делаются на другом его конце.

    Очередь иногда называют ***циклической памятью*** или ***списком типа*** **FIFO** (**"First In - First Out"** - ***"первым включается - первым исключается"***).

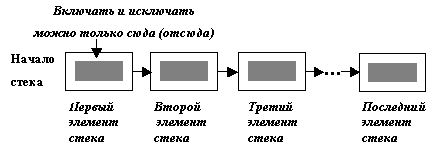
   Информация помещается в конец очереди и удаляется в момент, когда, наконец, достигает ее начала. Изобразим это схематически:

  
Рис.1. Очередь

***Стек*** - список магазинного типа, в котором все включения и исключения звеньев делаются в одном конце списка.

    Из стека мы всегда исключаем "младший" элемент из имеющихся в списке (тот элемент, который был включен позже других). Для очереди справедливо в точности противоположное правило: исключается всегда самый "старший" элемент; элементы "покидают" список в том порядке, в котором они в него вошли.

    Существуют и другие названия стека: ***магазин, список типа*** **LIFO** (**"Last In - First Out"** - ***"последним включается - первым исключается"***); ***"пуш-даун" список*** (**"push-down"**), ***реверсивная память, гнездовая память***.

  
Рис.2. Стек

***Дек*** (**"Double-Ended Queue"** - ***"двухсторонняя очередь"***) - список магазинного типа, в котором все включения и исключения звеньев делаются на обоих концах списка.

    Очевидно, что дек обладает большей общностью, чем стек или очередь; он имеет некоторые общие свойства с колодой игральных карт.

## Очереди

    Отметим, что очередь - динамическая структура данных, так как с течением времени длина очереди (количество входящих в нее звеньев) изменяется.

Пример: похоже на очередь в магазине

## Формирования очереди.

    Запишем алгоритм формирования очереди:

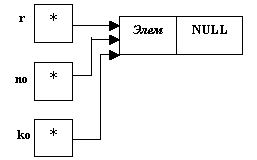
1. r = new (node);

(\*r).elem = *Элем*;

(\*r).sled = NULL;

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris16_1.jpg  
Рис.1. Первый элемент в очереди

1. no = r; ko = r;

  
Рис.2. Настройка указателей начала и конца очереди

1. r = new (node);

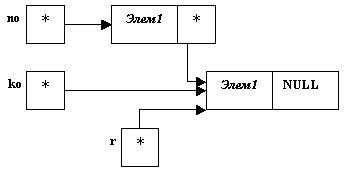
(\*r).elem = *Элем1*;

(\*r).sled = NULL;

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris16_3.jpg  
Рис.3. Создали новый элемент очереди

1. (\*ko).sled = r;

ko = r;

  
Рис.4. Настроили указатель на конец очереди

    Таким образом, очередь уже содержит два звена и, нам думается, что процесс построения понятен.

    Представим описанный алгоритм в виде функции на языке **C++**:

void POSTROENIE (node \*\*no, node \*\*ko)

// Построение очереди на базе однонаправленного

// линейного списка без заглавного звена:

// \*no - указатель на начало очереди,

// \*ko - указатель на конец очереди.

{

node \*r;

int el;

cin>>el;

if (el!=0)

{

r = new (node);

(\*r).elem = el;

(\*r).sled = NULL;

\*no = r;

\*ko = r;

cin>> el;

while (el!=0)

{ r = new (node);

(\*r).elem = el; (\*r).sled = NULL;

(\*\*ko).sled = r; \*ko = r; cin>>el;}

}

else

{ r = NULL; \*no = r; \*ko = r;}

}

    Тут же приведем функцию для просмотра содержимого очереди:

void VYVOD (node \*\*no, node \*\*ko)

// Вывод содержимого очереди.

// \*no - указатель на начало очереди,

// \*ko - указатель на конец очереди.

{

node \*r;

cout<< "Очередь: "; r = \*no;

while (r!=NULL)

{ cout<<(\*r).elem<<" "; r = (\*r).sled; }

cout<<endl;

}

## Алгоритм добавления звена к очереди.

    Приступим к описанию алгоритма добавления звена к очереди. Напомним, что звено добавляется ***в конец очереди***.

  
Рис.1. Исходная очередь

Вначале построим добавляемое звено:

1. r = new (node);

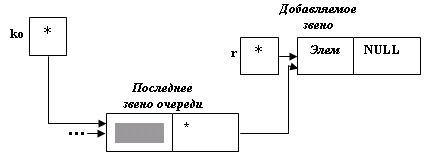
(\*r).elem = *Элем*;

(\*r).sled = NULL;

  
Рис.1. Заполнение добавляемого звена

Присоединяем звено к очереди:

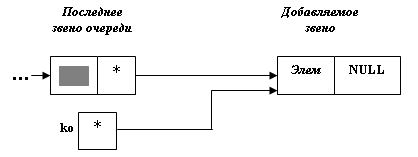
1. (\*ko).sled = r;

  
Рис.2. Результат присоединения звена

"Настраиваем" указатель **ko** на конец очереди:

1. ko = r;

    Изобразим результат добавления звена:

  
Рис.3. "Настройка" указателя

    В результате добавляемое звено стало последним звеном очереди.

    Оформим алгоритм в виде функции на языке **C++**:

void DOBAVLENIE (node \*\*no, node \*\*ko,int el)

// Добавление звена с информационным полем el

//\* к очереди, определенной указателями \*no и \*ko.

{

node \*r;

r = new (node);

(\*r).elem = el; (\*r).sled = NULL;

if (\*no!=NULL)

{ (\*\*ko).sled = r; \*ko = r;}

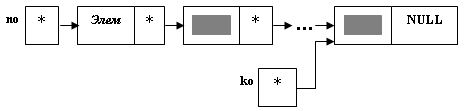
else

{ \*no = r; \*ko = r;}

}

## Алгоритм удаления звена из очереди.

    Пусть очередь не пуста (**no!=NULL**). Изобразим ее схематически:

  
Рис.1. Очередь

    Приступим к удалению звена. Напомним, что звено удаляется из очереди из ее начала.

Сохраним удаляемый элемент:

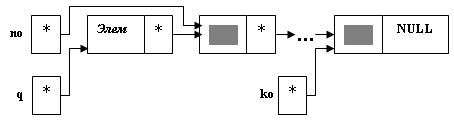
1. klad = (\*no).elem;

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris18_2.jpg  
Рис.2. Сохранение удаляемого элемента

Сохраним указатель на удаляемый элемент и "перенастроим" указатель на начало очереди:

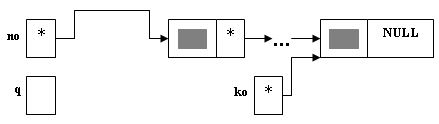
1. q = no;

no = (\*no).sled;

  
Рис.3. "Перенастройка" указателя на начало очереди

Теперь необходимо включить в список свободной памяти удаленное из очереди звено с помощью вызова функции:

1. delete q;

  
Рис.4. Возврат памяти в кучу

    Запишем приведенную схему в виде функции на языке **C++**:

void YDALENIE (node \*\*no, node \*\*ko, int klad)

// Удаление звена из очереди, определенной указателями \*no

// и \*ko. Значение информационного поля удаленного звена

// сохраняется в параметре klad.

{

node \*q;

if (\*no==NULL)

cout<< "Удалить нельзя, так как очередь пуста!\n";

else

{ \*klad = (\*\*no).elem; q = \*no; \*no = (\*\*no).sled; delete q;}

}

# 60.Динамические структуры данных C++. Стек. Формирование стека. Включение звена в стек. Удаление звена из стека.

## Стек. Формирование стека.

С***тек*** - это специально организованная память, выборка и занесение данных в которую подчиняется дисциплине **LIFO** (***"последним вошел - первым обслужен"***).

***Стек на базе линейного однонаправленного списка*** - линейный однонаправленный список, в котором все включения и исключения звеньев делаются в одном (выбранном нами) конце списка.

    Опишем алгоритм помещения в стек информации.

1. Вначале стек пуст:

**stk = NULL;**

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris19_1.jpg

Рис.1. Стек пуст

1. Содержимое стека будем вводить с клавиатуры, ввод заканчивается нулем:

**cin>>Элем;**

**t =** new **(node);**

**(\*t).elem = Элем; (\*t).sled = stk;**

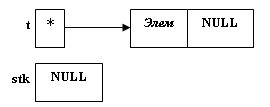


Рис.2. Новый элемент

1. "Настраиваем" указатель стека на созданный элемент:

**stk = t;**

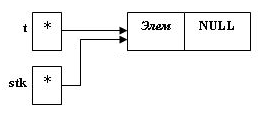


Рис.3. "Настройка" указателя стека

1. В результате в стек будет помещено первое звено:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris19_4.jpg

Рис.4. Первый элемент в стеке

1. Продолжим заполнение стека:

**cin>>Элем1;**

**t =** new **(node);**

**(\*t).elem = Элем1;**

**(\*t).sled = stk;**

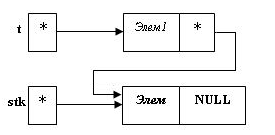


Рис.5. Размещение в стеке второго элемента

1. "Настраиваем" указатель стека на созданный элемент:

**stk = t;**

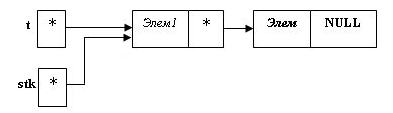


Рис.6. "Настройка" указателя стека

1. Теперь стек содержит уже два звена:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris19_7.jpg

Рис.7. В стеке два элемента

Продолжение процесса построения стека достаточно очевидно.

Оформим алгоритм в виде функции языка C++:

void **POSTROENIE (node \*\*stk)**

//Построение стека, заданного указателем \*stk с клавиатуры.

{

**node \*t;**

int **el;**

**\*stk = NULL;**

**cin>>el;**

while **(el!=0)**

**{**

**t =** new **(node);**

**(\*t).elem = el; (\*t).sled = \*stk; \*stk = t;**

**cin>>el;**

**}**

}

## Алгоритм включения элемента в стек.

    Опишем алгоритм включения звена с информационным полем ***Элем*** в стек.

1. Исходное состояние стека:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris20_1.jpg

Рис.1. Исходное состояние стека

1. Создаем новый элемент:

**q =** new **(node);**

**(\*q).elem = Элем;**

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris20_2.jpg

Рис.2. Новый элемент

1. Включаем элемент в начало стека:

**(\*q).sled = stk;**

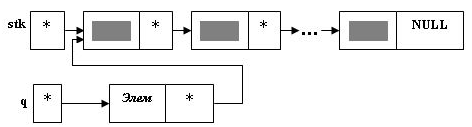


Рис.3. Включение элемента в стек

1. "Настроим" указатель вершины стека:

**stk = q;**

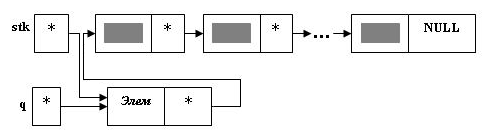


Рис.4. "Настройка" указателя вершины стека

Функция имеет вид:

void **W\_S (node \*\*stk,** int **el)**

//Включение звена с элементом el в стек,

// заданный указателем \*stk.

{

**node \*q;**

**q =** new **(node);**

**(\*q).elem = el; (\*q).sled = \*stk; \*stk = q;**

}

Заметим, что функцию [**POSTROENIE()**](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0019.html#1) можно переписать следующим образом, если воспользоваться функцией W\_S():

void **POSTROENIE (node \*\*stk)**

// Построение стека, заданного указателем \*stk с клавиатуры.

{

int **el;**

**\*stk = NULL; cin>>el;**

while **(el!=0)**

**{ W\_S (stk,el); cin>>el;}**

}

## Алгоритм удаления звена из стека.

    Перед удалением звена из стека проверяем, пуст ли стек.

Пусть стек не пуст.

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris21_1.jpg

Рис.1. Исходный стек

Тогда спокойно приступаем к удалению.

1. Сохраняем удаляемый элемент:

**klad = (\*stk).elem;**

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris21_2.jpg

Рис.2. Сохранение удаляемого элемента

1. "Перенастраиваем" указатель стека и сохраняем адрес удаляемого элемента:

**q = stk; stk = (\*stk).sled;**

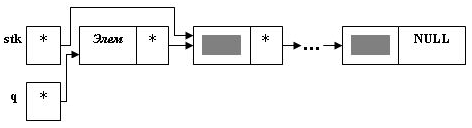


Рис.3. "Перенастройка" указателя стека

1. Возвращаем память в кучу:

delete **q;**

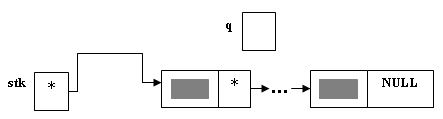


Рис.4. Возврат памяти в кучу

Запишем полученный алгоритм в виде функции:

void **YDALENIE (node \*\*stk,** int **klad)**

// Удаление звена из стека, заданного указателем \*stk, и

// помещение значения информационного поля удаленного звена

// в параметр klad.

{

**node \*q;**

if **(\*stk==NULL) cout<<**"Стек пуст!\n";

else

**{ \*klad = (\*\*stk).elem; q = \*stk;**

**\*stk = (\*\*stk).sled;** delete **q;}**

}

# 61.Динамические структуры данных C++. Дек.

**Дек ("двухсторонняя очередь")** на базе однонаправленного линейного списка - список магазинного типа на базе однонаправленного линейного списка, в котором все **включения и исключения звеньев делаются на обеих концах очереди**.

Мы приведем лишь демонстрационный пример.

**Пример**. Формирование дека, просмотр его содержимого, добавление элемента к деку и удаление элемента из дека.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **elem;**

**node \*sled;**

};

class **Spisok**

{

private:

**node \*ld,\*rd;**

int **el\_left,el\_right;**

public:

void **POSTROENIE ();**

void **VYVOD ();**

void **VSTAV1 (**int);

void **VSTAV2 (**int);

int **SetElLeft() {**return **el\_left;}**

int **SetElRight() {**return **el\_right;}**

void **YDALE1 ();**

void **YDALE2 ();**

void **OCHISTKA();**

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

int **el;**

**A.POSTROENIE (); A.VYVOD ();**

**cout<<**"Добавим звено справа.\n";

**cout<<**"Введите элемент добавляемого звена: ";

**cin>>el;**

**A.VSTAV1 (el); A.VYVOD ();**

**cout<<**"Добавим звено слева.\n";

**cout<<**"Введите элемент добавляемого звена: ";

**cin>>el;**

**A.VSTAV2 (el); A.VYVOD ();**

**cout<<**"Удалим звено справа.\n";

**A.YDALE1 (); A.VYVOD (); cout<<A.SetElRight()<<endl;**

**cout<<**"Удалим зввено слева.\n";

**A.YDALE2 (); A.VYVOD (); cout<<A.SetElLeft()<<endl;**

**A.OCHISTKA();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **Spisok::POSTROENIE ()**

//Построение дека :

// ld - указатель на левый конец дека,

// rd - Указатель на правый конец дека.

{

**node \*k;**

int **el;**

**cout<<**"Вводите содержимое звеньев дека: \n";

**cin>>el;**

if **(el!=0)**

**{**

**k =** new **(node);**

**(\*k).elem = el; (\*k).sled = NULL;**

**ld = k; rd = k; cin>>el;**

while **(el!=0)**

**{VSTAV1 (el); cin>>el;}**

**}**

else

**{rd = NULL; ld = NULL;}**

}

void **Spisok::VYVOD ()**

//Вывод содержимого дека:

// ld - указатель на левый конец дека.

{

**node \*k;**

**k = ld; cout<<**"Дек: ";

while **(k!=NULL)**

**{cout<<(\*k).elem<<**" "; k = (\*k).sled;}

**cout<<endl;**

}

void **Spisok::VSTAV1 (**int **el)**

// Помещение звена, содержащего элемент el, в дек справа.

// ld - указатель на левый конец дека,

// rd - указатель на правый конец дека.

{

**node \*k;**

**k =** new **(node);**

**(\*k).elem = el; (\*k).sled = NULL;**

if **(rd!=NULL)**

**{(\*rd).sled = k; rd = k;}**

else

**{rd = k; ld = k;}**

}

void **Spisok::VSTAV2 (**int **el)**

//Помещение звена, содержащего элемент el, в дек слева.

// ld - указатель на левый конец дека,

// rd - указатель на правый конец дека.

{

**node \*k;**

**k =** new **(node);**

**(\*k).elem = el; (\*k).sled = ld;**

if **(ld!=NULL) ld = k;**

else **{ld = k; rd = k;}**

}

void **Spisok::YDALE1 ()**

//Удаление звена из дека справа

//с сохранением удаляемого звена в переменной el\_right.

// ld - указатель на левый конец дека,

// rd - указатель на правый конец дека.

{

**node \*z;**

**node \*k;**

if **(rd==ld)**

**{**

**el\_right = (\*rd).elem;** delete **rd;**

**ld = rd = NULL; cout<<**"Дек пуст!\n";

**}**

else

**{**

**z = ld; k = (\*ld).sled;**

while **(k!=rd)**

**{z = k; k = (\*k).sled;}**

**el\_right = (\*rd).elem; (\*z).sled = NULL;** delete **rd;**

**rd = z;**

**}**

}

void **Spisok::YDALE2 ()**

// Удаление звена из дека слева

// с сохранением удаляемого звена в переменной el\_left.

// ld - указатель на левый конец дека,

// rd - указатель на правый конец дека.

{

**node \*q;**

if **(ld!=NULL)**

**{**

**el\_left = (\*ld).elem; q = ld;**

**ld = (\*ld).sled;** delete **q;**

**}**

else **cout<<**"Дек пуст!\n";

}

void **Spisok::OCHISTKA()**

{

**node \*k,\*q;**

**k = ld;**

if **(k!=NULL)**

**{**

**q = (\*k).sled;**

while **(q!=NULL)**

**{**delete **k; k = q; q=(\*k).sled;}**

delete **k;**

**}**

}

# 63. Динамические структуры данных C++. Вставка звена в двунаправленный список (1-й случай). Вставка звена в двунаправленный список (2-й случай). Удаление звена из двунаправленного списка. Указатель на удаляемое звено (1-й случай). Удаление звена из двунаправленного списка. После звена (2-й случай).

