## 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. Она задается своим именем, а операнды (формальные параметры) - в списке параметров через запятую, заключенном в круглые скобки. Результатом функции может быть возвращаемое значение. Об отсутствии возвращаемого значения сообщает ключевое слово void. Действия функции составляют её тело, заключенное в фигурные скобки. Тип возвращаемого значения, имя, список параметров и тело составляют определение функции. Пример:

void sum(int a, int b) {

cout << a + b << endl;

}

int main() {

setlocale(LC\_CTYPE, "rus");

int a, b;

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

cin >> a >> b;

sum(a, b);

return 0;

}

Прототип функции описывает её интерфейс и состоит из типа возвращаемого значения, имени и списка параметров. Тип может быть встроенным (int, double), составным (int&(ссылка), double\*(указатель)) или определенным пользователем (перечисление, класс). Пример:

#### bool look\_up(int\*, int);

#### double calc(double);

#### int count(const string&, char);

#### Date& calendar(const char);

#### void sum(vector<int>&, int);

Функция или встроенный массив не могут быть типом возвращаемого значения. Пример ошибочного кода:

int[10] foo\_bar(); // ошибка

Можно вернуть указатель на первый элемент массива:

int\* foo\_bar();

Список параметров не может быть опущен. Функция, не требующая параметров, должна иметь пустой список или список с void. Пример:

int fork();

int fork(void);

В списке параметров не разрешается использовать сокращенную запись:

int manip(int vl, v2); // ошибка

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

Имена параметров не могут повторяться. В объявлении функции имена параметров не обязательны и служат для документирования интерфейса. Пример:

void print(int \*array, int size);

Передача аргументов может происходить по ссылке или по значению. Ссылка - это второе имя того же объекта. При передаче по значению передается копия объекта. Пример:

void func\_1(int a) {

a = a + 100500;

}

void func\_2(int& a) {

a = a + 100500;

}

int main() {

int x = 12345;

func\_1(x);

cout << x; // 12345

func\_2(x);

cout << x; // 112845

}

В первом случае передается копия x, во втором - сам объект x.

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

Использование указателей для связи между функциями

I Variant

Мы только прикоснулись к обширному и увлекательному миру указателей. Сейчас нашей целью является использование указателей для решения задачи об установлении связи между функциями. Ниже приводится программа, в которой указатели служат средством, обеспечивающим правильную работу функции, которая осуществляет обмен значениями переменных.

#include<stdio.h>

void interchange(int \* ,int \* );

void main()

{

int x = 5, y = 10;

printf(" Начальные значения x = %d and y = %d.\n" , x, y);

interchange(&x,&y); /\* передача адресов функции \*/

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

}

void interchange(int \*u, int \*v) /\* u и v являются указателями \*/

{

int temp;

temp = \*u; /\* temp присваивается значение, на которое указывает u \*/

\*u = \*v;

\*v = temp;

}

Посмотрим, как она работает. Во-первых, теперь вызов функции выглядит следующим образом:

interchange(&x, &y);

Вместо передачи значений х и y мы передаем их адреса. Это означает, что формальные аргументы u и v, имеющиеся в спецификации, при обращении будут заменены адресами и, следовательно, они должны быть описаны как указатели. Поскольку х и у — целого типа, а u и v являются указателями на переменные целого типа, и мы вводим следующее описание:

int \*u, \*v;

Далее в теле функции оператор описания

int temp

используется с целью резервирования памяти. Мы хотим поместить значение переменной х в переменную temp, поэтому пишем

temp = \*u;

Вспомните, что значение переменной u — это &х, поэтому переменная и ссылается на х. Это означает, что операция \*u дает значение х, которое как раз нам и требуется. Мы не должны писать, например, так:

temp = u; /\* неправильно \*/

поскольку при этом происходит запоминание адреса переменной х, а не ее значения; мы же пытаемся осуществить обмен значениями, а не адресами.

Точно так же, желая присвоить переменной у значения переменной х, мы пользуемся оператором

\*u = \*v;

который соответствует оператору

х = у;

Подведем итоги. Нам требовалась функция, которая могла бы изменять значения переменных x и у. Путем передачи функции адресов переменных х и y мы предоставили ей возможность доступна к ним. Используя указатели и операцию \*, функция смогла извлечь величины, помещенные в соответствующие ячейки памяти, и поменять их местами.

Вообще говоря, при вызове функции информация о переменной может передаваться функции в двух видах. Если мы используем форму обращения

function1(х);

происходит передача значения переменной х. Если же мы используем форму обращения

function2(&x);

происходит передача адреса переменной х. Первая форма обращения требует, чтобы определение функции включало в себя формальный аргумент того же типа, что и х:

function1(num)

int num;

Вторая форма обращения требует, чтобы определение функции включало в себя формальный аргумент, являющийся указателем на объект соответствующего типа:

function2(ptr)

int \*ptr;

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. Разработаем программу, которая использует указатель на функцию. Программа должна находить НОД – наибольший общий делитель. НОД – это наибольшее целое число, на которое без остатка делятся два числа, введенных пользователем. Входные числа также должны быть целыми.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | // pointer\_onfunc.cpp: определяет точку входа для консольного приложения.    #include "stdafx.h"  #include <iostream>  using namespace std;  int nod(int, int ); // прототип указываемой функции  int main(int argc, char\* argv[])  {      int (\*ptrnod)(int, int); // объявление указателя на функцию      ptrnod=nod; // присваиваем адрес функции указателю ptrnod      int a, b;      cout << "Enter first number: ";      cin >> a;      cout << "Enter second number: ";      cin >> b;      cout << "NOD = " << ptrnod(a, b) << endl; // обращаемся к функции через указатель      system("pause");      return 0;  }  int nod(int number1, int number2) // рекурсивная функция нахождения наибольшего общего делителя НОД  {      if ( number2 == 0 ) //базовое решение          return number1;      return nod(number2, number1 % number2); // рекурсивное решение НОД  } |

Данная задача решена рекурсивно, чтоб уменьшить объём кода программы, по сравнению с итеративным решением этой же задачи. В строке 9 объявляется указатель, которому в строке 10 присвоили адрес функции. Как мы уже говорили до этого, адресом функции является просто её имя. То есть данный указатель теперь указывает на функцию nod(). При объявлении указателя на функцию ни в коем случае не забываем о скобочках, в которые заключаются символ указателя и его имя. При объявлении указателя в аргументах указываем то же самое, что и в прототипе указываемой функции. Результат работы программы (см. Рисунок 5).

CppStudio.com

Enter first number: 16

Enter second number: 20

NOD = 4

Для продолжения нажмите любую к

**Рекурсия**

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

int rgcd( int vl, int v2 )

{

if ( v2 != 0 )

return rgcd( v2, vl%v2 );

return vl;

}

Такая функция обязательно должна определять условие окончания, в противном случае рекурсия будет продолжаться бесконечно. Подобную ошибку так иногда и называют – бесконечная рекурсия. Для rgcd() условием окончания является равенство нулю остатка.  
Вызов rgcd( 15, 123 ); возвращает 3 (см. табл. 7.1).

**Таблица 7.1. Трассировка вызова rgcd (15,123)**

|  |  |  |
| --- | --- | --- |
| v1 | v2 | Return |
| 15 | 123 | rgcd(123,15) |
| 123 | 15 | rgcd(15,3) |
| 15 | 3 | rgcd(3,0) |
| 3 | 0 | 3 |

Последний вызов,

rgcd(3,0);

удовлетворяет условию окончания. Функция возвращает наибольший общий делитель, он же возвращается и каждым предшествующим вызовом. Говорят, что значение всплывает (percolates) вверх, пока управление не вернется в функцию, вызвавшую rgcd() в первый раз.  
Рекурсивные функции обычно выполняются медленнее, чем их нерекурсивные (итеративные) аналоги. Это связано с затратами времени на вызов функции. Однако, как правило, они компактнее и понятнее.  
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. Это делает функции языка Си несколько отличными от процедур Паскаля, поскольку процедуры в Паскале могут быть вложены в другие процедуры (причем, процедуры, содержащиеся в одном гнезде, являются недоступными для процедур, расположенных в другом).

     Нет ли у функции **main( )** какой-то специфики? Безусловно, есть; 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**

Во многих случаях полезно передавать данные в вызываемую функцию, чтобы она могла с ними как-то взаимодействовать. Например, если мы хотим написать функцию умножения двух чисел, то нам нужно сообщить функции при её вызове, что это будут за числа. В противном случае, как она узнает, что на что нужно умножать? Здесь нам на помощь приходят параметры и аргументы.

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | // Эта функция не имеет параметров  void doPrint()  {     scout << "In doPrint()" << std::endl;  }    // Эта функция имеет один параметр типа int - a  void printValue(int a)  {     scout << 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:

|  |  |
| --- | --- |
| 1  2 | printValue(7); // 7 – аргумент функции printValue()  add(4, 5); // 4 и 5 – аргументы функции add() |

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

# **Формальные и фактические параметры функции.**

Параметр при описании самой функции вне main() – это формальный параметр.

Аргументы, задаваемые при вызове функции, называются фактическими параметрами.

Список фактических параметров, как правило, должен в точности соответствовать списку формальных параметров, хотя в С++ возможны исключения.

Аргумент типа void

 аргумент типа void используется для обозначения того, что функция не принимает никаких параметров. В теле функции это указывается как void в списке параметров. Рассмотрим это подробнее.

библиотечная функция С sqrt()\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

using namespace std;

void voutput(void);

main()

{

/\* Программа определяет квадратный корень \*/

printf("This programm will find the square root\n\n\n");

voutput();

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

\_getch();

return(0);

}

void voutput(void)

{

double dt=12345.0;

double du;

du=sqrt(dt);

printf("The squre root of %lf is %lf\n",dt,du);

}

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'(8 бит).
* Расширенные символьные литералы типа wchar\_t, например L'a'(2 байта).
* 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

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. Примеры.

18.46

38.

Если указан показатель степени, он задает порядок числа в виде степени 10, как показано в следующем примере:

18.46e0 // 18.46

18.46e1 // 184.6

Показатель степени указывается с помощью символа **e** или **E**, которые имеют одинаковое значение, затем следует необязательный знак (+ или -) и последовательность цифр. Если указан показатель степени, десятичная точка в конце целых чисел не требуется, например 18E0.

По умолчанию литералы с плавающей запятой имеют тип **double(64 бита)**. С помощью суффиксов **f** или **l** (или **F** или **L** — регистр суффикса не учитывается) литерал можно опреде

-+

лить как **float (32 бита)** или long double соответственно.

Хотя long double и **double** имеют одинаковое представление, это разные типы. Например, можно задать приведенные ниже перегруженные функции

void func( double );

void func( long double );

**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 одномерного массива в функцию в C++ происходит по адресу, поэтому при вызове функции достаточно указать имя массива, что эквивалентно указанию адреса его первого элемента. Рассмотрим пример программы, которая вызывает функцию для вывода элементов массива.