Вначале рассмотрим **алгоритм вставки звена после звена, на которое указывает ссылка Res**. Пусть ссылка **Res** указывает на звено, после которого будет производиться вставка нового звена. Изобразим это схематически:

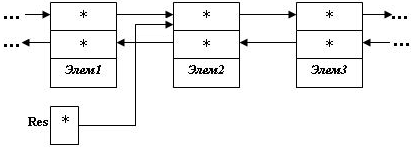


Рис.1. "Начальная позиция"

Проверим, является ли звено, на которое указывает ссылка **Res**, последним в списке. Это осуществляется путем анализа значения операции отношения **(\*Res).sled != NULL**.

Пусть звено, на которое указывает **Res, не является последним**.

1. В **heap**-области резервируем место для нового динамического объекта, а в информационное поле этого объекта помещаем значение информационного поля звена, которое желательно вставить в двунаправленный список:

**q =** new(node);

**(\*q).elem = Элем;**

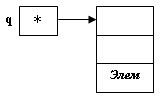


Рис.2. Вставляемый элемент

1. "Настраиваем" указатели вставляемого элемента:

**(\*q).sled = (\*Res).sled;**

**(\*q).pred = (\*\*Res.sled).pred;**

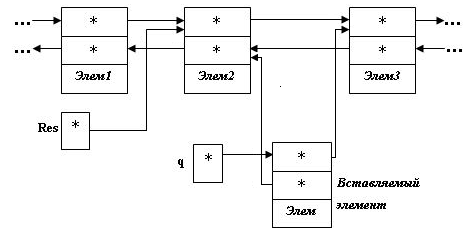


Рис.3. "Настройка" указателей вставляемого элемента

1. "Настраиваем" указатели списка на вставляемый элемент:

**(\*\*Res.sled).pred = q;**

**(\*Res).sled = q;**

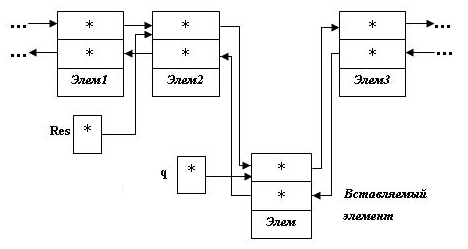


Рис.4. "Настройка" указателей элементов списка на вставляемый элемент

1. В итоге получим:

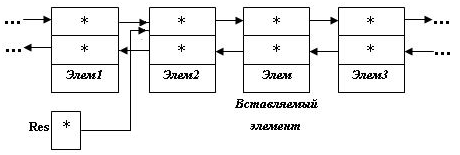


Рис.5. Результат вставки элемента

Если элемент, после которого размещается новый элемент, окажется **последним в двунаправленном списке**, тогда алгоритм изменится следующим образом:

1. Создадим новый элемент и заполним его поля:

**q =** new(node);

**(\*q).elem = Элем;**

**(\*q).sled = NULL;**

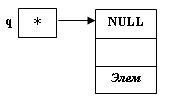


Рис.6. Вставляемый элемент

1. Осуществим "настройку" указателей:

**(\*q).pred = Res; \*ksp = q;**

**(\*Res).sled = q;**

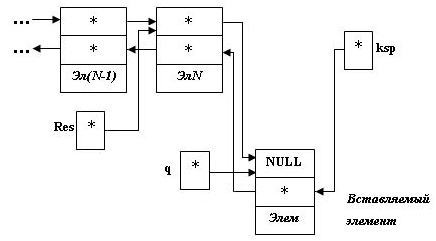


Рис.7. "Настройка" указателей

Оформим алгоритм вставки звена в список в виде функции:

void **InsAfter (**int **el, node \*\*nsp, node \*\*ksp, node \*Res)**

// Вставление звена с информационным полем el в

// двунаправленный список, заданный указателями

// \*nsp и \*ksp, после звена, на которое указывает Res.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el;**

if **((\*Res).sled!=NULL)**

**{**

**(\*q).sled = (\*Res).sled;**

**(\*q).pred = (\*(\*Res).sled).pred; (\*(\*Res).sled).pred = q; (\*Res).sled = q;**

**}**

else

**{ (\*q).sled = NULL; (\*q).pred = Res; \*ksp = q; (\*Res).sled = q;}**

}

Рассмотрим **алгоритм вставки звена перед звеном, на которое указывает ссылка Res**. Опишем его с помощью схем "до и после" Д.Кнута:

1. Создаем элемент:

**q =** new(node);

**(\*q).elem = Элем;**

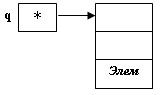


Рис.1. Создание элемента

1. "Настраиваем" его указатели:

**(\*q).sled = (\*(\*Res).pred).sled;**

**(\*q).pred = (\*Res).pred;**

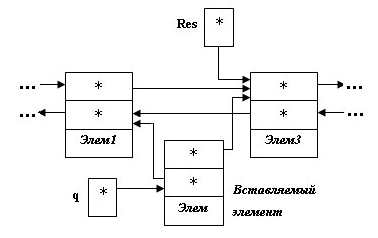


Рис.2. "Настройка" указателей созданного элемента

1. "Настраиваем" указатели элементов списка на вставляемый элемент:

**(\*(\*Res).pred).sled = q;**

**(\*Res).pred = q;**

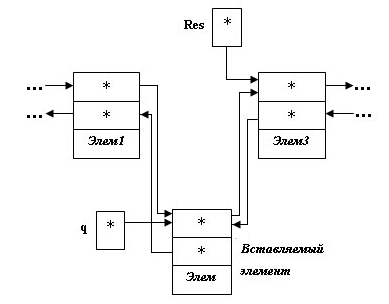


Рис.3. "Настройка" указателей элементов списка на созданный элемент

1. В итоге получим:

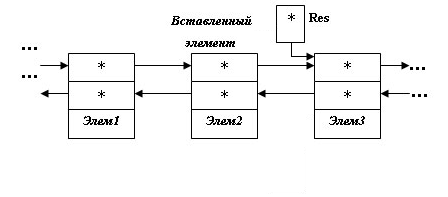


Рис.4. Итоговый результат

Оформим алгоритм помещения звена в список в виде функции:

void **InsBefore (**int **el, node \*\*nsp, node \*\*ksp, node \*Res)**

// Вставление звена с информационным полем el в двуна-

// правленный список, заданный указателями \*nsp и \*ksp,

// перед звеном, на которое указывает Res.

{

**node \*q;**

**q =** new **(node);**

**(\*q).elem = el;**

**(\*q).sled = (\*(\*Res).pred).sled;**

**(\*q).pred = (\*Res).pred;**

**(\*(\*Res).pred).sled = q; (\*Res).pred = q;**

}

рассмотрим **первый случай удаления звена**.

Пусть нам **известен указатель Res на удаляемое звено**. Возможны два случая:

* удаляемый элемент является последним в двунаправленном списке,
* удаляемый элемент не является последним в двунаправленном списке.

Вначале рассмотрим ситуацию, когда он **не последний**, то есть **(\*Res).sled != NULL**. Изобразим схему "до", соответствующую этому случаю:

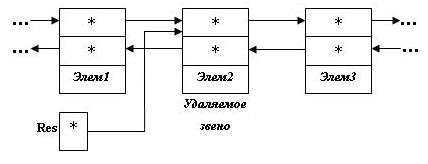


Рис.1. "Начальная" позиция

1. Исключим звено из списка, "перенастроив" указатели предыдущего и последующего звеньев:

**(\*(\*Res).sled).pred = (\*Res).pred;**

**(\*(\*Res).pred).sled = (\*Res).sled;**

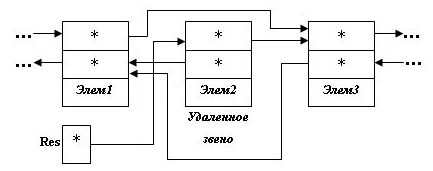


Рис.2. "Перенастройка" указателей

1. Осталось только удалить звено, на которое указывает Res, из heap-области:

delete **Res;**

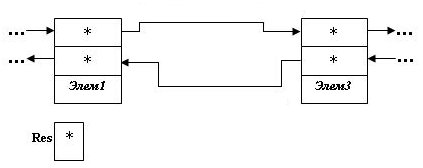


Рис.3. Освобождение памяти

Пусть теперь **удаляемый элемент оказался последним**.

Взгляните на схему "до":

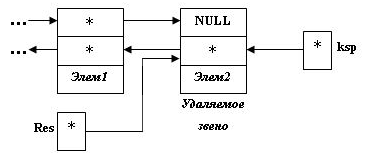


Рис.4. "Начальная" позиция

1. "Настроим" указатели предпоследнего элемента и конца списка:

**(\*(\*Res).pred).sled = NULL;**

**ksp = (\*ksp).pred;**

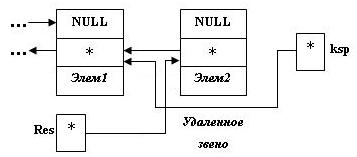


Рис.5. "Настройка" указателей

1. Теперь удалим из кучи звено, на которое указывает Res:

delete **Res;**

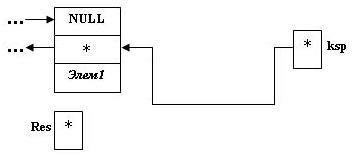


Рис.6. Возврат памяти в кучу

Запишем функцию для удаления звена, на которое указывает ссылка:

void **Delete (node \*\*nsp, node \*\*ksp, node \*Res)**

// Удаление звена из двунаправленного списка.

// \*nsp - указатель на начало списка,

// \*ksp - указатель на конец списка,

// Res - указатель на удаляемое звено.

{

if **((\*Res).sled!=NULL)**

**{ (\*(\*Res).sled).pred = (\*Res).pred;**

**(\*(\*Res).pred).sled = (\*Res).sled;** delete **Res;}**

else

**{ (\*(\*Res).pred).sled = NULL; \*ksp = (\*\*ksp).pred;** delete **Res;}**

}

**алгоритм удаления звена из двунаправленного списка**.

Построим теперь **алгоритм для удаления звена, стоящего после звена, на которое указывает ссылка Res**. Пусть удаляемое звено не является последним, то есть **(\*(\*Res).sled).sled!=NULL**. Изобразим этот факт на схеме "до":

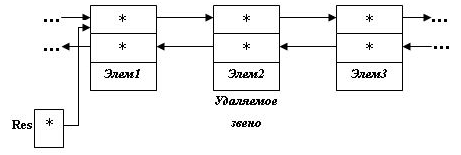


Рис.1. "Начальная" позиция

1. Исключим звено из списка, "перенастроив" указатели предыдущего и последующего звеньев, а также сохраним адрес удаляемого звена:

**q = (\*Res).sled;**

**(\*(\*(\*Res).sled).sled).pred = Res;**

**(\*Res).sled = (\*(\*Res).sled).sled;**

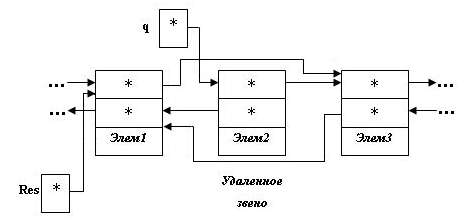


Рис.2. "Перенастройка" указателей

1. Осталось только удалить звено, на которое указывает **q**, из **heap**-области:

delete **q;**

Если же удаляемый элемент является **последним** в двунаправленном списке,

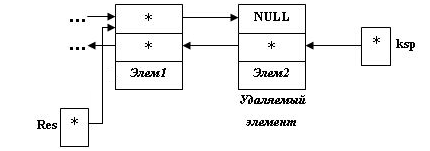


Рис.3. "Начальная" позиция

то алгоритм становится тривиальным:

1. "Настроим" указатели предпоследнего элемента и конца списка:

**q = (\*Res).sled; (\*Res).sled = NULL; ksp = (\*ksp).pred;**

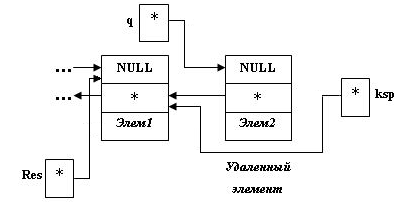


Рис.4. "Настройка" указателей

1. Теперь удалим из кучи звено, на которое указывает **q:**

delete **q;**

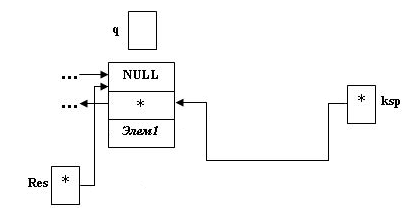


Рис.5. Возврат памяти в кучу

Функция для удаления звена, стоящего после звена, на которое указывает ссылка **Res**, имеет вид:

void **DelAfter (node \*\*nsp, node \*\*ksp, node \*Res)**

// Удаление звена из двунаправленного списка.

// \*nsp - указатель на начало списка,

// \*ksp - указатель на конец списка,

// Res - указатель на звено, предыдущее удаляемому.

{

**node \*q;**

if **((\*Res).sled==NULL)**

**cout<<**"Вы хотите удалить звено за последним звеном!\n";

else

if **((\*(\*Res).sled).sled!=NULL)**

**{**

**q = (\*Res).sled;**

**(\*(\*(\*Res).sled).sled).pred = Res;**

**(\*Res).sled = (\*(\*Res).sled).sled;**

delete **q;**

**}**

else

**{ q = (\*Res).sled; (\*Res).sled = NULL;**

**\*ksp = (\*\*ksp).pred;** delete **q; }**

}

# 64. Динамические структуры данных C++. Двунаправленные кольцевые списки.

В программировании двунаправленные списки часто преобразовывают следующим образом: "обычный" линейный двунаправленный список замыкают в своеобразное "кольцо": при движении по списку **в прямом направлении** можно от последнего звена переходить к звену, следующему прямо за заглавным звеном, а при движении **в обратную сторону** - от заглавного звена переходить сразу к последнему звену. В связи с этим мы будем называть двунаправленные списки подобного типа **кольцевыми двунаправленными списками (двунаправленными кольцами** или просто **кольцами**).

Подобная организация списка упрощает процедуру поиска или перебора звеньев с любого места списка с автоматическим переходом от конца к началу или наоборот.

Мы приведем два варианта структуры кольцевых двунаправленных списков.

Рассмотрим первый вариант структуры кольцевого списка, которую мы будем называть **кольцевым** **двунаправленным списком с удаленным заглавным звеном**. Уже само название структуры говорит о том, что заглавное звено двунаправленного списка в кольцо не включается.

Взгляните на приведенную ниже схему:

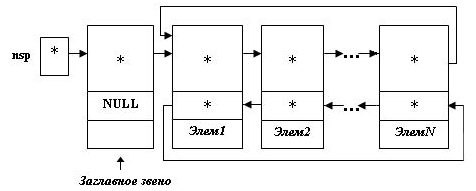


Рис.1. Кольцевой двунаправленный список с удаленным заглавным звеном

Пустое кольцо можно представить следующим образом:

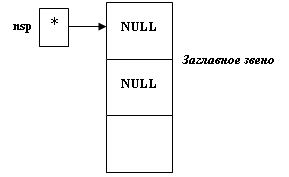


Рис.2. Пустой кольцевой двунаправленный список

Проще всего построить кольцевой двунаправленный список с удаленным заглавным звеном с использованием алгоритма построения двунаправленного списка с заглавным звеном и последующего "замыкания" кольца. Это реализовано в приведенной ниже функции:

void **BuiltRing (node \*\*nsp)**

// Построение двунаправленного кольцевого списка nsp

// с удаленным заглавным звеном.

// \*nsp - указатель на заглавное звено списка.

{

**node \*r;**

int **el;**

// Построим заглавное звено будущего кольцевого списка.

**\*nsp =** new(node);

**r = \*nsp; (\*\*nsp).pred = NULL; (\*\*nsp).sled = NULL;**

**cout<<**"Вводите элементы звеньев списка: \n";

**cin>>el;**

while **(el!=0)**

**{ (\*r).sled =** new(node); (\*((\*r).sled)).pred = r;

**r = (\*r).sled; (\*r).sled = NULL; (\*r).elem = el;**

**cin>>el; }**

// Образуем кольцевой список с удаленным заглавным звеном.

if **((\*\*nsp).sled!=NULL)**

**{(\*((\*\*nsp).sled)).pred = r; (\*r).sled = (\*\*nsp).sled;}**

else **cout<<**"Кольцевой список пуст!\n";

}

Обход кольцевого списка реализовать несложно. Приведем, например, функцию, позволяющую обойти кольцо "по часовой стрелке":

void **VyvodLeftRight (node \*\*nsp)**

// Вывод содержимого двунаправленного кольцевого списка

// с удаленным заглавным звеном "по часовой стрелке".

// \*nsp - указатель на заглавное звено списка.

{

**node \*r;**

**cout<<**"Кольцевой список: ";

if **((\*\*nsp).sled!=NULL)**

**{ cout<<(\*((\*\*nsp).sled)).elem)<<**" ";

**r = (\*((\*\*nsp).sled)).sled;**

while **(r!=(\*\*nsp).sled)**

**{ cout<<(\*r).elem)<<**" "; r = (\*r).sled; }

**cout<<endl;}**

else **cout<<**"пуст!";

}

# 65. Динамические структуры данных C++. Деки на базе двунаправленных списков. Формирование дека и его просмотр. Добавление звена в начало дека. Добавление звена в конец дека. Удаление звена из дека слева. Удаление звена из дека справа.

Мы будем моделировать дек с помощью **двунаправленного списка без заглавного звена**.

Вначале опишем алгоритм формирования двунаправленного списка без заглавного звена из двунаправленного списка с заглавным звеном, изображенного ниже:

1. "Начальная" позиция:

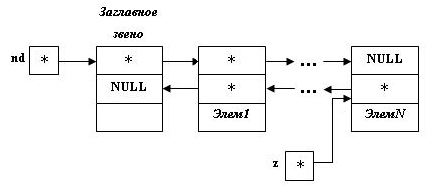


Рис.1. "Начальная" позиция

1. Исключим заглавное звено из списка:

**q = nd;**

**nd = (\*nd).sled;**

**(\*nd).pred = NULL;**

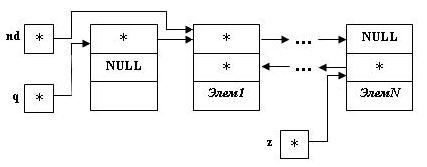


Рис.2. Исключение заглавного звена

1. Возврат памяти в кучу и "настройка" указателя на конец дека:

delete **q;**

**kd = z;**

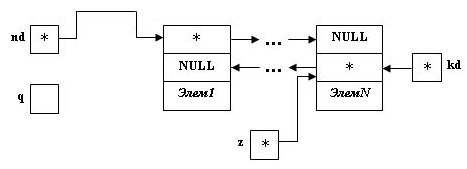


Рис.3. "Настройка" указателя и возврат памяти в кучу

Очевидно, что суть алгоритма состоит в "отбрасывании" заглавного звена двунаправленного списка.

Оформим данный алгоритм в виде функции:

void **BuiltDeck (node \*\*nd, node \*\*kd)**

// Построение дека на базе двунаправленного

// списка с заглавным звеном.

// \*nd - указатель на начало дека.

// \*kd - указатель на конец дека.

{

**node \*q, \*z;**

int **el;**

// Построение заглавного звена.

**\*nd =** new(node);

**z = \*nd;**

**(\*\*nd).pred = (\*\*nd).sled = NULL;**

**cout<<**"Введите последовательность: \n";

**cin>>el;**

while **(el!=0)**

**{ (\*z).sled =** new(node);

**(\*((\*z).sled)).pred = z; z = (\*z).sled;**

**(\*z).sled = NULL; (\*z).elem = el; cin>>el; }**

if **((\*\*nd).sled!=NULL)**

**{ q = \*nd; \*nd = (\*\*nd).sled; (\*\*nd).pred = NULL; \*kd = z;** delete **q; }**

else

**{** delete **\*nd; \*nd = \*kd = NULL;}**

}

Со следующего шага мы начнем рассматривать основные операции над деками.

### Добавление звена в начало дека

На этом шаге мы рассмотрим **алгоритм добавления звена в начало дека**.

Алгоритм добавления звена в начало дека заключается в создании из кучи нового элемента и настройки указателя начала дека на новый элемент:

**q =** new(node);

**(\*q).elem = Элем;**

**(\*q).sled = nd; (\*q).pred = NULL;**

**(\*nd).pred = q; nd = q;**

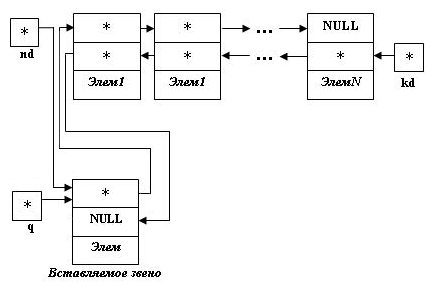


Рис.1. Вставка звена в начало дека

Приведем текст функции:

void **InsLeft (node \*\*nd, node \*\*kd,**int **el)**

// Вставка звена, содержащего элемент el, в дек слева.

// \*nd - указатель на начало дека.

// \*kd - указатель на конец дека.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el;**

if **(\*nd==NULL)**

**{** // Если дек пуст, то...

**\*nd = q; (\*q).sled = (\*q).pred = NULL; \*kd = q;}**

else

**{ (\*q).sled = \*nd; (\*q).pred = NULL; (\*\*nd).pred = q; \*nd = q;}**

}

На следующем шаге мы рассмотрим **алгоритм добавления звена в дек справа**.

### Добавление звена в конец дека

На этом шаге мы рассмотрим алгоритм добавления звена в конец дека.

Алгоритм добавления звена в конец дека заключается в выделении памяти под новый элемент и включение его в конец дека:

**q =** new(node);

**(\*q).elem = Элем;**

**(\*q).sled = NULL;**

**(\*q).pred = \*kd;**

**(\*kd).sled = q;**

**\*kd = q;**

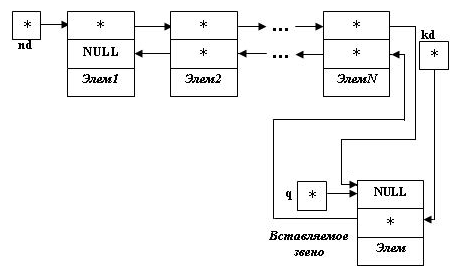


Рис.1. Вставка звена в конец дека

Приведем текст функции:

void **InsRight (node \*\*nd, node \*\*kd,**int **el)**

// Добавление звена, содержащего элемент el, в дек справа.

// \*nd - указатель на начало дека.

// \*kd - указатель на конец дека.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el;**

if **(\*kd==NULL)**

**{**// Если дек пуст, то...

**\*nd = q; (\*q).sled = (\*q).pred = NULL; \*kd = q;}**

else

**{ (\*q).sled = NULL; (\*q).pred = \*kd; (\*\*kd).sled = q; \*kd = q;}**

}

Со следующего шага мы начнем рассматривать **алгоритмы удаления звена из дека**.

### Удаление звена из дека слева

На этом шаге мы рассмотрим **первый алгоритм удаления звена дека**.

Рассмотрим **алгоритм удаления звена слева**.

1. Пусть уже построен дек на базе двунаправленного списка без заглавного звена:

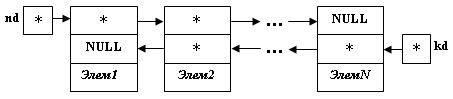


Рис.1. "Начальная" позиция

1. Сохраним адрес удаляемого элемента и "настроим" указатель начала дека:

**q = nd;**

**nd = (\*nd).sled;**

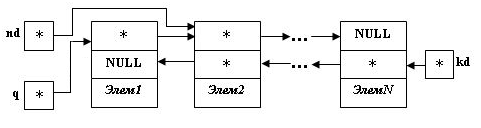


Рис.2. "Настройка" указателей

1. Возвратим память в кучу:

delete **q;**

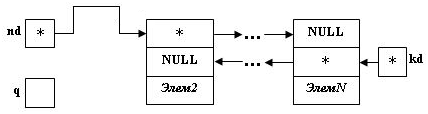


Рис.3. Возврат памяти в кучу

Запишем по описанному алгоритму функцию на языке C++:

void **DelLeft (node \*\*nd, node \*\*kd,**int **\*el)**

// Удаление звена из дека слева с помещением элемента

// удаляемого звена в переменную el.

// \*nd - указатель на начало дека.

// \*kd - указатель на конец дека.

{

**node \*q;**

if **((\*\*nd).sled!=NULL)**

**{ q = \*nd; \*el =(\*q).elem; \*nd = (\*\*nd).sled; (\*\*nd).pred = NULL;** delete **q;}**

else

**{** //В деке находится один элемент.

**q = \*nd; \*el =(\*q).elem; \*nd = \*kd = NULL;**

delete **q; cout<<**"Дек пуст!\n";}

}

На следующем шаге мы рассмотрим **алгоритм удаления дека справа**.

### Удаление звена из дека справа

На этом шаге мы рассмотрим **алгоритм удаления звена из дека справа**.

Рассмотрим **алгоритм удаления звена справа**. Пусть уже построен дек на базе двунаправленного списка без заглавного звена:

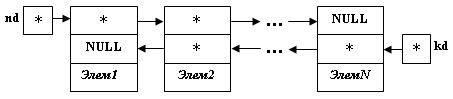


Рис.1. "Начальная" позиция

Приступим к пошаговому выполнению алгоритма:

1. Сохраним адрес удаляемого элемента и "настроим" указатель конца дека:

**q = kd;**

**kd = (\*kd).pred;**

**(\*kd).sled = NULL;**

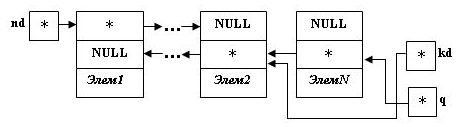


Рис.2. "Настройка" указателей

1. Возвратим память в кучу:

delete **q;**

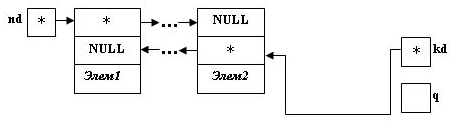


Рис.3. Возврат памяти в кучу

Запишем функцию на языке C++, реализующую рассмотренный алгоритм:

void **DelRight (node \*\*nd, node \*\*kd,**int **\*el)**

// Удаление звена из дека справа с помещением элемента

// удаляемого звена в переменную el.

// \*nd - указатель на начало дека.

// \*kd - указатель на конец дека.

{

**node \*q;**

if **((\*\*kd).pred!=NULL)**

**{ q = \*kd; \*el =(\*q).elem; \*kd = (\*\*kd).pred; (\*\*kd).sled = NULL;** delete **q;}**

else

**{** // В деке находится один элемент.

**q = \*kd; \*el =(\*q).elem; \*nd = \*kd = NULL;**

delete **q; cout<<**"Дек пуст!\n";}

}

# 66. Динамические структуры данных C++. Дерево. Бинарные деревья. Основная терминология. Бинарные деревья поиска. Построение бинарного дерева поиска (рекурсивный алгоритм). Анализ алгоpитма поиска с включениями. Дерево отрезков

**Дерево** — одна из наиболее широко распространённых структур данных в информатике, эмулирующая древовидную структуру в виде набора связанных узлов. Является связным графом, не содержащим циклы. Большинство источников также добавляют условие на то, что рёбра графа не должны быть ориентированными. В дополнение к этим трём ограничениям, в некоторых источниках указывается, что рёбра графа не должны быть взвешенными.

* *Корневой узел* — самый верхний узел дерева (узел 8 на примере).
* *Корень* — одна из вершин, по желанию наблюдателя.
* *лист*, *листовой* или *терминальный* *узел* — узел, не имеющий дочерних элементов
* *Внутренний узел* — любой *узел* дерева, имеющий *потомков*, и таким образом, не являющийся *листовым узлом*
* Дерево считается *ориентированным*, если в корень не заходит ни одно ребро.
* *Полный сцепленный ключ* — идентификатор записи, который образуется путём [конкатенации](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%BA%D0%B0%D1%82%D0%B5%D0%BD%D0%B0%D1%86%D0%B8%D1%8F) всех ключей экземпляров родительских записей (групп).

Бинарное дерево

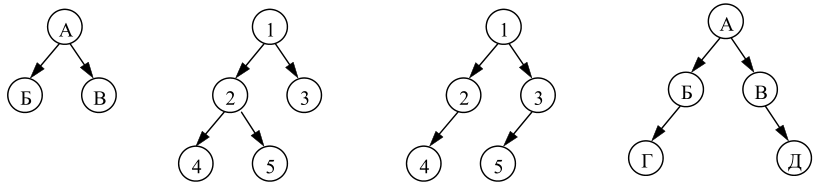
**Двоичным деревом** называется дерево, каждый узел которого имеет не более двух сыновей.

Двоичные деревья **упорядочены**, то есть различают левое и правое поддеревья.

**Строго двоичным деревом** называется дерево, у которого каждая внутренняя вершина имеет непустые левое и правое поддеревья.

Это означает, что в строго двоичном дереве нет вершин, у которых есть только одно поддерево. На рисунке даны деревья а) и б) являются строго двоичными, а в) и г) – нет.

**а) б) в) г)**



**Полным двоичным** деревом называется дерево, у которого все листья находятся на одном уровне и каждая внутренняя вершина имеет непустые левое и правое поддеревья.

Каждая вершина бинарного дерева является структурой, состоящей из четырех полей. Содержимым этих полей будут, соответственно:

* информационное поле (ключ вершины),
* служебное поле (их может быть несколько!),
* указатель на левое поддерево,
* указатель на правое поддерево.

Таким образом, каждая вершина бинарного дерева описываются на языке C++ следующим образом:

struct **node**

**{**

int **Key;** // Ключ вершины.

int **Count;** // Счетчик количества вершин с одинаковыми ключами.

**node \*Left;** // Указатель на "левого" сына.

**node \*Right;** // Указатель на "правого" сына.

**};**

Или второй вариант описания вершины

struct Node

{

int key; //полезные данные (ключ)

Node \*left, \*right; //указатели на сыновей

};

typedef Node \*PNode; // указатель на вершину

**нерекурсивный алгоритм построения дерева поиска**

**Tree = NULL;** //Построение пустого дерева.

Обозначим **p** - вспомогательный указатель на вершину дерева.

1. Пусть ключ первой, поступающей в дерево, вершины равен 100. Создаем первую вершину:

**p =** new(node);

**(\*p).Key = 100; (\*p).Count = 1;**

**(\*p).Left = NULL; (\*p).Right = NULL;**

**Tree = p;**

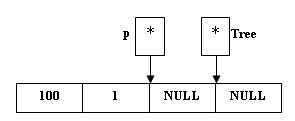


Рис.1. Создание первой вершины

1. Пусть ключ второй, поступающей в дерево, вершины равен 50. Выполняем следующие операции:

* вначале создаем новую вершину:

**p =** new(node);

**(\*p).Key = 50; (\*p).Count = 1;**

**(\*p).Left = NULL; (\*p).Right = NULL;**

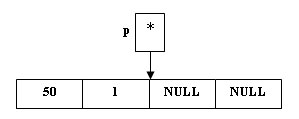


Рис.2. Создание новой вершины

* так как 100>50, то по определению бинарного дерева поиска мы должны сделать вновь поступившую вершину левым сыном корня дерева:

**(\*Tree).Left = p;**

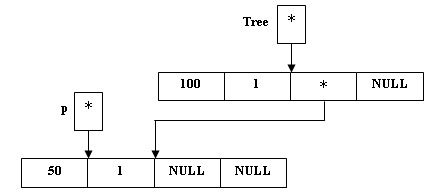


Рис.3. Размещение вершины в левом поддереве

1. Пусть ключ третьей вершины, поступающей в дерево, равен 200. Порядок наших действий:

* создаем новую вершину:

**p =** new(node);

**(\*p).Key = 200; (\*p).Count = 1;**

**(\*p).Left = NULL; (\*p).Right = NULL;**

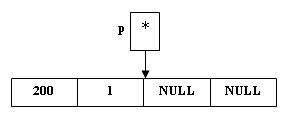


Рис.4. Создание новой вершины

* так как 200>100, то по определению бинарного дерева поиска, мы должны сделать вновь поступившую вершину правым сыном корня дерева:

**(\*Tree).Right = p;**

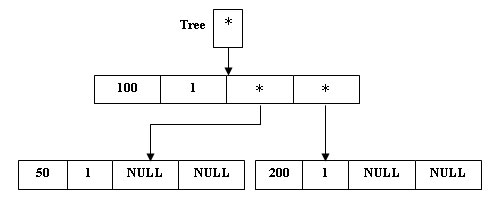


Рис.5. Размещение вершины в правом поддереве

Осталось определить, как же нам действовать в случае поступления, например, в дерево вновь вершины с ключом 100. Оказывается, что поле **Count** и применяется для учета повторяющихся ключей!

Точнее, в этом случае выполняется оператор присваивания **(\*Tree).Count = (\*Tree).Count + 1;**, результат выполнения которого изобразим на схеме:

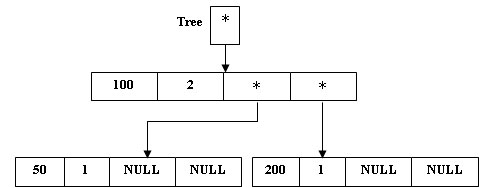


Рис.6. Размещение вершины с ранее встречавшимся ключом

**Анализ алгоpитма поиска с включениями**

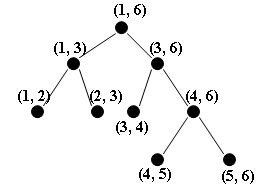
***Теоpема Хопкpофта-Ульмана***

Сpеднее число сpавнений, необходимых для вставки n случайных элементов в деpево поиска, пустое вначале, pавно O(nlog2n) для n>=1.

**Дерево отрезков**

**Деpево отpезков** - это стpуктуpа данных, созданная для pаботы с такими интеpвалами на числовой оси, концы котоpых пpинадлежат фиксиpованному множеству из N абсцисс. Поскольку множество абсцисс фиксиpовано, то деpево отpезков пpедставляет собой статическую стpуктуpу по отношению к этим абсциссам, т.е. стpуктуpу, на котоpой не pазpешены вставки и уделения абсцисс; кpоме того эти абсциссы можно ноpмализовать, заменяя каждую из них ее поpядковым номеpом пpи обходе их слева напpаво.

**Деpево отpезков** - это двоичное деpево с коpнем.



# ***67. Динамические структуры данных C++. Дерево. Обход бинарного дерева. Левосторонний обход бинарного дерева поиска. Концевой обход бинарного дерева поиска. Обратный обход бинарного дерева поиска.***

Одной из необходимых операций при работе с деревьями является **обход дерева**, во время которого надо посетить каждый узел по одному разу и (возможно) вывести информацию, содержащуюся в вершинах.

Пусть в результате обхода надо напечатать значения поля данных всех вершин в определенном порядке. Существуют **три варианта обхода**:

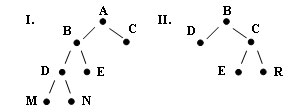
* **КЛП (корень – левое – правое)**: сначала посещается корень (выводится информация о нем), затем левое поддерево, а затем – правое;
* **ЛПК (левое – правое – корень)**: сначала посещается левое поддерево, затем правое, а затем – корень.
* **ЛКП (левое – корень – правое)**: сначала посещается левое поддерево, затем корень, а затем – правое;

Левостоонний обход

Для того, чтобы просмотреть информационные поля всех вершин построенного дерева, необходимо совершить его **обход (посетить каждую его вершину)**.

алгоритм **левостороннего обхода дерева**:

* **посетите корень дерева;**
* **обойдите левое поддерево;**
* **обойдите правое поддерево.**

 **A B D M N E C**

void **ObhodLeft (node \*\*w)**

// Левосторонний обход дерева.

// \*w - указатель на корень дерева.

{

if **(\*w!=NULL)** //пустое дерево?

**{ cout<<(\*\*w).Key<<**" "; вывод информации о корне

**ObhodLeft (&((\*\*w).Left));** //обход левого поддерева

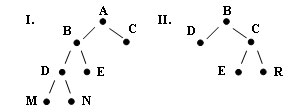
**ObhodLeft (&((\*\*w).Right)); }**//обход правого поддерева

}

Правосторонний обход

Существует **алгорим концевого обхода дерева**, который заключается в следующем:

* **обойдите левое поддерево;**
* **обойдите правое поддерево;**
* **посетите корень дерева.**

 **M N D E B C A**

void **ObhodEnd (node \*\*w)**

// Концевой обход дерева.

// \*w - указатель на корень дерева.

{

if **(\*w!=NULL)** //пустое дерево?