#include <iostream>

using namespace std;

void print(int x[ ], int n); // Прототип функции

// или void print(int \*x, int n)

int main() {

const int n = 5;

int x[n] = {3, 5, 1, 7, 4}; // Инициализация массива

print(x, n); // Вызов функции print

return 0;

}

void print(int x[ ], int n) {

cout << "Массив:" << endl;

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

cout << x[i] << endl; // Вывод элементов массива

}

}

**Аргументы по умолчанию.**

В C++ можно задать аргументы по умолчанию для функции, что позволяет при вызове функции опускать некоторые её аргументы. Эти аргументы должны быть указаны в прототипе функции. Если несколько аргументов имеют значения по умолчанию, они должны идти справа от остальных. Опуская аргументы, необходимо соблюдать порядок: если пропущен первый параметр, все остальные тоже должны быть пропущены. Если пропущен параметр в середине, то все последующие тоже должны быть пропущены. Рассмотрим пример для лучшего понимания.

double heron\_space(const double a = 5, const double b = 6.5, const double c = 10.7); //параметры функции инициализированы по умолчанию

double heron\_space(const double a, const double b, const double c) // функция

{

    const double p = (a + b + c) / 2;   // полупериметр

    cout << "a = " << a << "\nb = " << b << "\nc = " << c << endl;

    return (sqrt(p \* (p - a) \* (p - b) \* (p - c))); // формула Герона для нахождения площади треугольника

}

# **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**. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | #include <iostream>    void factorial(int);    int main()  {      factorial(-3);      factorial(5);      factorial(4);      return 0;  }    void factorial(int n)  {      if(n<1)      {          std::cout << "Incorrect number" << std::endl;          return;      }      int result = 1;      for(int i = 1; i <=n; i++)      {          result \*= i;      }      std::cout << "Factorial of " << n << " is equal to " << result << std::endl;  } |

В данном случае функция factorial вычисляет факториал переданного числа. Однако если число меньше 1, то функция выводит соответствующее сообщение, и с помощью оператора return осуществляется выход из функции.

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;

Функции типа void

Если ключевое слово void указывает возвращаемый тип функции, оно означает, что данная функция не возвращает никакого значения.

#include <stdio.h>

void printMessage() {

printf("Hello, World!\n");

}

int main() {

printMessage(); // Вызов функции, которая ничего не возвращает

return 0;

}

Функции типа char

Тип char был исходным символьным типом в C и C++. Он может использоваться для хранения символов. Тип char не подходит для текста во многих языках. В современных программах для текста обычно используется один из типов с расширенными символами.

#include <iostream>

#include <cctype> // Для функции tolower

using namespace std;

char clowercase(char c);

int main() {

char upperChar, lowerChar;

cout << "Enter an uppercase character: ";

cin >> upperChar;

lowerChar = clowercase(upperChar);

cout << "Entering character: " << upperChar << "\n";

cout << "Converting character: " << lowerChar << "\n";

return 0;

}

char clowercase(char c) {

return tolower(c);

}

Функции типа int

Функции типа int возвращает данные типа int.  Следующая функция принимает от вызывающего ее объекта два целых числа и возвращает их сумму. a и b — параметры типа int.

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

Функция считывает по очереди целые числа и возвращает

их значения, возведенные в куб\*/

#include <iostream>

#include <conio.h>

#include <stdio.h>

using namespace std;

int icube(int ivalue);

main()

{

int k,inumbercube;

for(k=0;k<20;k+=2)

{

inumbercube=icube(k);

/\* Куб числа ... равен ... \*/

printf("The cube of the number %2d is %d\n",k,inumbercube);

}

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

\_getch();

return(0);

}

int icube(int ivalue)

{

return(ivalue\*ivalue\*ivalue);

}

Функции типа long

Функции типа long в C и C++ — это функции, которые возвращают значение типа long. Тип long используется для хранения целых чисел с большим диапазоном, чем тип int. В C и C++ функции могут возвращать значения различных типов, и функции, которые возвращают значение типа long, называются функциями типа long.

#include <iostream>

using namespace std;

// Определение функции, которая возвращает значение типа long

long lpower(int ivalue) {

long lseed = 1;

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

lseed \*= 2;

}

return lseed; // Возвращаем значение типа long

}

int main() {

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

long lanswer = lpower(k); // Вызов функции типа long

cout << "2 raised to the " << k << " power is " << lanswer << endl;

}

return 0;

}

Функции типа float

В следующем примере массив типа float передается в качестве параметра в функцию, которая возвращает значение типа float.

#include <iostream>

using namespace std;

float calculateProduct(float farray[], int size);

int main() {

float fmyarray[5] = {4.3, 1.8, 6.12, 3.19, 0.1};

int arraySize = 5;

float product = calculateProduct(fmyarray, arraySize);

cout << "The product of all array elements is: " << product << endl;

return 0;

}

float calculateProduct(float farray[], int size) {

float result = 1.0f; // Начальное значение для произведения

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

result \*= farray[i];

}

return result;

}

Функции типа double

В следующем примере на С передаются и возвращаются значения типа double.

#include <iostream>

#include <cmath> // Для функции cos

using namespace std;

double calculateCosine(double angle);

int main() {

double angle;

cout << "Enter an angle in radians: ";

cin >> angle;

double cosine = calculateCosine(angle);

cout << "The cosine of " << angle << " radians is " << cosine << endl;

return 0;

}

double calculateCosine(double angle) {

return cos(angle);

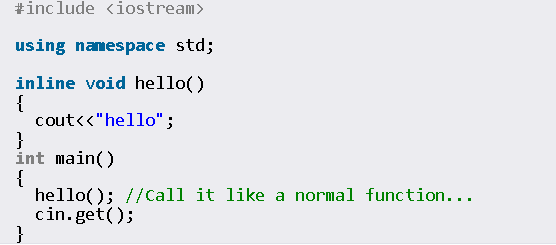
}

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

Встраивание(inline)

Кроме обычных функций в C++, о которых вы уже знаете, есть еще встроенные функции. Встроенные функции не так значимы, но желательно в них разбираться. Основная идея в том, чтобы ускорить программу ценой занимаемого места. 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.

Как это делает программу быстрее? Легко, вызовы функций занимают больше времени, чем написание всего кода без функции. Просмотр вашей программы и замена функции, которую вы использовали 100 раз с кодом из функции, займет очень много времени. Конечно, используя встроенные функции для замены обычных вызовов функций, вы также значительно увеличите размер вашей программы.



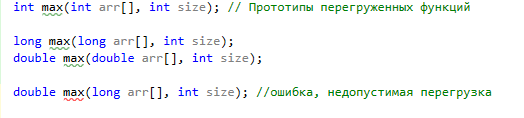
Однако, как только программа будет скомпилирована, вызов hello(); будет заменен на код функции.

Встроенные функции очень хороши для ускорения программы, но, если вы используете их слишком часто или с большими функциями, у вас будет чрезвычайно 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.

Наконец, обратите внимание, что компилятор может, по своему желанию, игнорировать ваши попытки сделать функцию встроенной. Так что, если вы ошибетесь и сделаете встроенной чудовищную функцию в пятьдесят строк, которая вызывается тысячи раз, компилятор может игнорировать вас.

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 :



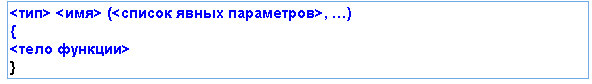
Приведённые функции разделяют одно общее имя, но разные списки параметров. Перегруженные функции можно различать по наличию параметров. Функции не могут быть перегружены, если описание их параметров отличается только модификатором *const* или наличием ссылки. Разные типы возвращаемых значений не могут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

Фактически все функции (не только перегруженные) должны иметь уникальные сигнатуры, определяемые именем и списком параметров, иначе программа не компилируется.

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:



Здесь:

<тип> - тип возвращаемого значения,

<список параметров> - список параметров известных до вызова функции.

После списка явных параметров следует необязательная запятая и троеточие, которое сообщает компилятору, что контроль типов и количества параметров при вызове функции не следует.

Когда говорят о специальном наборе макроопределений в контексте функций с переменным числом параметров в C++, речь идет об использовании макросов из стандартной библиотеки <cstdarg>. Этот набор макроопределений облегчает работу с переменным числом аргументов, передаваемых в функцию.

Вот некоторые из основных макросов из <cstdarg>:

1. va\_list: Это тип данных, который представляет список аргументов переменной длины. Объявляется переменная типа va\_list, которая используется для доступа к аргументам.
2. va\_start: Этот макрос инициализирует объект va\_list. Он принимает два аргумента: первый - имя объекта va\_list, второй - имя последнего аргумента, известного функции.
3. va\_arg: Этот макрос извлекает следующий аргумент из списка аргументов переменной длины. Он принимает два аргумента: первый - объект va\_list, второй - тип аргумента, который нужно извлечь из списка.
4. va\_end: Этот макрос завершает работу с объектом va\_list. Он должен быть вызван после того, как все аргументы были извлечены из списка.

С помощью этих макросов можно создавать функции, которые принимают переменное число аргументов и работать с ними в функции. Этот специальный набор макросов обеспечивает переносимость и согласованность в работе с аргументами переменной длины между различными компиляторами и платформами.

#include <iostream>

#include <cstdarg>

using namespace std;

// Функция, которая принимает переменное число аргументов и возвращает их сумму

int sum(int count, ...) {

va\_list args; // Объявляем объект va\_list для работы с аргументами переменного числа

va\_start(args, count); // Инициализируем объект va\_list передаваемыми аргументами

int total = 0;

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

total += va\_arg(args, int); // Извлекаем аргументы типа int из списка

}

va\_end(args); // Завершаем работу с объектом va\_list

return total;

}

int main() {

// Вызов функции с разным числом аргументов

cout << "Sum of 3, 5, and 10: " << sum(3, 3, 5, 10) << endl;

cout << "Sum of 4, 7, 2, and 9: " << sum(4, 4, 7, 2, 9) << endl;

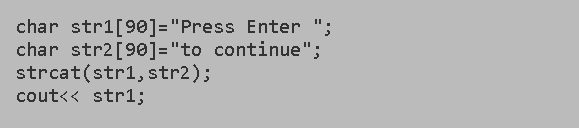
return 0;

}

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

|  |  |
| --- | --- |
| **Объединение строк** | |
| **strcat(**s1,s2**)** | объединяет строку s2 со строкой s1. Результат сохраняется в s1 |

Рассмотрим пример:

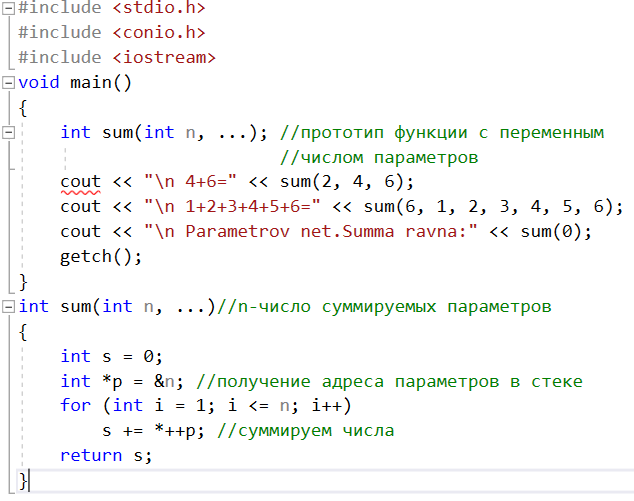


В результате на экран выведется и в строке str1 будет записано: Press Enter to continue

**Функция strncat**

|  |  |
| --- | --- |
| **strncat(**s1,s2,n**)** | объединяет n символов строки s2 со строкой s1. Результат сохраняется в s1 |

**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.



Для доступа к параметрам, которые один за другим попали в стек, используется адрес первого параметра **(\*p = &n;)**, таким образом, указатель устанавливается на начало списка параметров в стеке (в памяти). Затем в цикле при помощи указателя p перемещаемся по параметрам и суммируем их, извлекая **(\*p)** из памяти. Все параметры должны иметь **одинаковый тип**.