**{ ObhodEnd (&((\*\*w).Left));** //обход левого поддерева

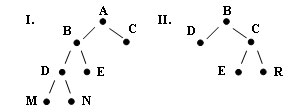
**ObhodEnd (&((\*\*w).Right));** //обход правого поддерева

**cout<<(\*\*w).Key<<**" ";} //вывод информации о корне

}

**алгорим обратного обхода дерева**

* **обойдите левое поддерево;**
* **посетите корень дерева;**
* **обойдите правое поддерево.**

 **M D N B E A C**

void **ObhodBack (node \*\*w)**

// Обратный обход бинарного дерева.

// \*w - указатель на корень дерева.

{

if **(\*w!=NULL)** //пустое дерево?

**{ ObhodBack (&((\*\*w).Left));** //обход левого поддерева

**cout<<(\*\*w).Key<<**" "; //вывод информации о корне

**ObhodBack (&((\*\*w).Right)); }**//обход правого поддерева

}

# **68. Динамические структуры данных C++. Дерево. Вывод бинарного дерева поиска. Пример программы с использованием бинарного дерева поиска**

void **Vyvod (node \*\*w,**int **l)**

// Изображение дерева w на экране дисплея.

// (рекурсивный алгоритм).

// \*w - указатель на корень дерева.

// l - "отступ" от левого края окна при выводе

// равен глубине вершины (расстояние от корня до этой вершины)

{

int **i;**

if **(\*w!=NULL)** //пустое дерево?

**{ Vyvod (&((\*\*w).Right),l+1);** //правое поддерево

for **(i=1; i<=l; i++) cout<<**" "; // расстояние от корня до вершины

**cout<<(\*\*w).Key<<endl;** //вывод информации о корне

**Vyvod (&((\*\*w).Left),l+1); }** //левое поддерево

}

Пример программы

реализуется построение бинарного дерева поиска, разнообразные его обходы, вывод дерева и определение его высоты, а также "очистка" дерева (рекурсивные алгоритмы).

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **Key;**

int **Count;**

**node \*Left;**

**node \*Right;**

};

class **TREE**

{

private:

**node \*Tree;** //Указатель на корень дерева.

void **Search (**int,node\*\*); //Поиск вершины с ключом int в дереве со вставкой

public:

**TREE() {Tree=NULL;}**

**node\*\* GetTree () {**return **&Tree;}** //Получение вершины дерева.

void **BuildTree ();** //Построение дерева

void **CleanTree (node \*\*);** //Очистка дерева

void **ObhodEnd (node \*\*);** //Концевой обход дерева

void **ObhodLeft (node \*\*);** //Левосторонний обход дерева

void **ObhodBack (node \*\*);** //Обратный обход дерева

void **Vyvod (node\*\*,**int);//Изображение дерева на экране дисплея

int **Height (node\*\*);** //Определение высоты бинарного дерева

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**TREE A;**

**A.BuildTree ();**

**cout<<**"\nВывод дерева:\n";

**A.Vyvod (A.GetTree(),0);**

**cout<<**"\nВысота дерева:"<<A.Height(A.GetTree())<<endl;

**cout<<**"\nЛевосторонний обход дерева: ";

**A.ObhodLeft (A.GetTree());**

**cout<<**"\nКонцевой обход дерева: "; A.ObhodEnd (A.GetTree());

**cout<<**"\nОбратный обход дерева: "; A.ObhodBack (A.GetTree());

**A.CleanTree (A.GetTree());**

**cout <<** "\n";

**system(**"PAUSE");

}

void **TREE::BuildTree ()**

// Построение бинарного дерева (рекурсивный алгоритм).

// Tree - указатель на корень дерева.

{

int **el;**

**cout<<**"Вводите ключи вершин дерева ...\n";

**cin>>el;**

while **(el!=0)**

**{ Search (el,&Tree); cin>>el; }**

}

void **TREE::Search (**int **x,node \*\*p)**

// Поиск вершины с ключом x в дереве со вставкой

// (рекурсивный алгоритм).

// \*p - указатель на корень дерева.

{

if **(\*p==NULL)**

**{**// Вершины в дереве нет; включить ее.

**\*p =** new(node);

**(\*\*p).Key = x; (\*\*p).Count = 1;**

**(\*\*p).Left = NULL; (\*\*p).Right = NULL; }**

else

if **(x<(\*\*p).Key) Search (x,&((\*\*p).Left));**

else

if **(x>(\*\*p).Key) Search (x,&((\*\*p).Right));**

else **(\*\*p).Count = (\*\*p).Count + 1;**

}

void **TREE::ObhodLeft (node \*\*w)**

//Левосторонний обход дерева.

//\*w - указатель на корень дерева.

{

if **(\*w!=NULL)**

**{**

**cout<<(\*\*w).Key<<**" ";

**ObhodLeft (&((\*\*w).Left));**

**ObhodLeft (&((\*\*w).Right));**

**}**

}

void **TREE::ObhodEnd (node \*\*w)**

//Концевой обход дерева.

//\*w - указатель на корень дерева.

{

if **(\*w!=NULL)**

**{ ObhodEnd (&((\*\*w).Left));**

**ObhodEnd (&((\*\*w).Right));**

**cout<<(\*\*w).Key<<**" "; }

}

void **TREE::ObhodBack (node \*\*w)**

//Обратный обход дерева.

//\*w - указатель на корень дерева.

{

if **(\*w!=NULL)**

**{ ObhodBack (&((\*\*w).Left));**

**cout<<(\*\*w).Key<<**" ";

**ObhodBack (&((\*\*w).Right)); }**

}

void **TREE::CleanTree (node \*\*w)**

//Очистка дерева.

//\*w - указатель на корень дерева.

{

if **(\*w!=NULL)**

**{ CleanTree (&((\*\*w).Left));**

**CleanTree (&((\*\*w).Right));**

delete **\*w; }**

}

void **TREE::Vyvod (node \*\*w,**int **l)**

//Изображение дерева \*w на экране дисплея

// (рекурсивный алгоритм).

//\*w - указатель на корень дерева.

{

int **i;**

if **(\*w!=NULL)**

**{ Vyvod (&((\*\*w).Right),l+1);**

for **(i=1; i<=l; i++) cout<<**" ";

**cout<<(\*\*w).Key<<endl;**

**Vyvod (&((\*\*w).Left),l+1); }**

}

int **TREE::Height (node \*\*w)**

//Определение высоты бинарного дерева.

//\*w - указатель на корень дерева.

{

int **h1,h2;**

if **(\*w==NULL)** return **(-1);**

else

**{**

**h1 = Height (&((\*\*w).Left));**

**h2 = Height (&((\*\*w).Right));**

if **(h1>h2)** return **(1 + h1);**

elsereturn **(1 + h2);**

**}**

}

# **69. Динамические структуры данных C++. Дерево. Постороение бинарного дерева (нерекурсивный алгоритм). Изображение бинарного дерева(нерекурсивный алгоритм). Пример программы и изображение биарного дерева(нерекурсивный алгоритм). Поиск вершины в бинарном дереве(нерекурсивный и рекусивный). Добавление вершины в бинарном дереве. Удаление вершины из бинарного дерева.**

Нерекурсивный алгоритм построения бинарного дерева поиска, который заключается в циклическом обращении к функции **TreeSearch (&Tree, el);** где **el** - значение информационного поля включаемой в дерево **Tree** вершины. Эта функция осуществляет поиск вершины дерева с информационным полем **el** и в случае неудачного поиска осуществляет включение вершины с информационным полем **el** в дерево.

1. Создаем "заглавное звено" бинарного дерева:

**Tree =** new(node);

**(\*Tree).Right = NULL;**

**p2 = Tree;**

**p1 = (\*p2).Right;**

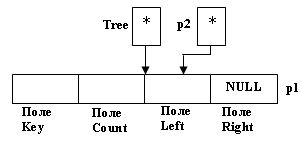


Рис.1. Создание "заглавного звена"

1. При поступлении информационного элемента Элем1 первой вершины дерева выполняем следующие команды:

**p1 =** new(node);

**(\*p1).Key = Элем1; (\*p1).Left = (\*p1).Right = NULL;**

**(\*p1).Count = 1;**

и получаем:

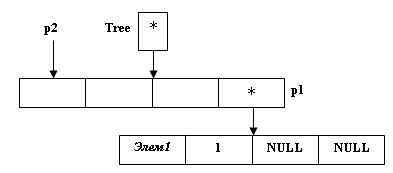


Рис.2. Размещение первой вершины дерева

1. Далее, в зависимости от поступающего значения информационного поля следующей вершины дерева, структура, содержащая этот элемент, будет "помещаться" либо в правое поддерево (если значение информационного поля больше значения информационного поля корня), либо в левое поддерево (если наоборот), либо будет изменено только содержимое поля **Count**.

Запишем алгоритм построения в виде функции на языке C++:

void **TreeSearch (node \*\*Tree,**int **el)**

// Поиск вершины с информационным полем el в дереве

// с последующим включением.

// \*Tree - указатель на корень дерева.

{

**node \*p1;**

**node \*p2;** // Указатель p2 "опережает" указатель p1.

int **d;** // Флаг для распознавания поддеревьев.

**p2 = \*Tree; p1 = (\*p2).Right;**

**d = 1;** // Флаг правого поддерева.

while **(p1!=NULL && d!=0)**

**{ p2 = p1;**

if **(el<(\*p1).Key) { p1 = (\*p1).Left; d = -1; }** //Флаг левого поддерева

else

if **(el>(\*p1).Key) { p1 = (\*p1).Right; d = 1; }**

else **d = 0; }**

if **(d==0) (\*p1).Count = (\*p1).Count + 1;**

else

**{ p1 =** new(node);

**(\*p1).Key = el; (\*p1).Left = (\*p1).Right = NULL; (\*p1).Count = 1;**

if **(d<0) (\*p2).Left = p1;** else **(\*p2).Right = p1;}**

}

Изображение бинарного дерева

struct **no** // Звено стека

**{**

**no \*sled;** // Указатель на вершину.

**node \*elem;** // Информационное поле.

int **ch;** // Уровень вершины.

**}**

**no \*stk;** // Стек "левых" ссылок.

**no \*stk1;** // Стек "правых" ссылок.

**node \*u;** // Указатель на структуру.

Теперь приведем и сам алгоритм:

**stk = stk1 = NULL;** // Первоначально оба стека пусты.

**n = 0;**

while **(t!=NULL)**

**{**

**PushStack (&stk1,&t,&n);**

// Пока не достигнут лист дерева, в стек stk помещаются

// левые ссылки тех структур, по которым перемещается

// указатель t.

if **((\*t).Right!=NULL)**

**{** if **((\*t).Left!=NULL) PushStack (&stk,&(\*t).Left,&n); t = (\*t).Right;}**

else **{**

// Как только будет достигнут самый правый лист, проверяем левую

//ссылку. Если она существует, то из стека правых ссылок удаляем

//последний адрес, и на печать выводим поле ключа этой вершины.

if **((\*t).Left!=NULL)**

**{** if **(stk1!=NULL)**

**{ PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl; }**

**t = (\*t).Left; }**

if **(stk==NULL) t = NULL;**

else **{**

// Если у вершины дерева указатели на правое и

// левое поддеревья равны NULL, а стек левых

// указателей не пуст, то...

while **((\*stk).elem!=(\*((\*stk1).elem)).Left)**

**{** // Удаляется адрес из стека правых ссылок

//и на печать выводится значение поля ключа

//вершины с данным адресом

**PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl; }**

// Как только выполнится условие

//((\*stk).elem==(\*((\*stk1).elem)).Left), удаля-

//ются адреса из обоих стеков, а поле ключа

//также выводится на экран дисплея.

**PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl;**

**PopStack (&stk,&t,&n); }**

**}**

**n = n + 1;**

**}**

**VyvodStack (&stk1);**

Оформим алгоритм в виде функции на языке C++:

void **VyvodTree (node \*t)**

// Изображение дерева, заданного указателем t,

// на экране дисплея (нерекурсивный алгоритм).

{

**no \*stk, \*stk1;**

**node \*u;**

int **i,n;**

**stk = stk1 = NULL; n = 0;**

while **(t!=NULL)**

**{ PushStack (&stk1,&t,&n);**

if **((\*t).Right!=NULL)**

**{** if **((\*t).Left!=NULL)**

**PushStack (&stk,&((\*t).Left),&n);**

**t = (\*t).Right; }**

else **{**

if **((\*t).Left!=NULL)**

**{** if **(stk1!=NULL)**

**{ PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl; }**

**t = (\*t).Left; }**

else

if **(stk==NULL) t = NULL;**

else

**{** while **((\*stk).elem!=(\*((\*stk1).elem)).Left)**

**{ PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl;**

**}**

**PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<< (\*u).Key<endl;**

**PopStack (&stk,&t,&n);**

**}**

**}**

**n = n + 1;**

**}**

**VyvodStack (&stk1);**

}

Пример программы и изображения

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **Key;**

int **Count;**

**node \*Left;**

**node \*Right;**

};

struct **no** // Звено стека

{

**node \*elem;** // Информационное поле.

int **ch;** // Уровень вершины.

**no \*sled;** // Указатель на вершину.

};

class **TREE**

{

private:

**node \*Tree;**

void **PushStack (no \*\*,node \*\*,**int **\*);**// Помещение звена с элементами \*el и n в

// стек. \*stk - указатель на стек.

void **PopStack (no\*\*,node \*\*,**int **\*);** // Извлечение из стека звена

// с элементами \*t и n.

// \*stk - указатель на стек

void **VyvodStack (no\*\*);** // Вывод содержимого стека на экран дисплея.

// \*stk - указатель на стек.

public:

**TREE () { Tree =** new(node); (\*Tree).Right = NULL; }

**node\* GetTreeRight () {**return **(\*Tree).Right;}**

void **TreeSearch (**int); // Поиск вершины с информационным полем el в дереве с

// последующим (в случае неудачного поиска!) включением

// в дерево. Tree - указатель на корень дерева.

void **VyvodTree (node \*);** //Построение дерева, заданного указателем t,

//на экране дисплея (нерекурсивный алгоритм).

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**TREE A;**

int **el;**

**cout<<**"Вводите значения информационных полей вершин: "<<endl;

**cin>>el;**

while **(el!=0)**

**{ A.TreeSearch (el); cin>>el; }**

**A.VyvodTree (A.GetTreeRight());**

**cout <<** "\n";

**system(**"PAUSE");

}

void **TREE::TreeSearch (**int **el)**

// Поиск вершины с информационным полем el в дереве с

// последующим (в случае неудачного поиска!) включением

// в дерево. Tree - указатель на корень дерева.

{

**node \*p1,\*p2;**

int **d;**

**p2 = Tree; p1 = (\*p2).Right; d = 1;**

while **(p1!=NULL && d!=0)**

**{**

**p2 = p1;**

if **(el<(\*p1).Key) {p1 = (\*p1).Left; d = -1;}**

else

if **(el>(\*p1).Key) {p1 = (\*p1).Right; d = 1;}**

else **d = 0;**

**}**

if **(d==0) (\*p1).Count = (\*p1).Count + 1;**

else

**{**

**p1 =** new(node);

**(\*p1).Key = el; (\*p1).Left = NULL;**

**(\*p1).Right = NULL; (\*p1).Count = 1;**

if **(d<0) (\*p2).Left = p1;**

else **(\*p2).Right = p1;**

**}**

}

void **TREE::VyvodTree (node \*t)**

//Построение дерева, заданного указателем t,

//на экране дисплея (нерекурсивный алгоритм).

{

**no \*stk,\*stk1;**

**node \*u;**

int **i,n;**

**stk = stk1 = NULL; n = 0;**

while **(t!=NULL)**

**{**

**PushStack (&stk1,&t,&n);**

if **((\*t).Right!=NULL)**

**{**

if **((\*t).Left!=NULL) PushStack (&stk,&((\*t).Left),&n);**

**t = (\*t).Right;**

**}**

else

**{**

if **((\*t).Left!=NULL)**

**{**

if **(stk1!=NULL)**

**{**

**PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl;**

**}**

**t = (\*t).Left;**

**}**

else

if **(stk==NULL) t = NULL;**

else

**{**

while **((\*stk).elem!=(\*((\*stk1).elem)).Left)**

**{**

**PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl;**

**}**

**PopStack (&stk1,&u,&n);**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*u).Key<<endl;**

**PopStack (&stk,&t,&n);**

**}**

**}**

**n = n + 1;**

**}**

**VyvodStack (&stk1);**

}

void **TREE::PushStack (no \*\*stk,node \*\*el,**int **\*n)**

// Помещение звена с элементами \*el и n в стек.

// \*stk - указатель на стек.

{

**no \*q;**

**q =** new(no);

**(\*q).elem = \*el; (\*q).ch = \*n;**

**(\*q).sled = \*stk; \*stk = q;**

}

void **TREE::PopStack (no\*\*stk,node \*\*t,**int **\*n)**

// Извлечение из стека звена с элементами \*t и n.

// \*stk - указатель на стек.

{

**no \*q;**

if **(\*stk!=NULL)**

**{**

**\*t = (\*\*stk).elem;**

**\*n = (\*\*stk).ch;**

**q = \*stk;**

**\*stk = (\*\*stk).sled;**

delete **q;**

**}**

}

void **TREE::VyvodStack (no\*\* stk)**

// Вывод содержимого стека на экран дисплея.

// \*stk - указатель на стек.

{

**node \*k;**

int **i,n;**

while **(\*stk!=NULL)**

**{**

**k = (\*\*stk).elem; n = (\*\*stk).ch;**

for **(i=0; i<=n; i++) cout<<**" ";

**cout<<(\*k).Key<<endl;**

**\*stk = (\*\*stk).sled;**

**}**

}

Поиск вершины

Нерекурсивный поиск проводится следующим образом:

#define **TRUE 1**

#define **FALSE 0**

void **Poisk (**int **k, node \*\*Tree, node \*\*Res)**

// Поиск вершины с ключом k в дереве (нерекурсивный алгоритм).

// \*Tree - указатель на вершину дерева.

// \*Res - указатель на найденную вершину

// или на лист, если вершины в дереве нет.

// B - глобальная булевская переменная:

// TRUE, если вершина с ключом k в дереве найдена,

// FALSE, в противном случае.