Проверка соответствия типов для дополнительных параметров не выполняется, поскольку компилятор не имеет информации, необходимой для проведения проверки. При вызове функции дополнительные параметры типа char и short передаются как **int** , а float — как **double**.

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

Когда программа вызывается из командной строки операционной системы, ей можно передать аргументы, которые будут доступны внутри программы для обработки. Эти аргументы передаются функции main() в виде параметров.

В языке C и C++ функция main() имеет два параметра:

1. argc (аргументы счетчика) - это целое число, которое представляет количество аргументов, переданных программе из командной строки, включая имя самой программы.
2. argv (аргументы значения) - это массив строк (массив указателей на строки), который содержит сами аргументы, переданные программе. Первый элемент этого массива (argv[0]) обычно содержит имя самой программы, а остальные элементы (argv[1], argv[2], и т.д.) содержат аргументы, переданные из командной строки.

Например, если программа была вызвана следующим образом:

./program arg1 arg2 arg3

то argc будет равно 4 (включая имя программы), а argv будет массивом:

argv[0]: "./program"

argv[1]: "arg1"

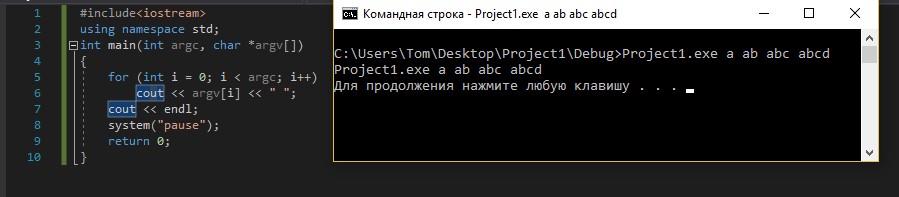
argv[2]: "arg2"

argv[3]: "arg3"

Таким образом, программа может использовать argc и argv для обработки аргументов, переданных из командной строки, и выполнения соответствующих действий в зависимости от этих аргументов.

**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().**

int argv[0] = atoi(argv[0]); // Преобразуем строку в целое число

**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().**

#include <stdio.h>

#include <stdlib.h> // Для использования atof()

#include <math.h> // Для использования cos()

int main(int argc, char \*argv[]) {

// Перебираем аргументы, начиная с первого (argv[1])

for (int i = 1; i < argc; i++) {

// Преобразование строкового аргумента в число с плавающей точкой

double angle = atof(argv[i]);

// Вычисление косинуса угла

double cosine = cos(angle \* M\_PI / 180.0); // Переводим угол в радианы

// Вывод результата

printf("Cosine of angle %s is: %.4f\n", argv[i], cosine);

}

return 0;

}

# **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++.**

Область видимости в C++ определяет, где в программе можно использовать определенные имена. Есть три основных типа области видимости: локальная, пространства имен и область видимости класса.

1. Локальная область видимости: Это часть кода программы, которая находится внутри определения функции или блока. Каждая функция имеет свою собственную локальную область видимости, и каждый блок внутри функции также создает свою локальную область. Имя переменной в локальной области видимости доступно только внутри этой функции или блока.

void myFunction() {

int localVar = 10; // Локальная переменная

std::cout << "Local variable inside myFunction: " << localVar << std::endl;

}

1. Область видимости пространства имен: Это часть программы, которая не находится внутри определения функции, класса или пространства имен. Самая внешняя часть программы также является областью видимости пространства имен, называемой глобальной областью видимости. Здесь могут быть определены объекты, функции, типы и прочее. Программист может создать пользовательские пространства имен для организации кода и изоляции имен. Переменные и функции, объявленные внутри пространства имен, доступны в пределах этого пространства имен.

#include <iostream>

namespace MyNamespace {

int globalVar = 20; // Глобальная переменная в пространстве имен

void myFunction() {

std::cout << "Global variable inside myFunction: " << globalVar << std::endl;

}

}

int main() {

MyNamespace::myFunction();

std::cout << "Accessing global variable from main: " << MyNamespace::globalVar << std::endl;

return 0;

}

1. Область видимости класса: Каждое определение класса создает свою собственную область видимости. Имя, объявленное внутри класса, доступно внутри этого класса и его методов.

#include <iostream>

class MyClass {

public:

int classVar = 30; // Переменная класса

void myMethod() {

std::cout << "Class variable inside myMethod: " << classVar << std::endl;

}

};

int main() {

MyClass obj;

obj.myMethod();

std::cout << "Accessing class variable from main: " << obj.classVar << std::endl;

return 0;

}

Каждая область видимости в C++ имеет свои правила разрешения имен, определяющие, какие имена можно использовать в каких частях программы. Разрешение имени начинается с самой близкой области и распространяется к внешним областям, если имя не найдено в текущей. Если имя не может быть найдено в любой области видимости, оно считается ошибочным.

Изменения внешней области видимости могут скрыть имена из внутренних областей, так что имена из внутренних областей могут быть перекрыты именами из внешних областей. Это может привести к неожиданному поведению программы.

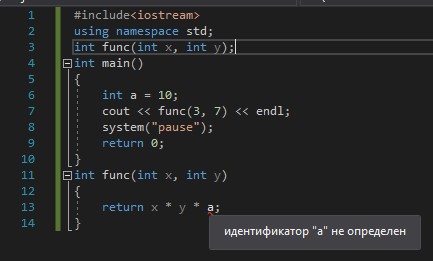
В C++ также можно объявлять переменные внутри управляющих конструкций, таких как циклы и условные операторы. Переменные, объявленные внутри таких конструкций, доступны только внутри этих конструкций и становятся недоступными после их завершения.

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

Если используются переменные с различной областью действия, то можно столкнуться с совершенно неожиданными результатами программирования, называемыми побочными эффектами. Например, как вы уже знаете, может существовать переменная (вернее две переменные с одинаковым именем) как с файловой, так и с локальной областью действия. Правила области действия констатируют, что переменная с локальной областью действия (называемая локальной переменной) имеет приоритет по сравнению с переменной с файловой областью действия (называемой глобальной переменной). Хотя это выглядит достаточно просто, давайте рассмотрим несколько проблем, которые не так очевидны и с которыми вы можете столкнуться при программировании.

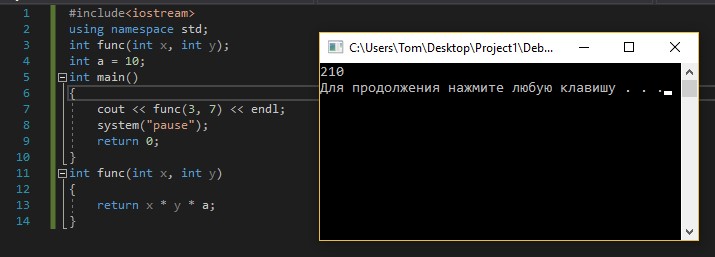
**Неопределенные символы в программе на С (Пример на C++).**

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



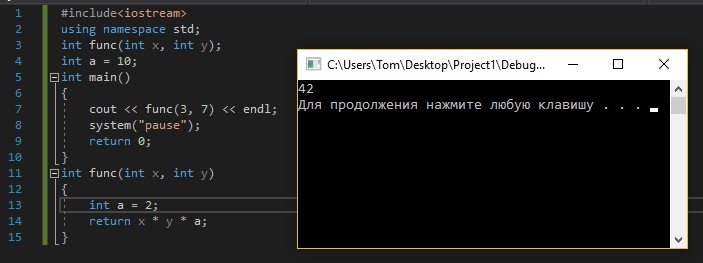
**Использование переменной с файловой областью действия.**

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



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

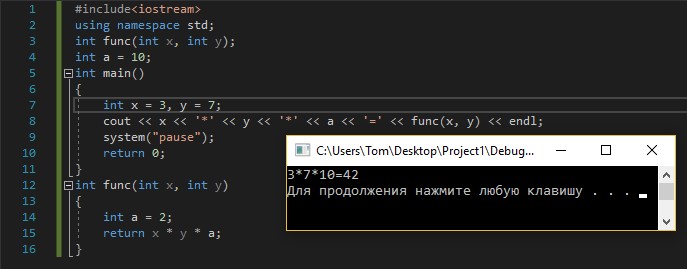
Правила области действия констатируют, что у переменной, имеющей как локальную, так и файловую область действия, используется ее локальное, а не глобальное значение. Это положение иллюстрирует следующая небольшая программа:



**Проблемы области действия в C++**.

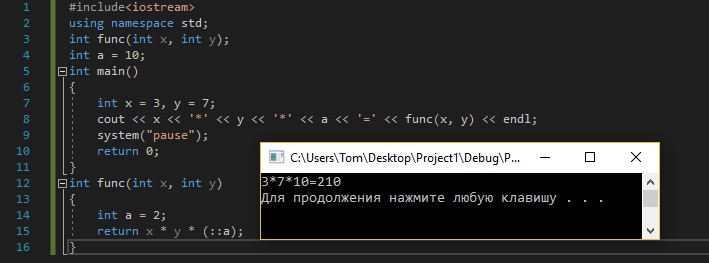
Следующий пример на **C++** работает нормально до момента вывода информации на экран.

Оператор **cout** правильно распечатывает значения переменных **x** и **y**. Но при обращении к переменной a он выбирает глобальную переменную с файловой областью действия. Результат, выдаваемый программой, **3\*7\*10 = 42,** является явной ошибкой. Как вы знаете, в подобной ситуации функция **func()** использует локальное значение переменной a.



**Операция уточнения области действия в C++.**

В следующем примере для того, чтобы избежать конфликтов между переменными как с файловой, так и с локальной областями действия, используется операция уточнения области действия **(scope resolution operator ::).**



**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 func(int x, int y) { } // описание самой функции
2. void main{
3. pfunc();
4. }

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

Во третьей строчке определяется переменная **pfunc**, которая является указателем на функцию **func**. В определении **pfunc** ей присваивается адрес функции **func.**

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

**Указатели на методы и члены данных.**

Рассмотрим следующий пример:

#include <iostream>

using namespace std;

struct Human {

int age;

char name;

Human(int x, char y) : age(x), name(y) {}

int work(){

return ++(age);

}

};

int main()

{

Human first(15, 'S');

int (Human:: \*ptrW)() = &Human::work; // создание ссылки на метод

char& ptrN = first.name; // создание ссылки на член структуры

cout << (first.\*ptrW)() << endl;

cout << ptrN;

return 0;

}

**Динамическое связывание функций.**

Указатели на функции в языке программирования C (и C++) играют ключевую роль в системном программировании и динамическом связывании функций. При статических способах, функции загружаются на уровне компиляции программы (тоесть загружаются до её выполнения), в динамических – загрузка функций происходит условно локально (при её вызове). Чтобы лучше понять, о чем идет речь, давайте разберем основные концепции и их применение.