{

**node \*p, \*q;**

**B = FALSE; p = q = \*Tree;**

if **(\*Tree!=NULL)**

do **{**

**q = p;**

if **((\*p).Key==k) B = TRUE;**

else

**{ q = p;**

if **(k<(\*p).Key) p = (\*p).Left;**

else **p = (\*p).Right; }**

**}** while **(!B && p!=NULL);**

**\*Res = q;**

}

а рекурсивный - так:

node Poisk\_1 (int **k, node\*\* Tree)**

// Поиск вершины с ключом k в дереве (рекурсивный алгоритм).

// \*Tree - указатель на вершину дерева.

// Функция возвращает указатель на вершину,

// содержащую ключ k.

{

if **(\*Tree==NULL)** return **(NULL);**

else

if **((\*\*Tree).Key==k)** return **(\*Tree);**

else **{**

if **(k<(\*\*Tree).Key)** return **Poisk\_1 (k,&((\*\*Tree).Left));**

elsereturn **Poisk\_1 (k,&((\*\*Tree).Right));**

**}**

}

Добавление вершины

**s =** new(node); (\*s).Key = Элем; (\*s).Count = 1;

**(\*s).Left = (\*s).Right = NULL;**

и "подвешиваем" вершину:

**\*Tree = s;**

Иначе, если вершина, после которой будем прикреплять новую, не является листом дерева, то перед "подвешиванием" определяем, в какое поддерево относительно этой вершины будет включена новая:

if **(k<(\*r).Key) (\*r).Left = s;**

else **(\*r).Right = s;**

Приведем текст функции добавления:

void **Addition (node \*\*Tree,** int **k)**

// Добавление вершины k в бинарное дерево.

// \*Tree - указатель на вершину дерева.

// B - глобальная булевская переменная:

// TRUE, если вершина с ключом k в дереве найдена,

// FALSE, в противном случае.

{

**node \*r, \*s;**

**Poisk (k,Tree,&r);**

if **(!B)**

**{**

**s =** new(node);

**(\*s).Key = k; (\*s).Count = 1;**

**(\*s).Left = (\*s).Right = NULL;**

if **(\*Tree==NULL) \*Tree = s;**

else

if **(k<(\*r).Key) (\*r).Left = s;**

else **(\*r).Right = s;**

**}**

else **(\*r).Count += 1;**

}

Удаление вершины

Алгоритм удаления из бинарного дерева вершины с заданным ключом различает **три случая**:

* **вершины с заданным ключом в дереве нет;**
* **вершина с заданным ключом имеет не более одной исходящей дуги;**
* **вершина с заданным ключом имеет две исходящие дуги.**

void **Delete (node \*\*Tree,** int **k)**

// Удаление вершины k из бинарного дерева.

// \*Tree - указатель на вершину дерева.

{

**node \*q;**

if **(\*Tree==NULL) cout <<** "Вершина с заданным ключом не найдена!\n";

else

if **(k<(\*\*Tree).Key) Delete (&((\*\*Tree).Left),k);**

else

if **(k>(\*\*Tree).Key) Delete (&((\*\*Tree).Right),k);**

else **{**

**q = \*Tree;**

if **((\*q).Right==NULL) { \*Tree = (\*q).Left;** delete **q; }**

else

if **((\*q).Left==NULL) { \*Tree = (\*q).Right;** delete **q;}**

else **Delete\_1 (&((\*q).Left),&q);** // вершина с заданным ключом имеет две исходящие дуги

**}**

}

Освобождение памяти, занятой динамической переменной, происходит при помощи конструкции delete.

void **Delete\_1 (node \*\*r, node \*\*q)**

{

//\*q указатель на удаляемый узел

//\*r указатель правой компоненты левого поддерева

//вызывается только в случае если вершина с заданным ключом имеет две исходящие дуги

//спускается вдоль самой правой ветви левого поддерева удаляемого узла q

//затем заменяет существенную информацию (ключ и счетчик) в q

//соответствующими значениями самой правой компоненты r этого левого поддерева

//после чего от r можно освободиться

**node \*s;**

if **((\*\*r).Right==NULL)**

**{ (\*\*q).Key = (\*\*r).Key; (\*\*q).Count = (\*\*r).Count;\*q = \*r;**

**s = \*r; \*r = (\*\*r).Left;** delete **s; }**

else **Delete\_1 (&((\*\*r).Right),q);**

}

# **70. Динамические структуры данных C++. Дерево. Хэширование с помощью леса. Древовидно-кольцевая динамическая структура данных. Деревья Хаффмена.**

**Хэширование** - это процесс, в котором вы **подаёте на вход** некоторого хэширующего алгоритма некоторые достаточно **большие по объёму данные** (допустим миллион байт) и получаете **на выходе** относительно короткую (допустим 32 байта), но при этом достаточно **уникальную строку**, которая позволяет **отличить эти** ваши **данные** (что были на входе) **от каких-то других данных**. Эта строка называется "**хэш**".

Схема стpуктуpы данных:

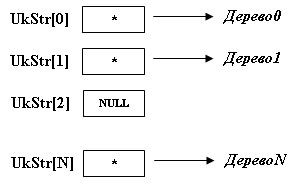


Рис.1. Структура данных

где **Деpево1, Деpево2, ..., ДеpевоN** - бинаpные деpевья поиска

Древовидно-кольцевая структура

Можно построить стpуктуpу данных, пpедставляющую собой совокупность **кольцевой** и **дpевовидной** стpуктуp. Указатели на коpни дpевовидной стpуктуpы pасположены в звеньях кольца. Само кольцо пpедставим с помощью однонапpавленного списка без заглавного звена:

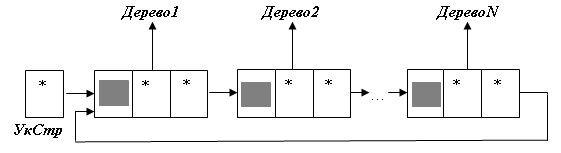


Рис.1. Древовидно-кольцевая структура данных

//Опишем деpевья..

struct **node**

{

int **key;**

int **count;**

**node \*Left;**

**node \*Right;**

};

//Тепеpь настала очеpедь кольца...

struct **zveno**

{

int **element;**

**Tree ukTree;**

**zveno\* sled;**

};

Деревья Хаффмена

**Построение кода Хаффмана сводится к построению соответствующего** [**бинарного дерева**](https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BA%D1%83%D1%87%D0%B0) **по следующему алгоритму:**

* Составим [список](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA) кодируемых символов, при этом будем рассматривать один символ как дерево, состоящее из одного элемента c весом, равным частоте появления символа в строке.
* Из списка выберем два узла с наименьшим весом.
* Сформируем новый узел с весом, равным сумме весов выбранных узлов, и присоединим к нему два выбранных узла в качестве детей.
* Добавим к списку только что сформированный узел вместо двух объединенных узлов.
* Если в списке больше одного узла, то повторим пункты со второго по пятый.

Алгоpитм Хаффмена выполняется в два этапа.

Hа **пеpвом этапе** множество символов pасполагается в поpядке уменьшения веpоятностей их появления. Каждому из символов будет соответствовать лист деpева, следовательно, можно пpедставить себе этот этап пpоцесса как постpоение линейного списка, содеpжащего листья будущего деpева.

Hа **втоpом этапе** алгоpитма пpоизводится повтоpяющееся сокpащение числа максимальных непеpесекающихся поддеpевьев посpедством объединения двух "легчайших" деpевьев для получения нового составного деpева.

Листья любого бинаpного деpева обpазуют пpефиксный код, и, наобоpот, для всякого пpефиксного кода существует такое деpево, что слова кода соответствуют его листьям.

Опишем используемое бинаpное деpево в теpминах языка С++:

struct **zveno**

{

char **Element;** //Символ.

float **Kol;** //Количество повтоpений,

//частота повтоpений.

**zveno\* Sled;**

**zveno\* Left;**

**zveno\* Right;**

**zveno\* Father;**

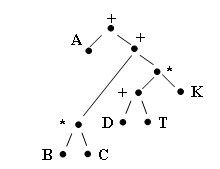
};

# **71. Динамические структуры данных C++. Дерево. Деревья-формулы. Построение дерева-формулы. Вычисления с помощью дерева-формулы. Бинарные деревья с различными листьями.**

Чаще всего аpифметические и логические выpажения описываются пpи помощи бинаpного деpева, котоpое в этом случае называется **деpевом-фоpмулой**. Все листья деpева-фоpмулы соответствуют пеpеменным или опеpандам, а все внутpенние веpшины соответствуют аpифметическим опеpациям.

Для пpимеpа pассмотpим выpажение

A + (B \* C + (D + T) \* K)



**восходящий обход узлов** **в постфиксной фоpме**:

A B C \* D T + K \* + + .

|  |  |
| --- | --- |
| a - b | a b - |
| a \* b + c | a b \* c + |
| a \* ( b + c ) | a b c + \* |

**нисходящий обход узлов** **пpефиксной фоpме**:

+ A + \* B C \* + D T K.

**смешанный обход**

А + B \* C + D + T \* K.

Построени дерева-формулы

Объявим структуру, описывающую узел такого дерева

struct Node

{

char data;

Node \*left, \*right;

};

typedef Node \*PNode;

Далее надо определить функцию, возвращающую приоритет операции, которая ей передана.

Определим приоритет 1 для сложения и вычитания и приоритет 2 для умножения и деления.

int Priority(char c)

{

switch(c){

case '+': case '-': return 1;

case '\*': case '/': return 2;

}

return 100; //это не арифметическая операция

}

(это уже другой код)

PNode MakeTree(char Expr[], int first, int last)

{

int MinPrt, i, k, prt;

PNode Tree = new Node; //создать в памяти новую вершину

if(first == last){ //конечная вершина: число или

Tree->data = Expr[first]; //переменная

Tree->left = NULL;

Tree->right = NULL;

return Tree;

}

MinPrt = 100;

for(i = first; i <= last; i ++){

prt = Priority ( Expr[i] );

if(prt <= MinPrt){ //ищем последнюю операцию

MinPrt = prt; //с наименьшим приоритетом

k = i;

}

}

Tree->data = Expr[k]; //внутренняя вершина (операция)

Tree->left = MakeTree(Expr, first,k-1); //рекурсивно строим

Tree->right = MakeTree(Expr, k+1,last); //поддеревья

return Tree;

}

Вычисление с помощью формулы.

#include <iostream>

#include <string.h>

#include <conio.h>

using namespace std;

struct Node

{

char data;

Node \*left, \*right;

};

typedef Node \*PNode;

int Priority(char c)

{

switch(c){

case '+': case '-': return 1;

case '\*': case '/': return 2;

}

return 100; //это не арифметическая операция

}

PNode MakeTree(char Expr[], int first, int last)

{

int MinPrt, i, k, prt;

PNode Tree = new Node; //создать в памяти новую вершину

if(first == last){ //конечная вершина: число или

Tree->data = Expr[first]; //переменная

Tree->left = NULL;

Tree->right = NULL;

return Tree;

}

MinPrt = 100;

for(i = first; i <= last; i ++){

prt = Priority ( Expr[i] );

if(prt <= MinPrt){ //ищем последнюю операцию

MinPrt = prt; //с наименьшим приоритетом

k = i;

}

}

Tree->data = Expr[k]; //внутренняя вершина (операция)

Tree->left = MakeTree(Expr, first,k-1); //рекурсивно строим

Tree->right = MakeTree(Expr, k+1,last); //поддеревья

return Tree;

}

int CalcTree(PNode Tree)

{

int num1, num2;

if (! Tree->left) //если нет потомков,

return Tree->data - '0'; //вернули число

num1 = CalcTree(Tree->left); //вычисляем поддеревья

num2 = CalcTree(Tree->right);

switch(Tree->data) { //выполняем операцию

case '+': return num1+num2;

case '-': return num1-num2;

case '\*': return num1\*num2;

case '/': return num1/num2;

}

return 32767; //неизвестная операция, ошибка!

}

void main()

{

setlocale(LC\_ALL, "Rus");

char s[80];

PNode Tree;

printf("Введите выражение > ");

gets(s);

Tree = MakeTree(s, 0, strlen(s)-1);

printf ( "= %d \n", CalcTree ( Tree ) );

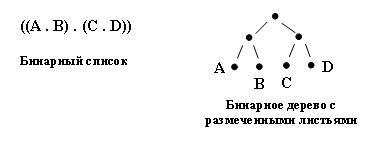
getch();

//system("PAUSE");

}

Бинарные деревья с разными листьями

Абстpактным типом данных, пpедставляющим значительный теоpетический и пpактический интеpес, является **бинаpный (двоичный) список**, котоpый pекуpсивно опpеделяется следующим обpазом:



Пеpечислим основные опеpации над типом данных бинаpное деpево с pазмеченными листьями:

* соединение двух деpевьев в одно (обозначается **CONS**);
* выделение левого поддеpева (обозначается **CAR**);

выделение пpавого поддеpева (обозначается **CDR**).

# **72 Структуры данных: двоичная куча (binary heap). Введение. Реализация двоичной кучи (binary heap). Реализация класса кучи. Конструктор кучи.**

**введение**

**Двоичная куча (binary heap)** – просто реализуемая структура данных, позволяющая быстро (за логарифмическое время) добавлять элементы и извлекать элемент с максимальным приоритетом (например, максимальный по значению).

**Двоичная куча**— структура данных, двоичное дерево, для которого выполнены три условия:

* Значение в любой вершине не меньше чем значение в потомках
* Расстояние до корня отличается не более чем на один уровень
* Заполняется слева на право

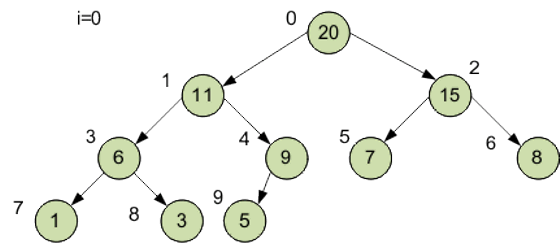
**Двоичная куча** представляет собой полное бинарное дерево, для которого выполняется основное свойство кучи: **приоритет каждой вершины больше приоритетов её потомков**.

В простейшем случае приоритет каждой вершины можно считать равным её значению. В таком случае структура называется ***max-куча***, поскольку корень поддерева является максимумом из значений элементов поддерева.

В качестве альтернативы, если сравнение перевернуть, то наименьший элемент будет всегда корневым узлом, такие кучи называют ***min-кучами***.

Двоичную кучу удобно хранить в виде одномерного массива, причем

* левый потомок вершины с индексом i имеет индекс 2\*i+1,
* правый потомок вершины с индексом i имеет индекс 2\*i+2.



Корень дерева (кучи) – элемент с индексом 0.

Высота двоичной кучи равна высоте дерева, то есть

**log2 (N+1)↑**,

где **N** – количество элементов массива, **↑** – округление в большую сторону до ближайшего целого.

Для представленной кучи

**log2 (10+1)↑ = 3,46↑ = 4**

Способ построить кучу из неупорядоченного массива – это по очереди добавить все его элементы. Временная оценка такого алгоритма оценивается как

**N·log2N**.

Можно построить кучу за **N** шагов. Для этого сначала следует построить дерево из всех элементов массива, не заботясь о соблюдении основного свойства кучи, а потом вызвать метод упорядочения для всех вершин, у которых есть хотя бы один потомок (так как поддеревья, состоящие из одной вершины без потомков, уже упорядочены).

Потомки гарантированно есть у первых **heapSize/2** вершин, где **heapSize** – размер кучи.

**Реализация класса кучи**

class Heap {

  static const int SIZE = 100; // максимальный размер кучи

  int \*h;         // указатель на массив кучи

  int HeapSize; // размер кучи  
public:

  Heap();  // конструктор кучи

  void addelem(int);  // добавление элемента кучи

  void outHeap();  // вывод элементов кучи в форме кучи

  void out();  // вывод элементов кучи в форме массива

  int getmax();  // удаление вершины (максимального элемента)

  void heapify(int);  // упорядочение кучи  
};

**Конструктор кучи**

Heap :: Heap() {

  h = new int[SIZE];

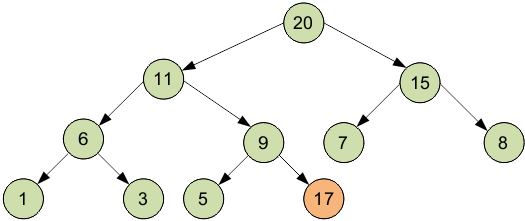
  HeapSize = 0;  
}

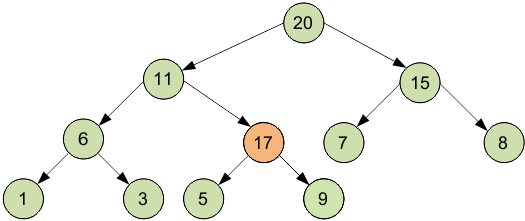
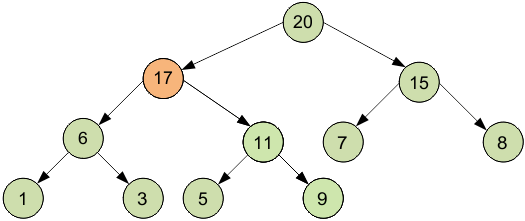
# **73 Структуры данных: двоичная куча (binary heap). Добавление элемента кучи. Вывод элементов кучи. Упорядочение кучи. Удаление вершины кучи (максимального элемента).**

**Двоичная куча** — структура данных, двоичное дерево, для которого выполнены три условия:

* Значение в любой вершине не меньше чем значение в потомках
* Расстояние до корня отличается не более чем на один уровень
* Заполняется слева на право

**Добавление элемента кучи**

Новый элемент добавляется на последнее место в массиве, то есть позицию с максимальным индексом.  


Возможно, что при этом будет нарушено основное свойство кучи, так как новый элемент может быть больше родителя. В таком случае новый элемент «поднимается» на один уровень (менять с вершиной-родителем) до тех пор, пока не будет соблюдено основное свойство кучи.  
  


Сложность алгоритма не превышает высоты двоичной кучи (так как количество «подъемов» не больше высоты дерева), то есть равна **log2 N**.

void Heap :: addelem(int n) {

  int i, parent;

  i = HeapSize;

  h[i] = n;

  parent = (i-1)/2;

  while(parent >= 0 && i > 0)  {

    if(h[i] > h[parent]) {

      int temp = h[i];

      h[i] = h[parent];

      h[parent] = temp;

    }

    i = parent;

    parent = (i-1)/2;

  }

 HeapSize++;  
}

**Вывод элементов кучи**

Вывод элементов в форме кучи

void Heap:: outHeap(void) {

  int i = 0;

  int k = 1;

  while(i < HeapSize) {

    while((i < k) && (i < HeapSize)) {

      cout << h[i] << » «;

      i++;

    }

    cout << endl;

    k = k \* 2 + 1;

 }  
}

Вывод элементов кучи в форме массива

void Heap:: out(void) {

  for(int i=0; i< HeapSize; i++) {

    cout << h[i] << » «; }

 cout<<endl;  
}

**Упорядочение кучи**

void Heap:: heapify(int i) {

  int left, right;

  int temp;

  left = 2\*i+1;

  right = 2\*i+2;

  if(left < HeapSize) {

    if(h[i] < h[left]) {

      temp = h[i];

      h[i] = h[left];

      h[left] = temp;

      heapify(left);

    }

  }

  if(right < HeapSize) {

    if(h[i] < h[right]) {

      temp = h[i];

      h[i] = h[right];

      h[right] = temp;

      heapify(right);

    }

  }  
}

В упорядоченном **max-heap** максимальный элемент всегда хранится в корне. Восстановить упорядоченность двоичной кучи после удаления максимального элемента можно, поставив на его место последний элемент и вызвав метод упорядочения для корня, то есть упорядочив все дерево.

**Удаление вершины кучи (максимального элемента)**

int Heap:: getmax(void) {

  int x;

  x = h[0];

  h[0] = h[HeapSize-1];

  HeapSize—;

  heapify(0);

  return(x);  
}

# **74. Рекурсия и рекурсивные алгоритмы. Сущность рекурсии. Виды рекурсии. Прямая рекурсия. Косвенная рекурсия. Линейная рекурсия. Ветвящаяся рекурсия. Бесконечная рекурсия. Сложная рекурсия. Опережающее описание.**

**Рекурсия** - метод определения класса объектов или методов предварительным заданием одного или нескольких (обычно простых) его базовых случаев или методов, а затем заданием на их основе правила построения определяемого класса, ссылающегося прямо или косвенно на эти базовые случаи

**Сущность рекурсии**

Процедура или функция может содержать вызов других процедур или функций. В том числе процедура может вызвать саму себя. Никакого парадокса здесь нет – компьютер лишь последовательно выполняет встретившиеся ему в программе команды и, если встречается вызов процедуры, просто начинает выполнять эту процедуру. Без разницы, какая процедура дала команду это делать.

**Базис рекурсии** - это предложение, определяющее некую начальную ситуацию или ситуацию в момент прекращения. Как правило, в этом предложении записывается некий простейший случай, при котором ответ получается сразу, даже без использования рекурсии.

**Шаг рекурсии** - это правило, в теле которого обязательно содержится, в качестве подцели, вызов определяемого предиката.

**Подпрограмма** - все, что находится внутри рекурсивной функции.

**Глубина рекурсии** - количество одновременно выполняемых процедур**.**

**Виды рекурсии:**

**Прямая рекурсия (простая)**

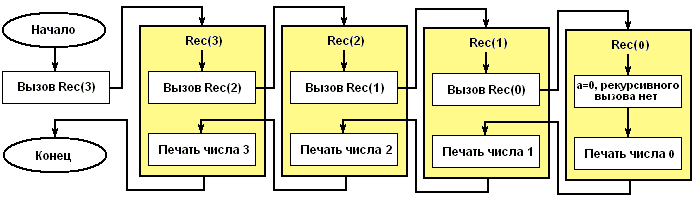
В случае прямой рекурсии вызов функцией самой себя делается непосредственно в этой же функции.

Пример:

int Rec(int a)

{if (a>0) Rec(a-1);

cout << a; }

Рассмотрим, что произойдет, если в основной программе поставить вызов, например, вида **Rec(3).** Ниже представлена блок-схема, показывающая последовательность выполнения операторов.

**Косвенная рекурсия** **(сложная)**

При этом виде рекурсии две или более функций вызывают друг друга попеременно. Для того чтобы стал возможен вызов функции, которая еще не описана (каждый идентификатор перед употреблением должен быть описан), вводится ***опережающее описание****,* т.е. прототипы функций (объявление функции, не содержащее тела функции, но указывающее имя функции, параметры и возвращаемый тип данных)

**Пример**. Определить, является ли введенное с клавиатуры число четным.

#include <iostream>

using namespace std;

bool isEven(int n); //четное

bool isOdd(int n); //нечетное

void main()

{ int n; bool m;

cout<<"Enter number ";

cin>>n;

m = isEven(n);

if(m == true) cout<<"Number is Even";

else cout<<"Number is Odd";

return 0; }

bool isEven(int n)

{ if(n == 0) return true; //условие окончания

else return isOdd(n - 1); }

bool isOdd(int n)

{ if(n == 0) return false;

else return isEven(n - 1);}

**Линейная рекурсия**

**Линейная рекурсия** - если исполнение подпрограммы приводит только к одному вызову этой же самой подпрограммы, то такая рекурсия называется линейной.

#include<iostream>

usingnamespace **std;**

void **function(**int **a);**

void **function (**int **a)**

{

**cout <<** " input a= " **<< a << endl;**

if **(a > 0)**

**function(a - 1);**

**cout <<** "output a= " **<< a << endl;**

}

int **main ( )**

{

**function(3);**

**system(**"PAUSE");

return **NULL;**

}

**Ветвящаяся рекурсия**

**Ветвящаяся рекурсия** - если каждый экземпляр подпрограммы может вызвать себя несколько раз, то рекурсия называется нелинейной или "ветвящейся".

#include<iostream>

usingnamespace **std;**

int **function(**int **a);**

int **function (**int **a)**

{

if **(a > 3)**

**a = function (a - 1) \* function(a - 2);**

return **a;**

}

int **main ()**

{

**cout << function(6) << endl;**

**system(**"PAUSE");

return **NULL;**

}

**Бесконечная рекурсия**

**Бесконечная рекурсия** (на самом деле это условное обозначение так как при переполнении памяти компьютера программа выдаст ошибку и/или завершит ее в аварийном режиме).

Одна из самых больших опасностей рекурсии - бесконечный вызов функцией самой себя.

Например:

void **function()**

{

**function();**

}

При использовании подобных алгоритмов появляется ошибка, предупреждающая о переполнении стека. Причиной такой проблемы чаще всего является отсутствие базиса, либо других точек останова, так же неправильно заданные точки прерывания рекурсии

# **75. Рекурсия и рекурсивные алгоритмы. Имитация работы цикла с помощью рекурсии. Пример 1. Инкремент счетчика итераций. Пример 2. Декремент счетчика итераций. Пример 3. Двойной цикл. Пример 4: Перевод числа в двоичную систему.**

Если функция вызывает сама себя, то, по сути, это приводит к повторному выполнению содержащихся в ней инструкций, что аналогично работе цикла. Некоторые языки программирования не содержат циклических конструкций вовсе, предоставляя программистам организовывать повторения с помощью рекурсии (например, Пролог, где рекурсия - основной прием программирования). Считается, что лучше использовать циклы, а рекурсию только для специализированных алгоритмов.

**Пример 1.**

void Loop(int i, int n) //0,10

{cout << “Hello №”<< i << endl;

if (i<n)

Loop(i + 1, n);} // выводится от 0 до 10

**Пример 2.**

void Loop(int i, int n) //0,10

{if (i<n)

Loop(i + 1, n);

cout << “Hello №”<< i << endl;} //выводится от 10 до 0

**Пример 3**.

void Loop(int,int);

void main()

{Loop(1,10);}

void Loop(int i, int n)

{ cout << “Hello №”<< i << endl;

if (i<n)

Loop(i+1,n);

cout << “Hello №”<< i << endl;} //выводится от 0 до 10 и сразу же от 10 до 0

**Пример 4.**

void Binary(int x)

{int c;

c=x%2;

x=x/2;

if (x>0)

Binary(x);

cout << c;} //вывод с конца

void main()

{cout << "Binary Representation of x : ";

Binary(9);} //р-т Binary Representation of x : 1001

# **76. Рекурсия и рекурсивные алгоритмы. Рекуррентные соотношения. Рекурсия и итерация. Произвольное количество вложенных циклов. Примеры рекурсивных алгоритмов.**

Говорят, что последовательность векторов \{\vec{x}_n\} задана рекуррентным соотношением, если задан начальный вектор \vec{x}_0=(x_0^1, \ldots, x_0^D) и функциональная зависимость последующего вектора от предыдущего

\vec{x}_n=\vec{f}(\vec{x}_{n-1})~~~~~(1)

Иными словами, **рекуррентные соотношения** - однотипные формулы, которые связывают между собой идущие друг за другом элементы некоторой последовательности (это может быть последовательность чисел, функций и т. д.). В зависимости от природы объектов, связанных Р. с., эти соотношения могут быть алгебраическими, функциональными, дифференциальными, интегральными и т. п.

Простым примером величины, вычисляемой с помощью рекуррентных соотношений, является факториал

n!=1 \cdot 2 \cdot 3 \cdot \ldots \cdot n

Очередной факториал n! можно вычислить по предыдущему как:

n!=(n-1)! \cdot n~~~~~(2)

Введя обозначение x_n=n! , получим соотношение:

x_n=x_{n-1} \cdot n,~x_0=1~~~~~(3)

Вектора \vec{x}_n из формулы **(1)** можно интерпретировать как наборы значений переменных. Тогда вычисление требуемого элемента последовательности будет состоять в повторяющемся обновлении их значений. В частности для факториала:

int x=1;

for(int i=2;i<n;i++)

x \*= i;

cout << x;

Каждое такое обновление **(x \*= i)** называется **итерацией**, а процесс повторения итераций – **итерированием**.

Обратим, однако, внимание, что соотношение **(1)** является чисто рекурсивным определением последовательности и вычисление n-го элемента есть на самом деле многократное взятие функции **f** от самой себя:

x_n=\displaystyle{\underbrace{f(f(...f(x_0)))}_n}~~~~~(4)

В частности для факториала можно написать:

int fact(int i)

{ if (i>1) return i \* fact(i - 1);

else return 1; }

Следует понимать, что вызов функций влечет за собой некоторые дополнительные накладные расходы, поэтому первый вариант вычисления факториала будет несколько более быстрым. Вообще итерационные решения работают быстрее рекурсивных. А, например, при нахождении числа Фибоначчи рекурсивно каждый вызов функции создает сразу две копии себя, каждая из копий – еще две и т.д. Количество операций растет с номером n экспоненциально, хотя при итерационном решении достаточно линейного по n количества операций.

**Итерационная реализация нахождения чисел Фибоначчи:**

void Fib(int n)

{

int F0 = 0, F1 = 1, F2;

cout << F0 << " " << F1 << " ";

for (int i = 2; i<n; i++)

{

F2 = F1 + F0;

F0 = F1;

F1 = F2;

cout << F1 << " ";

}

}

**Произвольное количество вложенных циклов**

Разместив рекурсивные вызовы внутри цикла, по сути, получим вложенные циклы, где уровень вложенности равен глубине рекурсии.

Для примера напишем процедуру, печатающую все возможные сочетания из **k** чисел от 1 до **n** (\mathrm{C}_n^k). Числа, входящие в каждое сочетание, будем печатать в порядке возрастания. Сочетания из трех чисел **(k=3)** печатаются так:

for(int i1 = 1; i<n; i++)

for(int i2 = i1+1; i<n;i++)

for(int i3=i2+1; i<n; i++)

cout << i1 << ' ' << i2 << ' ' << i3<<endl;

Однако, если количество чисел в сочетании задается переменной, то придется прибегнуть к рекурсии.

**Примеры рекурсивных алгоритмов**

**Пример.** Разработать рекурсивный алгоритм вычисления квадрата натурального числа

Известно, что (n + 1)2 = n2 + 2 \* n + 1 и 12 = 1,

отсюда

|  **1**,если **n = 1;**

  kv(n) = |

| kv(n - 1) + 2(n - 1) + 1, если n > 1

int kv(int i)

{if (i == 1)return 1;

else return kv(i - 1) + 2 \* (i - 1) + 1; }

void main()

{int i;

for(i = 1; i < 10; i++)

cout<<kv(i)<<endl; }

**Пример** 1. Написать рекурсивную функцию для поиска максимального элемента в одномерном массиве:

int max (int \*a, int i)

{ if (i==1) return a[1]; else if a[i]>max(a,i-1) return a[i]; else return max(a,i-1); }

**Пример** 2. Написать рекурсивную функцию, переводящую целое число из десятичной системы счисления в восьмеричную:

void convert (int z)

{ if (z>1) convert(z/8); cout<<(z % 8); }

**Пример 3.** Составить функцию подсчета количества x(m) разбиений натурального числа m (вводится с клавиатуры), то есть его представлений в виде суммы натуральных чисел.

P(m,n) разбиений натурального числа m со слагаемыми, не превосходящими n.

x(m)=P(m,m) - дать пояснения свойства:

P(m,1)=1 m=1+1+…+1 P(1,n)=1

P(m,n)=P(m,m) при n>m P(m,m)=P(m,m-1)+1 P(m,n)=P(m,n-1)+P(m-n,n) (n<m).

int deco (int m, int n) { if (m==1) return 1; if (n==1) return 1; if (n>m) return deco(m,m); else if (m==n) return 1+deco(m,m-1);

else return(deco(m, n-1)+deco(m-n,n)); }

# **77. Рекурсия и рекурсивные алгоритмы. Избавление от рекурсии. Явное использование стека. Запоминание последовательности рекурсивных вызовов. Определение узла дерева по его номеру.**

Любой рекурсивный алгоритм может быть переписан без использования рекурсии. Заметим, что быстродействие алгоритмов при избавлении от рекурсии, как правило, повышается. Еще одной причиной чтобы избавиться от рекурсии является ограничение на объем хранимых программой локальных переменных и значений параметров одновременно выполняющихся процедур. При очень глубокой рекурсии этот объем возрастает, и программа перестает работать, выдавая **ошибку «Stack overflow**» (переполнение стека). Так почему же люди продолжают пользоваться рекурсивными алгоритмами? Очевидно, потому что это проще и естественнее, чем соответствующие нерекурсивные решения.

**Явное использование стека**

**Стеком** называется структура данных, в которой добавление и извлечение данных происходит с одного конца, называемого вершиной стека. Наглядным образом стека может служить стопка тарелок – добавлять или забрать тарелки можно только сверху. Каждая тарелка соответствует элементу данных.

Когда одна функция вызывает другую, то параметры первой функция, а также место, с которого ее выполнение должно продолжиться после того как отработает вызванная функция (точка возврата), запоминаются в так называемом **стеке вызовов**. Если вызванная функция в свою очередь чего-нибудь вызывает, то ее параметры и точка возврата также добавляются в стек.

При рекурсивных вызовах стек вызовов хранит цепочку из данных об одновременно работающих функция. Во всех продвинутых средах разработки эту цепочку вместе с запомненными параметрами процедур можно просмотреть во время отладки.

Универсальный **способ избавиться от рекурсии** – это самостоятельно запрограммировать те действия со стеком, которые фактически происходят, когда вы используете рекурсивные вызовы

**Запоминание последовательности рекурсивных вызовов**

Мы знаем, что рекурсивные вызовы образуют дерево, где каждый узел соответствует вызову одной процедуры. Последовательность выполнения этих процедур соответствует тому или иному алгоритму обхода узлов. Если требуется много раз обойти узлы одного и того же дерева, то можно один раз обойти их рекурсивно, запомнить количество и последовательность узлов, а затем, пользуясь этой информацией, обходить узлы уже нерекурсивно.

Например, имеется задача вычисления арифметических выражений, заданных строкой. Может возникнуть ситуация, когда одно и то же выражение потребуется вычислить много раз при различных значениях переменной **x**. Синтаксическое дерево, которое требуется обходить при таких вычислениях, не зависит от **x**. Можно обойти его один раз, построив при этом массив, где каждый элемент будет соответствовать узлу дерева, а их последовательность – **порядку обхода**. Повторные вычисления при новом**x** потребуют только нерекурсивного перебора элементов массива.

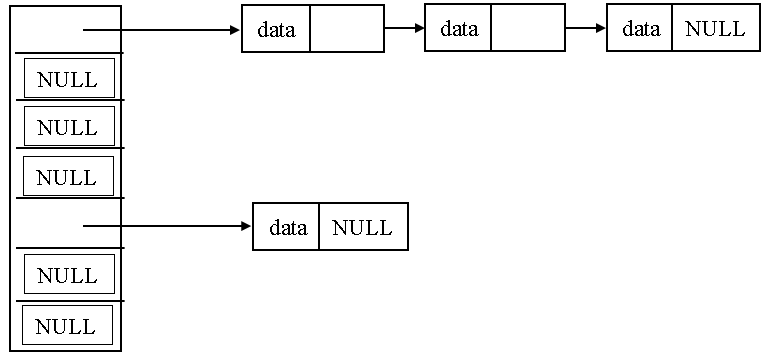
Такой подход не избавляет нас от рекурсии полностью. Однако он позволяет ограничиться только одним обращением к рекурсивной функции, что может быть достаточно, если мотивом является забота о максимальной производительности.

**Определение узла дерева по его номеру**

Идея данного подхода в том, чтобы заменить рекурсивные вызовы простым циклом, который выполнится столько раз, сколько узлов в дереве, образованном рекурсивными процедурами. Что именно будет делаться на каждом шаге, следует определить по номеру шага. Сопоставить номер шага и необходимые действия – задача не тривиальная и в каждом случае ее придется решать отдельно.

# **78.Хеш-таблицы. Ключевые термины темы. Простое представление хеш-таблиц. Практическое применение хеш-таблиц.**

Хэш-таблицы (hash tables – перемешанные таблицы, таблицы с вычисляемыми адресами) – одно из величайших изобретений информатики. Сочетание массивов и списков с небольшой добавкой математики позволило создать эффективную структуру для хранение и получения динамических данных .



Идея хэш-реализации состоит в том, что работа с одним большим множеством сводится к работе с массивом небольших множеств. Рассмотрим, к примеру , записную книжку. Она содержит список фамилий людей с их телефонами( телефоны – это нагрузка элементов множества). Страницы записной книжки помечены буквами алфавита; страница, помеченная некоторой буквой, содержит только фамилии, начинающиеся с этой буквы. Таким образом, всё множество фамилий разбито на 28 подмножеств, соответствующих буквам русского алфавита. При поиске фамилии мы сразу открываем записную книжку на странице, помеченной первой буквой фамилии, и в результате поиск значительно убыстряется.