**Основные виды связывания функций**

1. *Статическое связывание в одном модуле*

Если определение функции и ее вызов находятся в одном модуле (файле), компилятор может вычислить относительный адрес функции во время компиляции. Это называется статическим связыванием, потому что адрес функции определяется на этапе компиляции и остается неизменным.

#include <stdio.h>

void sayHello() {

printf("Hello!\n");

}

int main() {

sayHello();

return 0;

}

1. *Статическое связывание между модулями*

Когда вызов функции и ее определение находятся в разных модулях (файлах), адрес функции определяется во время компоновки программы из объектных модулей. Это связывание также называется статическим, хотя и происходит на этапе компоновки.

// foo.cpp

void foo() {

// some code

}

// main.cpp

extern void foo(); // экспорт функкции

int main() {

foo(); // компоновщик определяет адрес функции

return 0;

}

1. *Динамическое связывание с использованием DLL*

При использовании динамически связываемых библиотек (DLL), функции из этих библиотек связываются с основной программой во время загрузки программы в память. Это связывание уже считается динамическим, так как адреса функций определяются только при загрузке программы.

#include <iostream>

// Базовый класс Animal

class Animal {

public:

// Виртуальная функция sound

virtual void sound() const {

std::cout << "Some generic animal sound" << std::endl;

}

// Виртуальный деструктор

virtual ~Animal() = default;

};

// Производный класс Dog

class Dog : public Animal {

public:

void sound() const override {

std::cout << "Woof!" << std::endl;

}

};

// Производный класс Cat

class Cat : public Animal {

public:

void sound() const override {

std::cout << "Meow!" << std::endl;

}

};

// Функция, которая принимает ссылку на базовый класс Animal

void makeSound(const Animal& animal) {

animal.sound();

}

int main() {

Dog dog;

Cat cat;

// Вызываем makeSound с объектами разных типов

makeSound(dog); // Должно вызвать Dog::sound()

makeSound(cat); // Должно вызвать Cat::sound()

return 0;

}

Виртуальные функции:

1. Переопределение: Виртуальная функция может быть переопределена в производном классе с тем же именем и сигнатурой, а обычная функция не может.
2. Позднее связывание: Вызов виртуальной функции определяется типом объекта во время выполнения программы, а не во время компиляции, что позволяет использовать функцию, определенную в производном классе, даже если через указатель или ссылку используется базовый класс.
3. Полиморфизм: Виртуальные функции обеспечивают механизм полиморфизма, позволяя объектам разных типов использовать один и тот же интерфейс для вызова функций.

**Таблицы функций, вызов по имени**

Тип данных вида **void (\*pp[])()** расшифровывается в соответствии с контекстным определением типа данных как массив указателей на функции с общим прототипом (схемой передачи параметров и результата) – 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).

В самом простом примере наличие таблицы функций позволяет сделать вызов функции по заданному символьному имени регулярным (**циклическим**). В принципе, то же самое можно сделать с помощью обычного переключателя (**switch)**, но для этого придется еа каждую новую функцию добавлять фрагмент программного кода. Здесь же достаточно внести в массивы символьное имя и ее адрес.

extern double sin(double);

extern double cos(double);

extern double tan(double);

char \*names[] = { "sin","cos","tan",NULL}; // Массив имен (указатели на строки)

double (\*pf[])(double) = { sin, cos, tan}; // Таблица функций (адреса функций)

//-------------------------------------------------------------------------------------93-01.cpp

//---- Вызов функции по имени из заданного списка

double call\_by\_name(char \*pn, double arg) {

for ( int i=0; names[i]!=NULL; i++)

if (strcmp(names[i],pn) == 0) { // Имя найдено -

return ((\*pf[i])(arg)); // вызов функции по i-му

} // указателю в массиве

pf return 0.;

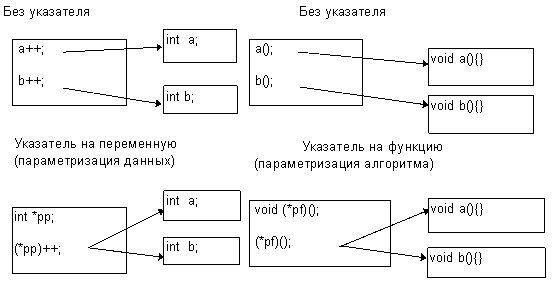
}

В принципе, используя технику работы с двоичными файлами и динамические массивы, можно было бы выполнить динамическую загрузку функций из программных файлов. Однако эта задача является архитектурно зависимой и требует учета особенностей перемещения программного кода и защиты памяти программы в операционной системе.

**Указатель на функцию как средство параметриза**

**ции алгоритма**

Оригинальность и обособленность указателя на функцию заключается в том, что указуемым объектом является не переменная (компонента данных программы), а функция (компонента алгоритма). Но сущность указателя при этом не меняется: 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).



**рис. 93-1. Параметризация алгоритма**

Для реализации указанного принципа основная функция должна получать необходимый для ее работы «**довесок**» в виде формального параметра – **указателя на функцию**. Частный пример подобного рода – сортировка одной и той же структуры данных по разным критериям. В массиве структур типа **user** для задания способа сравнения используется указатель на внешнюю функцию сравнения, которая в качестве параметров получает две ссылки на сравниваемые структуры.

//------------------------------------------------------93-03.cpp

//------Сортировка массива структур по разным критериям struct user{ char name[20]; int account;

double time;

} S[]={…};

//-------- Вставка погружением

// cmp - указатель на функцию сравнения двух struct user

void sort(user A[], int (\*cmp)(user&,user&)){ for (int i=1;A[i].name[0]!=0;i++) for(int j=i; j>0 && (\*cmp)(A[j],A[j-1])<0; j--)

{ user c=A[j];

A[j]=A[j-1]; // Обмен элементов массива - A[j-1]=c; } // структурированных переменных

}

//--------- функция сравнения по имени int cmp\_name(user &u1, user &u2){ return strcmp(u1.name,u2.name); } //--------- функция сравнения по account int cmp\_account(user &u1, user &u2){ return u1.account - u2.account; } //--------- функция сравнения по time int cmp\_time(user &u1, user &u2){

return u1.time - u2.time; }

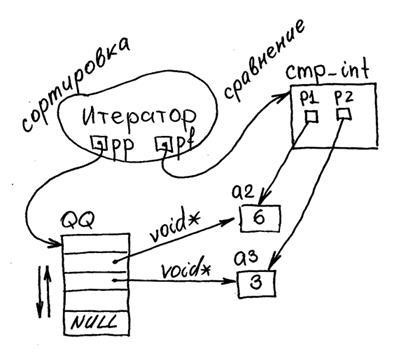
void main(){ sort(S,cmp\_name); sort(S,cmp\_account); sort(S,cmp\_time); }

Этот пример можно обобщить на случай использования данных любого типа. В Си отсутствует понятие произвольный (неопределенный) тип, но понятие указатель на неопределенный (произвольный) тип существует – это **void\*.** Однако прямое использование этого указателя невозможно. Какой же здесь выход?

Если существуют несколько идентичных структур данных, отличающихся типом хранимых данных, то один и тот же алгоритм будет отличаться только отдельными операциями, касающимися переменных этого типа. Например, при сортировке массива указателей на строки сравнение реализуется выражением вида **strcmp(p[i],p[i-1]),** а в массиве указателей на целые используется непосредственное сравнение **\*p[i]<\*p[i-1].** Если эту операцию вынести за пределы алгоритма, реализовать отдельной функцией, а указатель на нее передавать в качестве параметра, то мы получим универсальную функцию сортировки массивов указателей на переменные любого типа данных. Можно назвать ее условно (хотя и не совсем точно) **итератором**.

Типичными итераторами являются:

* итератор обхода (**foreach**), выполняющий для каждой переменной в структуре данных указанную функцию;
* итераторы проверки и поиска по условию (**firstthat**), возвращающие указатель на первую (последнюю, очередную) переменную, которая удовлетворяет условию, проверяемому в функции;
* итераторы, работающие на упорядоченных данных, использующие сравнение: сортировка, поиск минимального, двоичный поиск, включение с сохранением порядка.



**Рис. 93-2. Структура итератора**

Итератор являет собой пример универсализации алгоритма, основанный на разделении «компетенций» между различенными его частями:

* структура данных, обрабатываемая итератором, содержит в своих элементах указатели на переменные произвольного (неизвестного для итератора) типа **void\*,** но одинакового в каждом экземпляре структуры данных. Структура данных «знает», где хранятся переменные, но «не знает», как их обработать;
* итератор получает в качестве параметров указатель на структуру данных и указатель на функцию обработки. Эта функция «знает», как обработать переменные, но «не знает», где их взять;
* итератор выполняет алгоритм обработки структуры данных в соответствии со своим назначением: **foreach** - обходит все переменные, **firstthat** - обходит и проверяет все переменные, итератор сортировки -сортирует указатели на хранимые объекты (или соответствующие элементы структуры данных, например, элементы списка);
* действие, которое надлежит выполнить над хранимыми объектами произвольного типа, реализуется вызовом внешней функции через указатель. Итератор извлекает из структуры данных указатели **void\*** на хранимые данные и передает их внешней функции.
* прототип (схема формальных параметров и результата) внешней функции различна для разных типов итераторов. Для **foreach** – указатель имеет вид  **void (\*pf)(void\*),** для**firstthat** – **int (\*pf)(void\*)**, для итераторов, использующих сравнение – **int (\*cmp)(void\*,void\*)**.

//------------------------------------------------------93-04.cpp

//----- Итераторы foreach, firstthat и поиска минимального для списка struct list { list \*next; // Указатель на следующий

void \*pdata; }; // Указатель на данные

//----- Итератор: для каждого элемента списка void ForEach(list \*pv, void (\*pf)(void\*) ) { for (; pv !=NULL; pv = pv->next)

(\*pf)(pv->pdata);

}

//----- Итератор: поиск первого в списке по условию

void \*FirstThat(list \*pv, int (\*pf)(void\*)) { for (; pv !=NULL; pv = pv->next) if ((\*pf)(pv->pdata)) return pv ->pdata; return NULL; }

//----- Итератор: поиск минимального в списке void \*FindMin(list \*pv, int (\*pf)(void\* ,void\*))

{ list \*pmin;

for ( pmin=pv; pv !=NULL; pv = pv->next) if ((\*pf)(pv->pdata ,pmin->pdata) <0) pmin=pv; return pmin; }

//----- Примеры использования итератора ------------------

//----- Функция вывода строки void print(void \*p) { puts((char\*)p); } //----- Функция проверки : длины строки >5 int bigstr(void \*p) { return strlen((char\*)p ) > 5; } //----- Функция сравнения строк по длине int scmp(void \*p1, void \*p2)

{ return strlen((char\*)p1)- strlen((char\*)p2); }

//----- Вызов итераторов для статического списка,

// содержащего указатели на строки

list a1={NULL,"aaaa"}, a2={&a1,"bbbbbb"}, a3={&a2,"ccccc"}, \*PH=&a3;

//----- Итератор сортировки для массива указателей

void Sort(void \*\*pp, int (\*pf)(void\*,void\*))

{ int i,k; do

for (k=0,i=1; pp[i] !=NULL; i++) if ( (\*pf)(pp[i-1],pp[i])>=0) // вызов функции сравнения { void \*q; // перестановка указателей k++; q = pp[i-1]; pp[i-1] = pp[i]; pp[i] = q;

} while(k); }

// Пример вызова итератора сортировки для массива

// указателей на целые переменные int cmp\_int(void \*p1, void \*p2) { return \*(int\*)p1-\*(int\*)p2; } int b1=5, b2=6, b3=3, b4=2; void \*PP[] = {&b1, &b2, &b3, &b4, NULL}; void main()

{ char \*pp; ForEach(PH,print);

pp = (char\*) FirstThat(PH,bigstr); if (pp !=NULL) puts(pp); pp = (char\*) FindMin(PH,scmp); if (pp !=NULL) puts(pp); Sort(PP,cmp\_int);

for (int i=0; PP[i]!=NULL;i++) printf("%d ",\*(int\*)PP[i]); puts("");}

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

**Массив указателей на функции**

Массив указателей на функции - это массив, элементы которого являются указателями на функции. Он позволяет хранить адреса функций в памяти и обращаться к ним по индексу массива.