**Типичное применение хэш-таблиц**

* Для поиска информации
* Таблица символов компилятора
* Программирование шахмат
* **Для баз данных телефонных номеров**.
* **Каталог книг**.
* **Для хранения паролей пользователей**.
* **Браузер хранит адреса посещенных страниц в хеш-таблице**.

**Ключевые термины**

**Вторичные ключи** – это ключи, не позволяющие однозначно идентифицировать запись в таблице.

**Закрытое хеширование или Метод открытой адресации** – это технология разрешения коллизий, которая предполагает хранение записей в самой хеш-таблице.

**Коллизия**– это ситуация, когда разным ключам соответствует одно значение хеш-функции.

**Коэффициент заполнения хеш-таблицы** – этоколичество хранимых элементов массива, деленное на число возможных значений хеш-функции.

**Открытое хеширование или Метод цепочек** – это технология разрешения коллизий, которая состоит в том, что элементы множества с равными хеш-значениями связываются в цепочку-список.

**Первичные ключи** – это ключи, позволяющие однозначно идентифицировать запись.

**Повторное хеширование** – это поиск местоположения для очередного элемента таблицы с учетом шага перемещения.

**Пространство записей** – это множество тех ячеек памяти, которые выделяются для хранения таблицы.

**Пространство ключей** – это множество всех теоретически возможных значений ключей записи.

**Синонимы**– это совпадающие ключи в хеш-таблице.

**Хеширование**– это преобразование входного массива данных определенного типа и произвольной длины в выходную битовую строку фиксированной длины.

**Хеш-таблица** – это структура данных, реализующая интерфейс ассоциативного массива, то есть она позволяет хранить пары вида «ключ- значение» и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу

# ***79*. Хеш-таблицы. Алгоритмы хеширования данных. Таблица прямого доступа. Метод остатков от деления. Метод функции середины квадрата. Метод свертки. Открытое хеширование. Закрытое хеширование (открытая индексация).**

# ***Алгоритмы***

Хеш-таблицы должны соответствовать следующим свойствам.

* Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение является индексом в исходном массиве.
* Количество хранимых элементов массива, деленное на число возможных значений хеш-функции, называется коэффициентом заполнения хеш-таблицы ( load factor ) и является важным параметром, от которого зависит среднее время выполнения операций.
* Операции поиска, вставки и удаления должны выполняться в среднем за время O(1).
* Механизм разрешения коллизий является важной составляющей любой хеш-таблицы.

# ***Таблица прямого доступа***

Простейшей организацией таблицы, обеспечивающей идеально быстрый поиск, является таблица прямого доступа. В такой таблице ключ является адресом записи в таблице или может быть преобразован в адрес, причем таким образом, что никакие два разных ключа не преобразуются в один и тот же адрес. При создании таблицы выделяется память для хранения всей таблицы и заполняется пустыми записями. Таблицы прямого доступа очень эффективны в использовании, но, к сожалению, область их применения весьма ограничена.

***Метод остатков от деления***

Простейшей хеш-функцией является деление по модулю числового значения ключа **Key** на размер пространства записи **HashTableSize**. Результат интерпретируется как адрес записи. Следует иметь в виду, что такая функция хорошо соответствует первому, но плохо – последним трем требованиям к хеш-функции. Операция деления по модулю обычно применяется как последний шаг в более сложных функциях хеширования, обеспечивая приведение результата к размеру пространства записей.

На практике, **метод деления** – самый распространенный.

//функция создания хеш-таблицы метод деления по модулюint Hash(int Key, int HashTableSize) {//HashTableSize return Key % HashTableSize;}

*Метод функции середины квадрата*

Следующей хеш-функцией является функция середины квадрата. Значение ключа преобразуется в число, это число затем возводится в квадрат, из него выбираются несколько средних цифр и интерпретируются как адрес записи.

*Метод свертки*

Еще одной хеш-функцией можно назвать функцию свертки. Цифровое представление ключа разбивается на части, каждая из которых имеет длину, равную длине требуемого адреса. Над частями производятся определенные арифметические или поразрядные логические операции, результат которых интерпретируется как адрес. Например, для сравнительно небольших таблиц с ключами – символьными строками неплохие результаты дает функция хеширования, в которой адрес записи получается в результате сложения кодов символов, составляющих строку-ключ.

***Открытое хеширование***

Основная идея базовой структуры при открытом (внешнем) хешировании заключается в том, что потенциальное множество (возможно, бесконечное) разбивается на конечное число классов. Для ***В*** классов, пронумерованных от 0 до http://ok-t.ru/studopedia/baza5/117194492528.files/image047.png, строится хеш-функция http://ok-t.ru/studopedia/baza5/117194492528.files/image049.pngтакая, что для любого элемента *х* исходного множества функция http://ok-t.ru/studopedia/baza5/117194492528.files/image049.pngпринимает целочисленное значение из интервала http://ok-t.ru/studopedia/baza5/117194492528.files/image052.png, соответствующее, классу, которому принадлежит элемент *х*. Часто классы называют *сегментами*, поэтому будем говорить, что элемент *х* принадлежит сегменту http://ok-t.ru/studopedia/baza5/117194492528.files/image049.png. Массив, называемый таблицей сегментов и проиндексированный номерами сегментов http://ok-t.ru/studopedia/baza5/117194492528.files/image052.png, содержит заголовки для *B* списков. Элемент *х*, относящийся к *i*-му списку – это элемент исходного множества, для которого http://ok-t.ru/studopedia/baza5/117194492528.files/image055.png.

***Закрытое хеширование***

При закрытом (внутреннем) хешировании в хеш-таблице хранятся непосредственно сами элементы, а не заголовки списков элементов. Поэтому в каждой записи (сегменте) может храниться только один элемент. При закрытом хешировании применяется методика ***повторного хеширования***. Если осуществляется попытка поместить элемент ***х*** в сегмент с номером http://ok-t.ru/studopedia/baza5/117194492528.files/image049.png, который уже занят другим элементом (коллизия), то в соответствии с методикой повторного хеширования выбирается последовательность других номеров сегментов http://ok-t.ru/studopedia/baza5/117194492528.files/image059.png, куда можно поместить элемент *х*. Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то, следовательно, таблица заполнена, и элемент *х* добавить нельзя.

При поиске элемента *х* необходимо просмотреть все местоположения http://ok-t.ru/studopedia/baza5/117194492528.files/image061.png, пока не будет найден *х* или пока не встретится пустой сегмент. Чтобы объяснить, почему можно остановить поиск при достижении пустого сегмента, предположим, что в хеш-таблице не допускается удаление элементов. Пусть http://ok-t.ru/studopedia/baza5/117194492528.files/image063.png– первый пустой сегмент. В такой ситуации невозможно нахождение элемента *х* в сегментах http://ok-t.ru/studopedia/baza5/117194492528.files/image065.pngи далее, так как при вставке элемент *х*вставляется в первый пустой сегмент, следовательно, он находится где-то до сегмента http://ok-t.ru/studopedia/baza5/117194492528.files/image063.png.

# **80. Хеш-таблицы. Основные, наиболее часто используемые функции хэширования.**

охарактеризуем **наиболее часто используемые функции хэширования**, применяемые для ЭЦП.

* **SHA** - Secure Hash Algorithm (1992)  
  160-разрядный хэш-код (дайджест). НЕ устойчив к коллизиям.  
  512-битовые блоки.
* **SHA-1** - Secure Hash Algorithm 1 (1995)  
  Модификация SHA. Исправлены недостатки. Решает проблему коллизий.
* **MAC** - Message Authentication Code - код аутентификации (проверки подлинности) сообщения.  
  Это зависящая от ключа однонаправленная хэш-функция. Простейшим способом преобразования хэш-функции в MAC является шифрование хэш-значения симметричным алгоритмом.
* **HMAC**  
  Один из вариантов добавления секретного ключа в уже существующий алгоритм хэширования. Функция хэширования в этом алгоритме интерпретируется как «черный ящик», то есть функция хэширования реализована как отдельный модуль и ее можно менять.  
  Алгоритм HMAC [представлен в документе RFC 2104] принят как обязательный в протоколе IPSec и используется в ряде других протоколов Internet (TLS, SET и другие)  
  Широко используются на практике также функции, разработанные Роном Ривестом:
* **MD2** - Message Digest #2  
  Низкоскоростной, но очень надежный алгоритм, создающий 128-разрядные дайджесты данных любого объема.
* **MD4** - Message Digest #4 (1990)  
  Более скоростной, но менее надежный алгоритм, создающий 128-разрядные дайджесты данных любого объема. 512-битовые блоки. Есть дефекты.
* **MD5** - Message Digest #5 (1992)  
  Версия MD4 с повышенной надежностью, преимущества также и в скорости. 128-разрядные дайджесты данных любого объема.  
  Неустойчив к коллизиям! Не используется для долговременных ЭЦП.

# 81. Хеш-таблицы. Коллизи (они же столкновения). Пример коллизий. Методы разрешения коллизий. Борьба с коллизиями. Метод цепочек. Открытая индексация (или закрытое хеширование). Переполнение таблицы и рехеширование. Оценка качества хеш-функции.

**Хеш-табли́ца** структура данных, реализующая интерфейс ассоциативного массива. В отличие от [деревьев поиска](https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0,_%D0%BD%D0%B0%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D1%80%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F), реализующих тот же интерфейс, обеспечивают меньшее время отклика в среднем.

Методы решения коллизий

**Методом цепочек** называется метод, в котором для разрешения коллизий во все записи вводятся указатели, используемые для организации списков √ цепочек переполнения.

**Метод открытой адресации** состоит в том, чтобы, пользуясь каким-либо алгоритмом, обеспечивающим перебор элементов таблицы, просматривать их в поисках свободного места для новой записи.Делится на 3 разновидности.

**Линейное опробование** сводится к последовательному перебору элементов таблицы с некоторым фиксированным шагом

a=h(key) + c\*i ,

где i - номер попытки разрешить коллизию.

**Квадратичное опробование** отличается от линейного тем, что шаг перебора элементов не линейно зависит от номера попытки найти свободный элемент

a = h(key2) + c\*i + d\*i2

Еще одна разновидность метода открытой адресации, которая называется **двойным хешированием**, основана на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хеш-функций

a=h1(key) + i\*h2(key).

Борьба

В идеальном случае, когда заранее известны все пары ключ-значение, достаточно легко реализовать идеальную хеш-таблицу, в которой время поиска будет постоянным.

Метод цепочек

**Каждая** **ячейка массива является указателем на связный** **список (цепочку) пар** **ключ-****значение, соответствующих одному и тому же хеш-значению ключа.** **Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента**

Открытая инцексация

**Если для поиска элемента необходимо более 3 – 4 сравнений, то эффективность использования такой хеш-таблицы пропадает и ее следует реструктуризировать (т.е. найти другую хеш-функцию), чтобы минимизировать количество сравнений для поиска элемента**

Переполнение

**Очевидно, что по мере заполнения хеш-таблицы будут происходить коллизии и в результате их разрешения методами открытой адресации очередной адрес может выйти за пределы адресного пространства таблицы. Что бы это явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адресов, выдаваемым хеш-функцией.**

Оценка

**При удачном построении хеш-функции таблица заполняется более равномерно, уменьшается число коллизий и уменьшается время выполнения операций поиска, вставки и удаления.**

# 82 Хеш-таблицы. Организация данных для ускорения поиска по вторичным ключам. Инвертированные индексы. Битовые карты.

**Хеш-табли́ца** структура данных, реализующая интерфейс ассоциативного массива. В отличие от [деревьев поиска](https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0,_%D0%BD%D0%B0%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D1%80%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F), реализующих тот же интерфейс, обеспечивают меньшее время отклика в среднем.

Для таблицы строится отдельный набор данных, содержащий так называемые инвертированные индексы. Вспомогательный набор содержит для каждого значения вторичного ключа отсортированный список адресов записей таблицы, которые содержат данный ключ. Поиск осуществляется по вспомогательной структуре достаточно быстро, так как фактически отсутствует необходимость обращения к основной структуре данных. Область памяти, используемая для индексов, является относительно небольшой по сравнению с другими методами организации таблиц.

Для таблиц небольшого объема используют организацию вспомогательной структуры данных в виде битовых карт. Для каждого значения вторичного ключа записей основного набора данных записывается последовательность битов. Длина последовательности битов равна числу записей. Каждый бит в битовой карте соответствует одному значению вторичного ключа и одной записи. Единица означает наличие ключа в записи, а ноль √отсутствие.

# **83. Сортировка данных. Основные понятия и определения. Оценка эффективности алгоритмов. Нотация Big-O. Формулировка задачи. Оценка алгоритма сортировки. Нотация "большое О". Список алгоритмов сортировки.**

**Сортировка** - это процесс упорядочения набора элементов в возрастающем или убывающем порядке.

Если в объектах содержится несколько данных-членов, нужно указать, какая переменная определяет порядок следования объектов. Эта переменная называется **ключом**. Например, если хранится информация о людях, то ключом может быть возраст, фамилия и пр.

**Алгоритмы сортировки** можно **классифицировать** по нескольким признакам.

По **размещению элементов**: **внутренняя** - в памяти, **внешняя** - в файле данных.

Методы внутренней сортировки можно разделить на две группы:

- методы, не требующие резерва памяти (метод выборки, "пузырька", вставки, Шелла);

- методы, требующие резерва памяти (метод квадратич. выборки, метод слияния и другие).

По **виду данных**: сортировка массивов, массивов указателей, списков и других структур данных.

По **способу выбора элементов**:

**ОБМЕННАЯ сортировка** (“пузырек”, шейкер-сортировка); **сортировка ВЫБОРОМ** (прямая выборка, линейный выбор с подсчетом, квадратичная сортировка с выбором); **сортировка ВСТАВКАМИ** (простая вставка, вставка погружением, сортировка Шелла);  **сортировка РАЗДЕЛЕНИЕМ** (быстрая сортировка разделением, поразрядная сортировка разделением); **сортировка ПОДСЧЕТОМ**; **ПИРАМИДАЛЬНАЯ сортировка**; **сортировка СЛИЯНИЕМ** (однократное слияние, циклическое слияние).

**Задача сортировки** — облегчение последующего поиска элементов в отсортированном множестве во внутренней или внешней памяти.

***Оценка эффективности алгоритмов*** – трудоемкость (время выполнения программы) и требуемый объем памяти. Аналитически оценки получают на основе подсчета базовых операций сравнения и обмена. Как правило, эти операции сосредоточены во внутренних циклах алгоритма.

***Основной*** характеристикой алгоритма сортировки является показатель вычислительной сложности, зависящий от количества сортируемых элементов.

***Вторым*** показателем эффективности алгоритма сортировки является объем дополнительной памяти, используемый для сортировки. По этому показателю алгоритмы делятся на три группы:

**1.**алгоритмы, которые организуют сортировку на том же месте и не используют дополнительную память (кроме стека и таблицы);

**2.** алгоритмы, которые используют представление в виде связанного списка или другие структуры указателей или индексов;

**3.** алгоритмы, которые требуют дополнительную память для размещения еще одной копии массива сортировки.

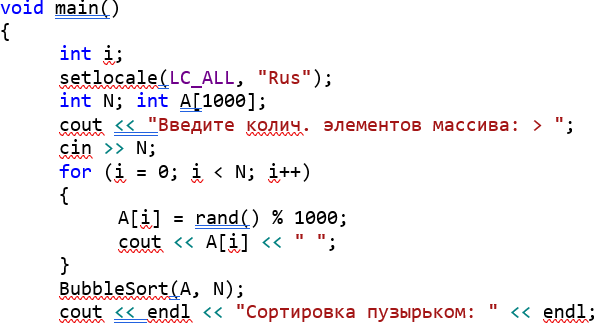
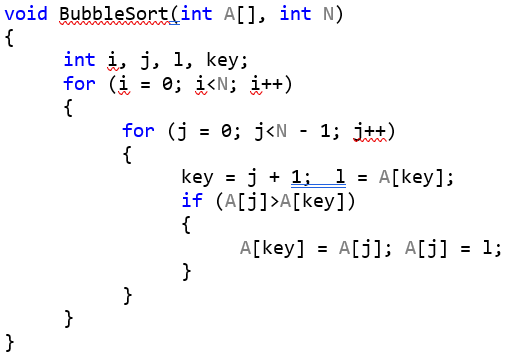
***Нотация Big-O***  - порядок роста или асимптотическая оценка роста. Big-O = O(n): О - объем данных, n – нотация.

Время сортировки массива зависит от размера сортируемого массива и его упорядоченности. Время работы алгоритма определяется числом шагов, которые он выполняет. Будем считать, что срока псевдокода требует не более чем фиксированного количества элементарных операций, если строка не является формулировкой сложных действий. В последнем случае строка соответствует программной функции, а в основном алгоритме ей соответствует вызов той функции. Вызов функции и ее выполнение отличаются по времени.

**Нотация «большое О».** Обычно она рассматривает верхнюю границу производительности алгоритмов, то есть рассматривается поведение в худшем случае. Для практического использование это самый безопасный способ рассмотрение производительности алгоритма. Так же нотация «большое О» рассматривает производительность алгоритма при росте размера задачи.