Вот пример использования массива указателей на функции на языке C++:

#include <iostream>

// Объявление функций

void func1() {

std::cout << "Вызвана функция 1" << std::endl;

}

void func2() {

std::cout << "Вызвана функция 2" << std::endl;

}

void func3() {

std::cout << "Вызвана функция 3" << std::endl;

}

int main() {

// Объявление массива указателей на функции

void (\*funcPtrArray[])() = { func1, func2, func3 };

// Вызов функций через массив указателей

for (int i = 0; i < 3; ++i) {

funcPtrArray[i](); // Вызываем функцию, на которую указывает i-й элемент массива

}

return 0;

}

Этот подход позволяет нам создавать массивы функций и обращаться к ним динамически во время выполнения программы.

**Шаблоны функций**

Шаблоны функций в C++ - это специальные инструкции, которые позволяют создавать функции, работающие с разными типами данных, без необходимости писать отдельные версии функций для каждого типа данных.

#include <iostream>

using namespace std;

// Шаблон функции max

template <typename T>

T max(T a, T b) {

return (a > b) ? a : b;

}

int main() {

// Вызов шаблонной функции max для разных типов данных

cout << "Максимум между 5 и 10: " << max(5, 10) << endl;

cout << "Максимум между 3.5 и 2.8: " << max(3.5, 2.8) << endl;

cout << "Максимум между 'a' и 'z': " << max('a', 'z') << endl;

return 0;

}

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

Функции round, roundf и roundl используются для округления чисел с плавающей точкой до ближайшего целого числа. Если дробная часть числа находится точно посередине между двумя целыми числами, то результат округления будет к ближайшему четному числу.

**Синтаксис:**

#include <cmath>

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 round, roundf и roundl принимают один аргумент типа double, float и long double соответственно. Возвращаемое значение также является числом с плавающей точкой того же типа, что и входной аргумент.

Если дробная часть числа находится на полпути между двумя целыми числами, результат округления зависит от четности самого числа: оно будет округлено к ближайшему четному.

*Пример:*

round(3.5) вернет 4.0, так как 3.5 находится на полпути между 3 и 4, и ближайшим четным числом является 4.

round(-2.5) вернет -2.0, так как -2.5 находится на полпути между -2 и -3, и ближайшим четным числом является -2.

**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. Все переменные, определённые внутри функции, являются автоматическими по умолчанию, и поэтому ключевое слово auto используется крайне редко. Пример:

int nVar = 5;

int auto nVar = 5;

Две эти записи абсолютно идентичны.

#### **2. Регистровый (register)**

В языке программирования C++, регистровый класс памяти (register) используется для указания компилятору, что переменная должна быть сохранена в процессорном регистре, если это возможно. Это может повысить производительность программы, так как доступ к регистрам быстрее, чем доступ к оперативной памяти.

register int counter;

* Переменные register не могут быть разыменованы, т.е. нельзя получить их адрес с помощью оператора &. Это связано с тем, что регистры не имеют адресов в оперативной памяти.
* Применять register можно только к near указателям и целому типу. Использовать register можно и при указании формальных параметров функций. Примеры:

#include <iostream>

// Функция, принимающая параметры, объявленные с использованием ключевого слова 'register'

void processArray(register int\* array, register int size) {

register int sum = 0;

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

sum += array[i];

}

std::cout << "Sum of array elements: " << sum << std::endl;

}

int main() {

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

int size = sizeof(data) / sizeof(data[0]);

// Вызов функции с параметрами, объявленными как 'register'

processArray(data, size);

return 0;

}

#### **3. Статический (static)**

Статический класс памяти в C и C++ используется для переменных, которые должны сохранять свое значение между вызовами функции или быть доступными на уровне файла (файловая область видимости). Переменные, объявленные со спецификатором static, имеют статическую продолжительность хранения. Это означает, что память для них выделяется при запуске программы и освобождается только после завершения программы.

Вот несколько ключевых аспектов статических переменных:

1. Статические локальные переменные:
   1. Объявляются внутри функции с ключевым словом static.
   2. Их область видимости ограничена телом функции, но они сохраняют свое значение между вызовами этой функции.
2. Статические глобальные переменные:
   1. Объявляются вне функций с ключевым словом static.
   2. Их область видимости ограничена файлом, в котором они объявлены, что предотвращает их использование в других файлах.
3. Статические члены класса (в C++):
   1. Объявляются в классе с ключевым словом static.
   2. Они принадлежат самому классу, а не конкретному объекту, и потому разделяются всеми объектами этого класса.

// статическая локальная переменная

#include <iostream>

void counter() {

static int count = 0; // статическая локальная переменная

count++;

std::cout << "Count: " << count << std::endl;

}

int main() {

counter();

counter();

counter();

return 0;

}

// статическая глобальная переменная

#include <iostream>

static int globalVar = 0; // статическая глобальная переменная

void increment() {

globalVar++;

std::cout << "Global Variable: " << globalVar << std::endl;

}

int main() {

increment();

increment();

increment();

return 0;

}

// статический член класса

#include <iostream>

class Example {

public:

static int staticVar; // статический член класса

static void increment() {

staticVar++;

}

static void display() {

std::cout << "Static Variable: " << staticVar << std::endl;

}

};

int Example::staticVar = 0; // определение статического члена класса

int main() {

Example::increment();

Example::increment();

Example::display();

return 0;

}

#### **4. Внешний (external)**

Внешний класс памяти в C и C++ относится к переменным, которые имеют глобальную область видимости и могут использоваться в разных файлах программы. Эти переменные объявляются с использованием ключевого слова extern. Внешние переменные обычно определяются в одном файле и объявляются в других файлах, чтобы предоставить доступ к ним.

Ключевые особенности внешнего класса памяти:

1. *Объявление и определение:*
   1. Объявление: используется для указания, что переменная существует и может быть использована в данном файле. Синтаксис: extern type variableName;.
   2. Определение: выделяет память для переменной. Синтаксис: type variableName;.
2. Область видимости:
   1. Внешние переменные имеют глобальную область видимости, т.е. они доступны во всех файлах, где они объявлены.
3. Использование:

Внешние переменные позволяют разделять данные между разными файлами программы, что упрощает модульное программирование.

Пример:

// Файл main.c

#include <stdio.h>

// Объявление внешней переменной

extern int globalVar;

int main() {

// Использование внешней переменной

printf("Global Variable: %d\n", globalVar);

return 0;

}

// Файл globals.c

// Определение внешней переменной

int globalVar = 10;

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

#### **Объявление переменных на внутреннем уровне**

Объявление переменных на внутреннем и внешнем уровне относится к области видимости переменных в программе и определяет их доступность из разных частей программы.

Объявление переменных на внутреннем уровне:

* Переменные, объявленные на внутреннем уровне, являются локальными для блока кода, в котором они определены.
* Они доступны только в пределах этого блока или функции, в которой они были объявлены.
* При выходе из блока переменные этого уровня уничтожаются, и их значения больше не доступны.
* Пример:

void exampleFunction() {

int localVar = 10; // localVar объявлена на внутреннем уровне

// localVar доступна только внутри этой функции

}

**2. Объявление переменных на внешнем уровне**

Объявление переменных на внешнем уровне:

* Переменные, объявленные на внешнем уровне, имеют глобальную область видимости и доступны из любой части программы.
* Они могут быть объявлены в начале файла до основной функции программы или в другом файле и использоваться в других файлах, если они объявлены с помощью ключевого слова extern.
* Значение этих переменных сохраняется в памяти на протяжении всего времени работы программы.

// Внешнее объявление переменной

int globalVar = 20; // globalVar объявлена на внешнем уровне

int main() {

// globalVar доступна внутри этой функции и в любой другой части программы

}

# Внешнее объявление переменной через ключевое слово extern позволяет указать компилятору, что переменная была объявлена в другом файле, и сообщить о ее существовании без явного определения.

# Предположим, у вас есть два файла: main.c и external.c, и вы хотите использовать переменную globalVar, объявленную в файле external.c, в вашей основной программе в файле main.c. В таком случае вы можете сделать следующее:

// external.c

int globalVar = 20; // Определение переменной globalVa

// main.c

#include <stdio.h>

// Внешнее объявление переменной globalVar

extern int globalVar;

int main() {

printf("Value of globalVar: %d\n", globalVar); // Использование переменной globalVar

return 0;

}

В этом примере:

* В файле external.c переменная globalVar определена и инициализирована.
* В файле main.c переменная globalVar объявлена с помощью ключевого слова extern, что позволяет компилятору знать о существовании этой переменной без необходимости повторного определения.
* При компиляции и сборке программы оба файла будут связаны, и переменная globalVar из файла external.c будет доступна для использования в файле main.c.

Таким образом, внешнее объявление переменной через extern позволяет создавать программы, состоящие из нескольких файлов, где переменные могут быть объявлены в одном файле и использоваться в другом.

# **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**

Ключевое слово volatile указывает на то, что данная переменная в любой момент может быть изменена в результате выполнения внешних действий, не контролируемых программой. Например, следующая строка описывает переменную event\_time, значение которой может измениться без ведома программы:

volatile int event\_time;

Подобное описание переменной может понадобиться в том случае, когда переменная event\_time обновляется системными устройствами, например, таймером. При получении сигнала от таймера выполнение программы прерывается и значение переменной изменяется.

Пример:

bool exit = true;

while( exit )

{

};

В данном случае значение переменной **exit** будет прочитано только один раз. На первый взгляд в этом не ничего страшного. Однако, если содержание переменно **exit** может изменять не только код текущего потока, но допустим другой поток (опустим сейчас вопросы атомарности операции) или вообще некое внешние устройство, то мы можем получить бесконечный цикл коим он по нашей задумке быть не должен.

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

Пример:

volatile bool exit = true;

while( exit )

{

};

Чисто теоретически если быть стопроцентно уверенным в атомарности операции, то можно использовать квалификатор **volatile** для взаимодействие переменной между потоками без использования дополнительных объектов синхронизации таких как мьютексы, но поскольку такую уверенность может дать разве что **char**, то применять **volatile** в этих целях будет не корректным.

**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.

class Exm

{

int a;

int b;

public:

int getA() const

{

return a; // все правильно

}

int setA(int i) const

{

a = i;// ошибка доступа

}

}

Помочь в данном случае может определение переменной, а с ключевым словом **mutable**.

Ключевое слово mutable используется в контексте языка программирования C++ для указания на то, что член класса может быть изменен внутри const-контекста, то есть внутри метода класса, который объявлен как const.

Внесём исправление в приведённый чуть выше пример:

class Exm

{

mutable int a; // добавили в объявление ключевое слово mutable

// позволяющие игнорировать модификатор const

// по отношению к данной переменной

int b;

public:

int getA() const //

{

return a; // все правильно

}

int setA(int i) const

{

a = i;// теперь всё правильно. Мы можем изменять переменную а

b = i; // Ошибка! Переменная b по прежнему не доступна для изменения.

}

**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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | // 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:

|  |  |
| --- | --- |
| 1  2 | printValue(7); // 7 – аргумент функции printValue()  add(4, 5); // 4 и 5 – аргументы функции add() |

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

# **Формальные и фактические параметры функции.**

**Аргумент типа void**

using namespace std;

void voutput(void);

main()

{

/\* Программа определяет квадратный корень \*/

printf("This programm will find the square root\n\n\n");

voutput();

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

\_getch();

return(0);

}

void voutput(void)

{

double dt=12345.0;

double du;

du=sqrt(dt);

printf("The squre root of %lf is %lf\n",dt,du);

}

**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'. – 8 бит (1 байт)
* 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'.

void voutput(char c)

**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).

Чтобы указать десятичный целочисленный литерал, начинайте спецификацию с любой цифры, кроме нуля.

void vside(int is);

**Три типа целых чисел В C/C++** поддерживаются три типа целых чисел. Наравне со стандартным типом int существуют типы shortint (короткое целое) и longint (длинное целое). Допускается сокращенная запись short и long. Хотя синтаксис самого языка не зависит от используемой платформы, размерность типов данных short, intи long может варьироваться. Гарантируется лишь, что соотношение размерностей таково: short <= int <= long. В Microsoft Visual C/C++ для переменных типа short резервируется 2 байта, для типов int и long — 4 байта

**Описание данных целого типа.**

При описании данных необходимо ввести только тип, за которым должен следовать список имен переменных. Ниже приведены некоторые возможные примеры описаний:

int erns;

short stops;

long Johns;

int hogs, cows, goats;

**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.

Однако в программах вычислительного характера часто применяются числа с плавающей точкой. В языке Си такие данные описываются типом **float** - с плавающей точкой одинарной точности (32 бита)

**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 (64 бита)

**Массивы в качестве параметров**

Если функция принимает в качестве параметра массив, то фактически в эту функцию передается указатель на первый элемент массива. То есть как и в случае с указателями нам доступен адрес, по которому мы можем менять значения. Поэтому следующие объявления функции будут по сути равноценны

void print(int numbers[]);

void print(int \*numbers);

void print(int\*);

int main()

{

    int nums[] {1, 2, 3, 4, 5};

    print(nums);

}

void print(int \*numbers)

{

    std::cout << "First number: " << \*numbers << std::endl;

}

**Аргументы по умолчанию.**

В программировании, аргумент по умолчанию является аргументом функции, который при вызове необязательно указывать.

**void f(int a = 1, int b = 2, int c = 3);**

**f(10); // будет вызвано f(10, 2, 3)**

**f(10, 20); // будет вызвано f(10, 20, 3)**

**f(10, 20, 30); // значения по умолчанию совсем не использованы**

Её можно вызвать вообще без параметров:

**f(); //для всех параметров использованы значения по умолчанию - f(1, 2, 3)**

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

**1. Пространства имен**

Пространство имен (namespace) в C++ представляет собой механизм для организации кода в логические группы и избегания конфликтов имен между различными частями программы. Пространства имен позволяют создавать отдельные области видимости для идентификаторов, таких как типы, функции, переменные и другие сущности.

Основная цель пространств имен - предоставить механизм для структурирования кода и предотвращения возможных конфликтов имен между различными частями программы, особенно при интеграции различных библиотек или модулей.

Пространства имен позволяют создавать изолированные области, внутри которых имена могут использоваться безопасно, так как они не конфликтуют с именами в других пространствах имен или в глобальной области.

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

namespace ContosoData // создание пространства

{

class ObjectManager

{

public:

void DoSomething() {}

};

void Func(ObjectManager) {}

}

Импорт полного имени:

ContosoData::ObjectManager mgr;

// мы поясняем что берём структуру из пространства ContosoData

mgr.DoSomething();

ContosoData::Func(mgr);

Чтобы добавить в область видимости один идентификатор(имя из пространства), 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);

**2. Функции и классы памяти. Объявление функции на внешнем уровне**

Функции в языке C++ могут иметь спецификаторы класса памяти, такие как static или extern. Эти спецификаторы определяют область видимости и время жизни функций в программе.

1. Функции с классом памяти static: Функция, объявленная с ключевым словом static, имеет локальную область видимости, что означает, что она доступна только в пределах того же исходного файла, в котором она определена. Такие функции нельзя вызвать из других исходных файлов. Функции с классом памяти static обычно используются для создания локальных помощников или вспомогательных функций, которые не должны быть доступны извне данного файла.

// Файл1.cpp

static void helperFunction() {

// ...

}

// Файл2.cpp

void anotherFunction() {

// Нельзя вызвать helperFunction из этого файла

}

1. Функции с классом памяти extern: Функция, объявленная с ключевым словом extern, доступна во всех исходных файлах программы. Такие функции могут быть вызваны из любого места в программе. Обычно функции, которые должны быть доступны из разных исходных файлов, объявляются как extern.

// Файл1.cpp

extern void sharedFunction(); // Объявление

// Файл2.cpp

void useSharedFunction() {

sharedFunction(); // Можно вызвать функцию из другого файла

}

1. Объявление функций без спецификатора класса памяти (по умолчанию extern): Если при объявлении функции не указан спецификатор класса памяти, она по умолчанию считается extern. Такие функции доступны во всех исходных файлах программы и могут быть вызваны из любого места программы.

// Файл1.cpp

void globalFunction(); // По умолчанию extern

// Файл2.cpp

void useGlobalFunction() {

globalFunction(); // Можно вызвать функцию из другого файла

}

Объявление функции обычно помещается в заголовочные файлы, а определение функции - в соответствующие файлы реализации. Это позволяет разделить интерфейс и реализацию функций, упрощая структуру программы.

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

Приведение типов в C++ представляет собой механизм, который позволяет изменять типы данных одного объекта на другие типы данных. Однако использование приведения типов следует минимизировать, так как это может быть признаком неправильного проектирования программы.

Вот краткое описание основных операторов приведения типов в C++:

1. const\_cast: Используется для удаления const или volatile квалификаторов из переменной. Например, если переменная была объявлена как const int, то с помощью const\_cast можно преобразовать её в int и изменить.
2. static\_cast: Предназначен для выполнения безопасных преобразований между совместимыми типами данных. Может применяться для приведения между встроенными типами данных, указателями и ссылками, а также между объектами классов в некоторых случаях, например, вверх или вниз по иерархии наследования. Ошибка приведения типов обычно обнаруживается на этапе компиляции.
3. dynamic\_cast: Используется для выполнения безопасных преобразований во время выполнения программы. Применяется в основном для приведения указателей и ссылок между объектами классов, особенно при использовании наследования. Если приведение невозможно, dynamic\_cast возвращает нулевой указатель или выбрасывает исключение bad\_cast.
4. reinterpret\_cast: Предназначен для выполнения опасных преобразований между несовместимыми типами данных. Не проводит никаких проверок и предполагает, что программист знает, что делает. Обычно используется для преобразования указателей или целых чисел друг в друга.

Приведение типов следует использовать очень осторожно, и в большинстве случаев можно обойтись без него. В некоторых ситуациях приведение типов может быть необходимым, но перед его использованием необходимо тщательно оценить возможные последствия и альтернативные подходы.

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

**Идентичные типы**

Идентичный тип в программировании обозначает типы данных, которые абсолютно одинаковы и могут использоваться взаимозаменяемо без каких-либо изменений. В C++ идентичные типы имеют одинаковую структуру, размер и представление в памяти.

Примеры идентичных типов в C++:

* int и signed int (синонимические типы)
* float и double (разные типы с плавающей запятой, но с одинаковым базовым типом)
* int\* и signed int\* (указатели на один и тот же базовый тип)

**Перечисляемые типы (enum)**

При объявлении переменной перечисляемого типа с ней связывается набор именованных целочисленных констант, называемых перечисляемыми константами. В каждый момент времени переменная может иметь значение одной из констант, и к ней можно обращаться по имени. К примеру, следующее описание создает перечисляемый тип air\_supply, перечисляемые константы EMPTY, USEABLE и FULL, и перечисляемую переменную instructor\_tank:

enum air\_supply {EMPTY,

USEABLE,

FULL=5 } instructor\_tank;

Все константы и переменные имеют тип int, каждая константа автоматически получает по умолчанию некоторое значение, если не указывается какое-либо другое. В предыдущем примере константа EMPTY имеет по умолчанию целое значение ноль, так как она — первая в списке и не переопределена особо. Значение константы USEABLE равно 1, поскольку она следует непосредственно за константой с нулевым значением. Константа FULL инициализирована отдельно и имеет значение 5; если бы вслед за ней в списке была еще одна константа, эта новая константа была бы равна 6.

**Хороший стиль программирования**

Использование перечислений вместо целых констант облегчает чтение программы.

Перечислимые константы могут определяться и инициализироваться произвольными целочисленными константами, а также константными выражениями:

**enum** ages (milton = 47, ira, harold = 56, philip = harold + 7};

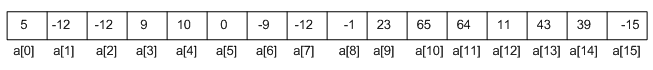
Обратите внимание на то, что когда нет явного инициализатора, применяется правило по умолчанию, таким образом - ira = 48. Значения перечислимых констант могут быть не уникальными.

**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](http://cppstudio.com/obuchenie_cpp/ukazateli).

Отдельная ячейка данных массива называется *элементом массива*.  Элементами массива могут быть  данные любого типа. Массивы могут иметь как одно, так и более одного измерений. В зависимости от количества измерений массивы делятся на одномерные массивы, двумерные массивы, трёхмерные массивы и так далее до n-мерного массива. Чаще всего в программировании используются одномерные и двумерные массивы

**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. Столбцы в одномерном массиве — это элементы массива. На рисунке 1 показана структура целочисленного одномерного массива a. Размер этого массива — 16 ячеек.



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

int a[16] = { 5, -12, -12, 9, 10, 0, -9, -12, -1, 23, 65, 64, 11, 43, 39, -15 }; // инициализация одномерного массива

int a[]={5,-12,-12,9,10,0,-9,-12,-1,23,65,64,11,43,39,-15}; // инициализации массива без определения его размера.