# **84.** **Сортировка данных. Сортировка пузырьком (Bubble sort). Описание алгоритма. Реализация. Анализ алгоритма.**

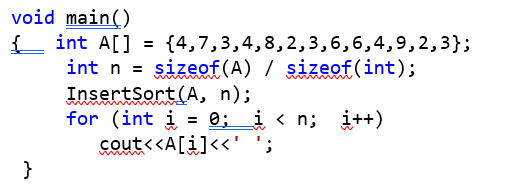
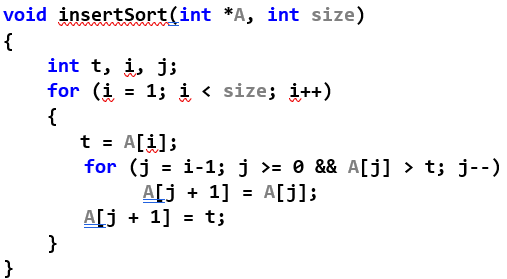
**Описание алгоритма.** Соседние элементы последовательности сравниваются между собой и, в случае надобности, меняются местами. Малоэффективен в больших массивах.  
**Реализация.**



**Анализ.** Малоэффективен в больших массивах.

# **85.** **Сортировка данных. Сортировка вставками (Insertion sort). Описание алгоритма. Реализация. Анализ алгоритма.**

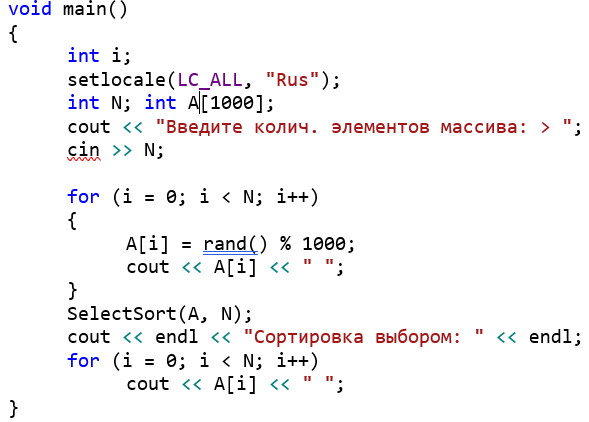
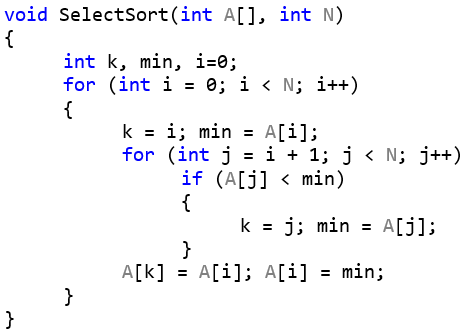
**Описание алгоритма.** Весь массив в процессе сортировки делится на две части: упорядоченную и неупорядоченную. Вначале весь массив не упорядочен. На каждом шаге из неупорядоченной части извлекается первый элемент, который вставляется на нужное место упорядоченной части. При этом размер упорядоченной части увеличивается на единицу. В конце весь массив окажется упорядоченным.



**Анализ.** Алгоритм эффективен на небольших наборах данных (на наборах данных до десятков элементов может оказаться лучшим) и на наборах данных, которые уже частично отсортированы. Это устойчивый алгоритм сортировки (не меняет порядок элементов, которые уже отсортированы). Ввиду своих особенностей хорош для списков.  Не требует дополнительной памяти.

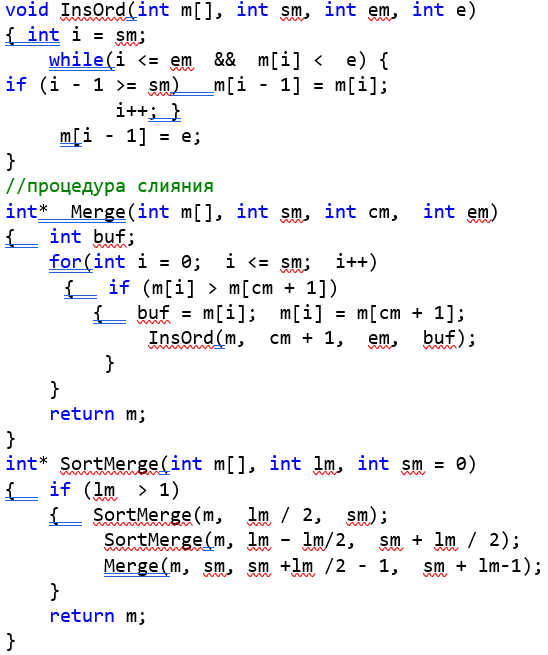
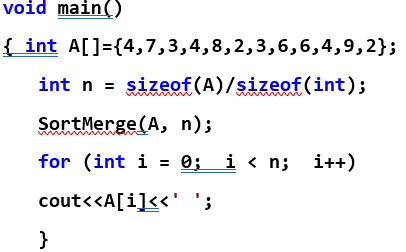
# **86.** **Сортировка данных. Сортировка выбором (Selection sort). Описание алгоритма. Реализация. Анализ алгоритма.**

**Описание алгоритма.** Вначале осуществляется поиск элемента, имеющего наименьшее значение. После того, как этот элемент найден, он меняется местами с 1-ым элементом. Затем, начиная со 2-го элемента массива, осуществляется поиск следующего наименьшего значения элемента. Найденный элемент меняется местами со 2-ым элементом. Этот процесс продолжается до тех пор, пока все числа не будут расположены в порядке возрастания.



# **87. Сортировка данных. Сортировка слиянием (Merge sort) [Рекурсивные алгоритмы]. Описание алгоритма. Реализация. Анализ алгоритма.**

**Описание алгоритма.** Исходный массив делится пополам, затем рекурсивно упорядочиваются обе половины и объединяются в одно целое. В ходе слияния элементы, стоящие в разных частях массива, попарно сравниваются друг с другом, и меньший элемент отправляется во временный массив. Этот процесс продолжается до тех пор, пока не будет использована полностью одна из двух частей массива. Затем копируются оставшиеся элементы во временный массив, и далее содержимое временного массива копируется обратно в исходный.

**Анализ.** Сортировка слиянием в среднем столь же эффективна, как и быстрая.**ы**

# **88.Сортировка данных. Быстрая сортировка (Quicksort) [Рекурсивные алгоритмы]. Описание алгоритма. Реализация. Анализ**

# 

**Сортировка** - это процесс упорядочения набора элементов в возрастающем или убывающем порядке.

Если в объектах содержится несколько данных-членов, нужно указать, какая переменная определяет порядок следования объектов. Эта переменная называется ключом.

Например, если хранится информация о людях, то ключом может быть возраст, фамилия и пр.

**Алгоритмы сортировки можно классифицировать по нескольким признакам.**

 По размещению элементов: внутренняя - в памяти, внешняя - в файле данных.

Методы внутренней сортировки можно разделить на две группы:

- методы, не требующие резерва памяти (метод выборки, "пузырька", вставки, Шелла);

- методы, требующие резерва памяти (метод квадратичной выборки, метод слияния и другие).

**QuickSort** является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как **«Пузырьковая сортировка» и «Шейкерная сортировка»**), известного, в том числе, своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы.

**Общая идея алгоритма состоит в следующем:**

* Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность (см.ниже).
* Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: «меньшие опорного», «равные» и «большие».[[1]](https://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0#cite_note-1)
* Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения

Алгоритм состоит из трёх шагов:

1. Выбрать элемент из массива. Назовём его опорным.
2. *Разбиение*: перераспределение элементов в массиве таким образом, что элементы меньше опорного помещаются перед ним, а больше или равные после.
3. Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

**Достоинства:**

* Один из самых быстродействующих (на практике) из алгоритмов внутренней сортировки общего назначения.
* Прост в реализации.
* Требует лишь {\displaystyle O(\log n)}дополнительной памяти для своей работы
* Хорошо сочетается с механизмами кэширования и виртуальной памяти.
* Допускает естественное распараллеливание
* Допускает эффективную модификацию для сортировки по нескольким
* Работает на связных списках и других структурах с последовательным доступом, допускающих эффективный проход как от начала к концу, так и от конца к началу.

**Недостатки:**

* Сильно деградирует по скорости в худшем или близком к нему случае, что может случиться при неудачных входных данных.
* Прямая реализация в виде функции с двумя рекурсивными вызовами может привести к ошибке переполнения стека, так как в худшем случае ей может потребоваться сделать {\displaystyle O(n)} вложенных рекурсивных вызовов.
* Неустойчив.

# **89. Сортировка данных. Сортировка подсчётом. Простой алгоритм. Алгоритм со списком. Алгоритм с двусвязным списком. Сортировка вставками.**

**Сортировка** - это процесс упорядочения набора элементов в возрастающем или убывающем порядке.

- сортировка **ВСТАВКАМИ** (простая вставка, вставка погружением, сортировка Шелла): очередной элемент помещается по месту своего расположения в выходную последовательность (массив);

- сортировка **ПОДСЧЕТОМ**: определяется количество элементов, больших или меньших данного элемента;

**Сортировка подсчетом**

Упорядоченный массив получается из исходного путем сравнения всех пар элементов массива.

Для **каждого** из элементов подсчитывается и запоминается **количество элементов,** которые **меньше** него. Это количество дает новое местоположение в выходном массиве.

**B = <20, -5, 10, 8, 7>**

**S = < 4,   0, 3, 2, 1>**

**B’ = <-5, 7, 8, 10, 20>**

Требуется **дополнительный** массив для размещения конечного результата и **дополнительный** временный массив для счетчиков.

**void CountSort(int in[], int out[], int n)**

**{ int i, j, cnt;**

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

**{ for ( cnt = 0, j = 0; j < n; ++j)**

**if (in[j]<in[i] || (in[j]==in[i] && i<j))**

**cnt++;** //счетчик чисел, больших текущего

**out[cnt] = in[i];**

**}**

**}**

Общее количество сравнений:

**(*n* –1 ) + (*n* – 2) +…+ 1 *= n* (*n* – 1) / 2 *≈ n*2 / 2**

**Алгоритм со списком**

Этот вариант используется, когда на вход подается массив структур данных, который следует отсортировать по ключам (key). Нужно создать вспомогательный массив C[0..k - 1], каждый C[i]в дальнейшем будет содержать список элементов из входного массива. Затем последовательно прочитать элементы входного массива A, каждый A[i]добавить в список C[A[i].key]. В заключении пройти по массиву C, для каждого {\displaystyle j\in \{0,...,k-1\}} в **j Э {0,…,k-1}** массив A последовательно записывать элементы списка C[j]. [Алгоритм устойчив](https://ru.wikipedia.org/wiki/%D0%A3%D1%81%D1%82%D0%BE%D0%B9%D1%87%D0%B8%D0%B2%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0).

ListCountingSort

for i = 0 to k – 1

C[i] = NULL;

for i = 0 to n - 1

C[A[i].key].add(A[i]);

b = 0;

for j = 0 to k - 1

p = C[j];

while p != NULL

A[b] = p.data;

p = p.next();

b = b + 1;

**Сортировка вставками**

Основным преимуществом алгоритма сортировки вставками является возможность сортировать массив по мере его получения. То есть, имея часть массива, можно начинать его сортировать.

Сортируемый массив надо разделить на две части — **отсортированная** часть и **неотсортированная**.

В начале сортировки первый элемент массива считается отсортированным, все остальные — не отсортированные.

Начиная со второго элемента массива и заканчивая последним, алгоритм вставляет неотсортированный элемент массива в **нужную** позицию в отсортированной части массива.

Таким образом, за один шаг сортировки отсортированная часть массива увеличивается на один элемент, а неотсортированная часть массива уменьшается на один элемент.

**void InsertSort(int A[], int n)** //сортировка вставками

**{ int t, i, j;**

**for (i = 1; i < n; i++)**

**{ t = A[i];**

**for (j = i-1; j >= 0 && A[j] >t; j--)**

**A[j + 1] = A[j];**

**A[j + 1] = t;**

**}**

**}**

Алгоритм эффективен на **небольших наборах данных** (на наборах данных до десятков элементов может оказаться лучшим) и на наборах данных, которые уже **частично** **отсортированы**.

# **90 Сортировка данных. Устойчивый алгоритм. Обобщение на произвольный целочисленный диапазон. Анализ. Квадратичный алгоритм сортировки подсчётом. Анализ.**

**Сортировка** - это процесс упорядочения набора элементов в возрастающем или убывающем порядке.

**Сортировка квадратичной выборкой**

Данный метод по сравнению с сортировкой выбором уменьшает число сравнений, но требует дополнительного объема памяти.

Сортируемый массив, состоящий из **n** элементов, разделяется на группы по **√n** элементов в каждой (если **n** не является точным квадратом, то таблица разделяется на **n'** групп, где **n'** - ближайший точный квадрат, больший **n)**.

В каждой группе выбирается наименьший элемент, который пересылается во вспомогательный список. Вспомогательный список просматривается и наименьший его элемент пересылается в зону вывода, где формируется отсортированный список.

Далее из групп выбираются новые наименьшие элементы, которые также помещаются во вспомогательный список. Новый наименьший элемент вспомогательного списка пересылается в зону вывода. Элементы групп, которые уже посланы во вспомогательный список, заменяются *фиктивными* величинами, заведомо большими, чем элементы исходного массива.

Так просматриваются попеременно, то вспомогательный список, то группы до тех пор, пока все элементы групп не будут исчерпаны.

Количество действий, требуемое для сортировки квадратичной выборкой, несколько меньше, чем в предыдущих методах и равно **n2**. Общее число сравнений равно приблизительно **2n\*sqrt(n)**, необходимый резерв памяти - поле длиной **(n+sqrt(n))** элементов.

# **91. Сортировка данных. Пирамидальная сортировка. Суть метода. Просеивание. Структура пирамиды. Построение пирамиды.**

**Сортировка** - это процесс упорядочения набора элементов в возрастающем или убывающем порядке.

- **ПИРАМИДАЛЬНАЯ** сортировка: строится пирамида, при сортировке выполняется просеивание пирамиды;

**Пирамидальная сортировка**

**Пи­ра­ми­да** — дво­ич­ное де­ре­во, в ко­то­ром зна­че­ние каж­до­го эле­мен­та боль­ше (меньше) **0** ли­бо рав­но зна­че­ниям до­чер­них эле­мен­тов.

За­пол­нив де­ре­во эле­мен­та­ми в про­из­воль­ном по­ряд­ке, мож­но его от­сор­ти­ро­вать, пре­вра­тив в пи­ра­ми­ду.

**Пирамида** – это последовательность **h1, h2  …hr** такая, что **h1 <= h2  <= … <= hr** для всякого **i = 1, …r / 2**.

Геометрическая интерпретация пирамиды:

**h1**

**/  \**

**h2             h3**

**/ \            / \**

**h4    h5     h6      h7**

**/  \  /  \    / \   /  \**

**h8 h9  h10 h11  h12 h13 h14 h15**

Например, для последовательности: **06 42 12 55 94 18 44**

**06**

**/  \**

**42          12**

**/   \          /   \**

**55    94    18   44**

При сортировке от­де­ля­ется вер­шин­ный эле­мент и за­пи­сы­ва­ется в ко­нец ре­зуль­ти­рую­ще­го мас­си­ва. На ме­сто вер­шин­но­го эле­мен­та помещается эле­мент из ниж­не­го уров­ня де­ре­ва. Вос­ста­нав­ли­вается (пе­ре­сор­ти­ро­вы­ва­ется) пи­ра­ми­да. Са­мый боль­шой эле­мент из остав­ших­ся элементов сно­ва в вер­ши­не. Он от­де­ля­ется и за­пи­сы­ва­ется в ка­че­стве пред­по­следнего эле­мен­та ре­зуль­та­та, и т. д.

**Фаза 1: построение пирамиды**

На каждом шаге добавляется новый элемент и  'просеивается' на свое место.

При добавлении:

1. Новый элемент **Х** помещается в вершину дерева.

2. Из элементов слева и справа выбирается  наи-меньший.

3. Если этот элемент меньше **Х**, то он меняется местами с **Х** и выполняется переход к п. 2. Иначе − конец первой фазы.

**Фаза 2: сортировка**

 Для того, чтобы отсортировать элементы, надо выполнить **n** шагов просеивания. После каждого шага очередной элемент берется с вершины пирамиды и помещается на свое место в результирующей последовательности.

**void Heapify (int A[], int pos, int n)**

**{ int t, tm;**

**while (2\*pos + 1 < n)**

**{ t = 2\*pos +1;**

**if (2\*pos+2<n && A[2\*pos+2] >= A[t])**

**t = 2\*pos + 2;**

**if (A[pos] < A[t])**

**{ tm = A[pos]; A[pos] = A[t];**

**A[t] = tm; pos = t;**

**}**

**else break;**

**}**

**}**

**void PiramSort(int A[], int n)**

**{ int tm;**

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

**Heapify(A, i, n);**

**while(n>0)**

**{ tm = A[0]; A[0] = A[n-1];**

**A[n-1] = tm; n--;**

**Heapify(A, 0, n);**

**}**

**}**

Преимущество ал­го­рит­ма в том, что пи­ра­ми­да хра­нит­ся пря­мо в ис­ход­ном мас­си­ве.

# **92.Сортировка данных. Поразрядная сортировка.**

**Сортировка** - это процесс упорядочения набора элементов в возрастающем или убывающем порядке.

**Поразрядная сортировка** — алгоритм сортировки, который выполняется за линейное время.

**Поразрядная (распределяющая) сортировка**

Основана на том, что все числа сортируются при помощи устойчивой сортировки сначала по младшему разряду, затем по остальным в порядке их возрастания. При этом ключи рассматриваются как числа, представленные в системе счисления с основанием **r**. Работа идет с отдельными цифрами чисел.

Предположим, что элементы исходного массива **A** − это Т-разрядные положительные десятичные числа.

**В(j, n)** – j-я справа цифра в десятичном числе **n >= 0** т.е. **В(j, n) = floor(n / m) % 10**, где **m = 10j - 1**.

Пусть **В0, В1…В9** – вспомогательные массивы, вначале пустые.

***Распределение*** заключается в том, что элемент **Кi (i = 1, … n)** из массива **В** добавляется как последний в массив **Вm**, где **m = В(j, Ki)**, и таким образом получаются десять массивов, в каждом из которых **j**-е разряды чисел одинаковы и равны **m**.

***Сборка*** объединяет массивы **В0, В1…В9** в этом же порядке, образуя один массив **В**.

**Пример**. Пусть есть последовательность натуральных чисел, каждое по **3 разряда**.

Распределяются числа в десять массивов независимо от того, какая в них последняя цифра.

Затем собираются все числа в одну последовательность.

Эти числа опять раскладываются по массивам, но теперь в зависимости от того, какая у них предпоследняя цифра.

Если теперь собрать все числа с очередей на выход, то последовательность окажется отсортированной в порядке возрастания.

**int VelichRazr(int chislo, int razr)**

**{ while(razr > 1)**

**{ chislo /= 10;**

**razr--;**

**}**

**return chislo % 10;**

**}**

**void PorazrSort(int B[n][n], int A[n], int razr)**

**{ int C[n], i, j, temp = 0;**

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

**C[i] = 0;**

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

**{ int a = VelichRazr(A[i], razr);**

**B[C[a]][a] = A[i];**

**C[a]++;**

**}**

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

**{ for(j = 0; j < C[i]; j++)**

**{ A[temp] = B[j][i];**

**temp++;**

**}**

**}**

**}**

Временная эффективность алгоритма – **Θ(T(n+k)),** где **T** – количество разрядов, **k** – диапазон значений разрядов.

**Недостаток** метода – использование дополнительной памяти.