**Двумерные массивы**

Простейшим видом многомерного массива является двумерный массив. Двумерный массив - это массив одномерных массивов. Двумерный массив объявляется следующим образом:  
  
*тип имя\_массива[размер второго измерения][размер первого измерения];*  
  
Следовательно, для объявления двумерного массива целых с размером 10 на 20 следует написать:  
  
int d[10] [20] ; // объявление (m,n)

int a[4][3] = { {4,8,9}, {5,-2,7}, {2,1,1}, {0,13,-1} };

//объявление и инициализация

где, m — количество строк двумерного массива;  
n — количество столбцов двумерного массива;  
m \* n— количество элементов массива.

**О типах функции:**

 В языке программирования С++ есть два типа функций:

1. Функции, возвращающие значение
2. Функции, которые ничего не возвращают (начинаются с зарезервированного слова void — это тип данных, который не может хранить какие-либо данные)

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. Объявление структуры приводит к образованию шаблона, используемого для создания объектов структуры

Пример кода:

*Struct name {*

*int hour;*

*int min;*

*int sec;*

*};*

Он похож на class, но есть два мелких различия:

- по умолчанию в struct действует видимость public, а в class — private

* по умолчанию struct наследует члены базовых структур/классов как публичные члены, а class — как приватные члены.

Объединение (union) в C++ - это специальная структура данных, которая позволяет использовать одно и то же область памяти для разных типов данных. В отличие от структуры, где каждый член имеет свою собственную область памяти, в объединении все члены совместно используют одну и ту же область памяти. Это означает, что объединение хранит только одно значение в любой момент времени.

union Data {

int intValue;

float floatValue;

char charValue;

};

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

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

*#include <iostream>*

*union Data {*

*int intValue;*

*float floatValue;*

*char charValue;*

*};*

*int main() {*

*Data data;*

*data.intValue = 10;*

*std::cout << "Int value: " << data.intValue << std::endl;*

*data.floatValue = 3.14f;*

*std::cout << "Float value: " << data.floatValue << std::endl;*

*data.charValue = 'A';*

*std::cout << "Char value: " << data.charValue << std::endl;*

*return 0;*

*}*

**Указатели**

Указатели — это переменные, которые содержат адреса памяти других переменных или объектов в компьютере

int x = 10;

int\* ptr = &x; // Указатель ptr теперь указывает на переменную x

int& ptr = x;

этой записи переменная ptr объявляется как ссылка на целое число (int&), и ей присваивается значение переменной x. Ссылка в C++ - это альтернативное имя переменной, которое может использоваться вместо оригинального имени переменной. Таким образом, ptr становится альтернативным именем для переменной x, и обе эти переменные будут указывать на одно и то же место в памяти. Изменение значения ptr приведет к изменению значения x и наоборот.

Используя унарную операцию взятия адреса **&**, мы извлекаем адрес переменной *i\_val* и присваиваем ее указателю. Здесь стоит обратить внимание на следующие вещи:

1. Тип, используемый при объявлении указателя в точности, должен соответствовать типу переменной, адрес которой мы присваиваем указателю.
2. Не следует путать оператор взятия адреса со ссылкой на некоторое значение, которое так же визуально отображается символом **&**.

# **Указатели на массивы**

Указатели на массивы в C++ представляют собой особый тип указателей, который указывает на первый элемент массива.  
  
int arr[5] = {1, 2, 3, 4, 5};

int\* ptr = arr; // Указатель ptr указывает на начало массива arr

ptr++; // Указывает на следующий элемент массива

А теперь будем обращаться к элементам массива, используя указатели:

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

int\* ptr = arr;

std::cout << \*ptr; // Выведет: 1

std::cout << \*(ptr + 1); // Выведет: 2

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

int\* ptr = arr;

for (int i = 0; i < 5; ++i) {

std::cout << \*(ptr + i) << " "; // Выведет: 1 2 3 4 5

}

# **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.**

**Макроопределения**

Макроопределения в языке программирования представляют собой инструкции препроцессора, которые позволяют создавать символьные замены для текстового кода до его компиляции. Это значит, что препроцессор заменяет каждое вхождение макроопределения в коде на соответствующий текст до передачи кода компилятору.

**Определение макросов**

Определение макросов (макроопределения) в языке программирования представляет собой создание символьных замен для определенных текстовых конструкций. Эти замены осуществляются препроцессором до того, как код будет передан компилятору.

**Макросы и параметры**

Макросы в языке программирования могут принимать параметры, что позволяет создавать более гибкие и мощные замены текста. Параметры макроса указываются в определении макроса в круглых скобках и могут использоваться внутри замещающего текста.

**Сложности при раскрытии макросов**

При раскрытии макросов могут возникать несколько сложностей:

1. *Приоритет операций*: Если замещающий текст макроса содержит выражения, важно учитывать приоритет операторов, чтобы избежать непредвиденных результатов. Например, в макросе #define SQUARE(x) x \* x необходимо использовать скобки вокруг x, чтобы избежать ошибок при раскрытии.
2. *Побочные эффекты*: Макросы могут вызывать побочные эффекты, особенно если они используются для изменения состояния или вызова функций. При раскрытии макросов с побочными эффектами может быть сложно предсказать результаты выполнения программы.
3. *Повторное вычисление аргументов*: Если аргументы макроса содержат выражения или функции, они могут быть вычислены несколько раз при раскрытии макроса. Это может привести к неожиданным результатам или даже к ошибкам в программе.
4. *Читаемость кода*: Использование макросов может затруднить чтение и понимание кода, особенно если они используются для создания сложных конструкций или вложенных выражений. Чрезмерное использование макросов может сделать код менее читаемым и поддерживаемым.
5. *Отладка*: При отладке программы может быть сложно отследить ошибки, связанные с макросами, особенно если они используются в большом количестве и в различных частях программы. Ошибки в макросах могут привести к непредсказуемому поведению программы и требовать дополнительного времени для их обнаружения и исправления.

Для минимизации этих сложностей рекомендуется использовать макросы с осторожностью, избегать сложных выражений в замещающем тексте и документировать их использование для облегчения понимания кода другими разработчиками.

**Создание и использование собственных макросов.**

Создание и использование собственных макросов позволяет программистам упрощать код, делать его более гибким и удобным для использования. Вот несколько шагов по созданию и использованию собственных макросов:

1. Определение макроса: Используйте директиву препроцессора #define для создания макроса. Например:

#define MAX(a, b) ((a) > (b) ? (a) : (b))

Этот макрос определяет макрос MAX, который возвращает максимальное значение между двумя аргументами a и b.

1. Использование макроса: После определения макроса вы можете использовать его в коде. Просто напишите имя макроса и передайте ему необходимые аргументы. Например:

int result = MAX(x, y);

Здесь MAX(x, y) заменится на ((x) > (y) ? (x) : (y)), и result получит большее из значений x и y.

1. Указание аргументов в скобках: Важно всегда заключать аргументы макроса в скобки, чтобы избежать непредвиденного поведения при использовании выражений с побочными эффектами. Например:

#define SQUARE(x) ((x) \* (x))

Здесь каждое вхождение x обрамлено скобками, чтобы гарантировать правильность раскрытия макроса.

1. Обработка препроцессором: Обратите внимание, что макросы обрабатываются препроцессором до компиляции кода. Это означает, что макросы заменяются исходным кодом перед тем, как компилятор начнет работу. Важно понимать, что макросы работают на уровне текста и не имеют понятия о типах данных или контексте выполнения.
2. Документирование и комментирование: При создании макросов важно документировать их использование, указывая ожидаемые аргументы и результаты. Это поможет другим разработчикам лучше понимать ваш код и использовать макросы правильно.

Создание и использование собственных макросов может значительно упростить и ускорить разработку программного обеспечения, но требует осторожности и внимания к деталям, чтобы избежать ошибок и нежелательного поведения.

**Макросы, поставляемые вместе с компилятором**

Макросы, поставляемые вместе с компилятором, часто предназначены для обеспечения портируемости кода между различными компиляторами, для определения характеристик целевой платформы или для управления различными аспектами компиляции. Вот несколько примеров таких макросов:

* \_\_cplusplus: Этот макрос используется для проверки версии стандарта C++, который поддерживается компилятором.

#ifdef \_\_cplusplus

// Код, зависящий от стандарта C++

#endif

* \_\_GNUC\_\_: Этот макрос определен компилятором GCC и его производными, такими как Clang, и может использоваться для проверки, используется ли кодовая база компилятора GCC.

#ifdef \_\_GNUC\_\_

// Код, зависящий от компилятора GCC

#endif

* \_\_WIN32: Этот макрос определен в среде Windows и может использоваться для написания кросс-платформенного кода.

#ifdef \_\_WIN32

// Код, выполняемый только в среде Windows

#endif

* \_\_unix\_\_: Этот макрос определен в UNIX-подобных операционных системах и может использоваться для написания кросс-платформенного кода.

#ifdef \_\_unix\_\_

// Код, выполняемый только в UNIX-подобных системах

#endif

Эти макросы помогают программистам создавать код, который может быть скомпилирован и выполнен на разных платформах и с разными компиляторами, делая код более переносимым и гибким.

**Выбор макроопределения или функции.**

Выбор между макроопределениями и функциями зависит от конкретной ситуации и требований вашего проекта. Вот некоторые соображения, которые могут помочь в принятии решения:

1. Производительность: Макросы обрабатываются препроцессором и просто заменяются на соответствующий код во время компиляции. В результате нет накладных расходов на вызов функций. Это может быть полезно для небольших и быстрых операций, особенно внутри циклов или других участков кода, где вызов функции может сказаться на производительности.
2. Повторное использование кода: Функции предоставляют более модульный и легко читаемый код. Они могут быть вызваны из разных частей программы и переиспользованы многократно. Это особенно важно, если вы планируете использовать определенные операции несколько раз в разных частях вашего проекта.
3. Побочные эффекты: Функции обеспечивают более безопасное поведение при работе с побочными эффектами и изменяемыми переменными. При использовании макросов необходимо быть внимательным к скобкам и порядку операций, чтобы избежать непредвиденного поведения.
4. Типы данных и проверка типов: Функции могут обрабатывать типы данных и выполнять проверку типов во время компиляции, что обеспечивает более безопасное программирование и предотвращает ошибки во время выполнения. Макросы работают на уровне текста и не имеют понятия о типах данных, поэтому они менее безопасны с этой точки зрения.
5. Отладка и профилирование: Использование функций облегчает отладку и профилирование кода, так как у вас есть возможность точно определить место вызова функции и анализировать ее работу. Макросы могут усложнить процесс отладки, так как они просто заменяются на соответствующий код.

В конечном счете выбор между макросами и функциями зависит от ваших конкретных потребностей и предпочтений, а также от стандартов и практик вашей команды разработки.

# **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 #. По окончании строк, обозначающих директивы в языке Си, точку с запятой можно не ставить.

Директивы препроцессора в языках программирования C и C++ используются для обработки исходного кода до его компиляции. Они позволяют включать, исключать или модифицировать части кода, а также предоставляют способы определения и использования макросов.

Директива #define:

Описание: Используется для создания макросов или символьных констант.

Пример:

#define PI 3.14159

Директива #include:

Описание: Используется для включения содержимого других файлов в текущий файл.

Пример:

#include <iostream>

Заголовочные файлы:

Описание: Файлы с расширением .h (или .hpp в C++), содержащие объявления функций, классов, макросов и других элементов, которые могут быть использованы в других файлах.

Пример:

iostream, vector, string.

Правильное использование заголовочных файлов:

* Использование директивы #include для включения нужных заголовочных файлов в исходный код.
* Использование инструкции #ifndef, #define, #endif для защиты заголовочных файлов от множественного включения.

Более эффективное использование заголовочных файлов:

* Минимизация количества включаемых заголовочных файлов.
* Использование предварительной компиляции заголовочных файлов.

Новый стиль заголовков:

* Использование расширений .hpp для заголовочных файлов в C++.
* Размещение объявлений в пространствах имен.

Предварительная компиляция заголовочных файлов:

* Применение предварительной компиляции для сокращения времени компиляции проекта.
* Использование опций компилятора, таких как -H в GCC, для анализа зависимостей между файлами.

Файлы limits.h и float.h:

* limits.h: Содержит макросы с граничными значениями для различных числовых типов данных, таких как INT\_MAX, INT\_MIN.
* float.h: Содержит макросы с информацией о представлении чисел с плавающей точкой, таких как FLT\_MAX, DBL\_EPSILON.

Эффективное использование директив препроцессора и заголовочных файлов позволяет улучшить структуру и производительность проекта, а также сделать код более поддерживаемым и масштабируемым.

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

Директивы препроцессора #ifdef, #endif, #undef, #ifndef, #if, #else, #elif предоставляют возможность условной компиляции кода в зависимости от наличия определенных макросов или условий.

Директива #ifdef:

Описание: Проверяет, определен ли указанный макрос.

Использование:

#ifdef MACRO\_NAME

// код, который будет выполнен, если макрос MACRO\_NAME определен

#endif

Директива #endif:

Описание: Окончание блока условной компиляции, начатого директивой #ifdef.

Использование: В паре с #ifdef, #ifndef, #if, #elif.

Директива #undef:

Описание: Снимает определение макроса, ранее созданного с помощью #define.

Использование:

#undef MACRO\_NAME

Директива #ifndef:

Описание: Проверяет, не определен ли указанный макрос.

Использование:

#ifndef MACRO\_NAME

// код, который будет выполнен, если макрос MACRO\_NAME не определен

#endif

Директива #if:

Описание: Позволяет выполнить условную компиляцию на основе выражения.

Использование:

#if выражение

// код, который будет выполнен, если выражение истинно

#endif

Директива #else:

Описание: Определяет альтернативный блок кода, который будет выполнен, если условие в #if или #ifdef ложно.

Использование: В паре с #if, #ifdef.

Директива #elif:

Описание: Сокращенная форма комбинации #else и #if. Позволяет добавить еще одно условие для проверки.

Использование:

#if выражение1

// код, который будет выполнен, если выражение1 истинно

#elif выражение2

// код, который будет выполнен, если выражение2 истинно

#else

// код, который будет выполнен, если ни одно из условий не истинно

#endif

Директивы условной компиляции позволяют создавать гибкий и настраиваемый код, который может быть скомпилирован с различными настройками в зависимости от потребностей проекта.

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

Директивы препроцессора #line, #error и #pragma предоставляют дополнительные возможности для управления процессом компиляции и выводом сообщений об ошибках.

Директива #line:

Описание: Устанавливает номер текущей строки и, при необходимости, имя исходного файла.

Использование:

#line номер\_строки "имя\_файла"

Директива #error:

Описание: Прерывает процесс компиляции и выводит заданное сообщение об ошибке.

Использование:

#error "Сообщение об ошибке"

Директива #pragma:

Описание: Позволяет задавать различные инструкции компилятору или препроцессору.

Использование:

#pragma инструкция

Примеры использования:

Директива #line:

#line 100 "example.cpp"

Директива #error:

#ifdef DEBUG

#error "Отладочная информация включена"

#endif

Директива #pragma:

#pragma once // Задает, что заголовочный файл будет включен только один раз

#pragma warning(disable: 4996) // Отключает предупреждение компилятора с определенным номером

Директивы #line, #error и #pragma предоставляют разработчикам дополнительные средства для управления компиляцией, отладкой и настройкой проекта.

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

Оператор defined и условная компиляция являются мощными инструментами препроцессора в языке С++ для контроля процесса компиляции в зависимости от определенных условий.

**Оператор defined:**

Описание: Оператор defined используется для проверки, определен ли макрос препроцессора.

Использование:

#if defined(МАКРОС)

// код, который выполнится, если МАКРОС определен

#endif

*Пример использования:*

#define DEBUG // Определение макроса DEBUG

#if defined(DEBUG)

// Код, который выполнится, если DEBUG определен

std::cout << "Отладочная информация" << std::endl;

#endif

**Условная компиляция:**

Описание: Условная компиляция позволяет компилировать или пропускать части кода в зависимости от условий, определенных препроцессором.

Использование:

#ifdef МАКРОС

// код, который выполнится, если МАКРОС определен

#else

// код, который выполнится, если МАКРОС не определен

#endif

*Пример использования:*

#define OS\_WINDOWS // Определение макроса OS\_WINDOWS

#ifdef OS\_WINDOWS

// Код, который выполнится, если OS\_WINDOWS определен

#include <Windows.h>

#else

// Код, который выполнится, если OS\_WINDOWS не определен

#include <unistd.h>

#endif

Оператор defined и условная компиляция позволяют программистам создавать гибкий и переносимый код, который может быть настроен в зависимости от определенных условий или параметров среды выполнения.

# **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:

Дополнительные операции препроцессора, такие как операция подстановки строки (#), операция конкатенации (##) и операция подстановки символа (#@), предоставляют дополнительные возможности для гибкой и мощной работы с макросами в языке C++.

**Операция подстановки строки (#):**

Описание: Эта операция преобразует аргумент макроса в строковый литерал, который представляет его текст.

Использование:

#define STRINGIFY(x) #x

std::cout << STRINGIFY(Hello) << std::endl; // Выводит "Hello"

**Операция конкатенации (##):**

Описание: Эта операция объединяет два токена в один новый токен.

Использование:

#define CONCAT(x, y) x##y

int xy = CONCAT(10, 20); // Эквивалентно int xy = 1020;

**Операция подстановки символа (#@):**

Описание: Эта операция позволяет макросам получать доступ к своим аргументам посредством символов, начинающихся с '@'.

Использование:

#define SHOW\_INT(x) std::cout << "Value: " << @x << std::endl;

int value = 42;

SHOW\_INT(value); // Выводит "Value: 42"

Эти операции расширяют возможности макросов, позволяя создавать более сложные и гибкие макросы для обработки различных сценариев в программировании на C++. Однако следует использовать их с осторожностью, поскольку их неправильное использование может привести к трудноотлавливаемым ошибкам в коде.

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

**Обработка ошибок: perror():**

Описание: Функция perror() используется для вывода сообщения об ошибке в стандартный поток ошибок (stderr) на основе глобальной переменной errno, которая содержит код последней произошедшей ошибки.

Использование:

#include <stdio.h>

#include <errno.h>

int main() {

FILE \*fp;

fp = fopen("file.txt", "r");

if (fp == NULL) {

perror("Error opening file");

return -1;

}

// Остальная часть программы

fclose(fp);

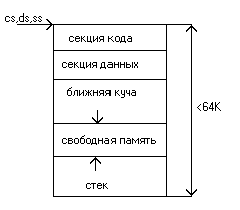
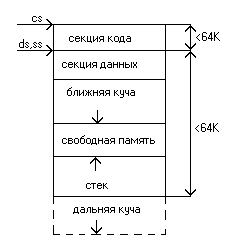
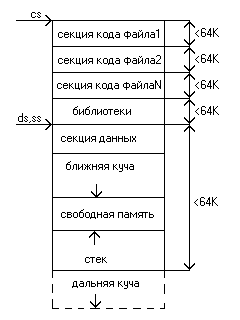
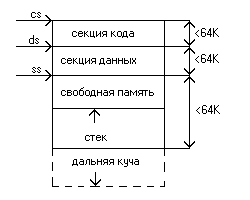
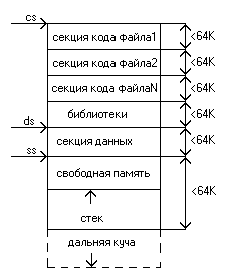
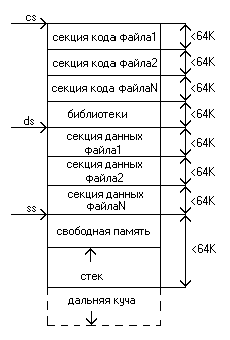
return 0;

}

Этот код выведет сообщение об ошибке, связанной с открытием файла, в стандартный поток ошибок с дополнительной информацией о причине ошибки, полученной из errno.

**Модели памяти:**

Описание: Модели памяти определяют, как операционная система управляет памятью приложения. В DOS было шесть основных моделей памяти: Tiny, Small, Medium, Compact, Large и Huge.

* Модель Tiny: Все программы и данные находятся в одном сегменте размером 64 КБ. 
* Модель Small: Имеет больше сегментов, но каждый ограничен размером 64 КБ. Можно использовать больше памяти, чем в Tiny, но все равно ограничено. 
* Модель Medium: Расширяет Small модель до 128 КБ для сегмента данных и 64 КБ для сегмента кода. 
* Модель Compact: Использует большие сегменты размером до 64 КБ для кода и данных, но меньше чем в Large. 
* Модель Large: Использует один сегмент размером более 64 КБ для кода и один для данных. Может использовать всю доступную память до 1 МБ. 
* Модель Huge: Поддерживает программы, которые используют больше 1 МБ памяти. 

Каждая модель имеет свои ограничения по размеру сегментов памяти и обеспечивает различные уровни доступа к памяти в зависимости от размера и типа приложения.

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

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

Первые два модификатора, pascal и cdecl, чаще всего используется в сложных приложениях. Microsoft Visual C/C++ позволяет создавать программы, которые с легкостью могут вызывать процедуры, написанные на разных языках. Также возможна связь и в обратном направлении. Например, можно написать программу на Паскале, вызывающую процедуру C++. При подобном смешении языков необходимо учитывать два момента: имена идентификаторов и способ передачи параметров.

**Pascal**

Применение модификатора **pascal** к идентификатору приводит к тому, что идентификатор преобразуется к верхнему регистру и к нему не добавляется символ подчеркивания. Этот идентификатор может использоваться для именования в программе на языке Си глобального объекта, который используется также в программе на языке Паскаль. В объектном коде, сгенерированном компилятором языка Си, и в объектном коде, сгенерированном компилятором языка Паскаль, идентификатор будет представлен идентично

**Cdecl**

Существует опция компиляции, которая присваивает всем функциям и указателям на функции тип **pascal**. Это значит, что они будут использовать вызывающую последовательность, принятую в языке Паскаль, а их идентификаторы будут приемлемы для вызова из программы на Паскале. При этом 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**.

**Модификатор pascal**

Применение модификатора **pascal** к идентификатору приводит к тому, что идентификатор преобразуется к верхнему регистру и к нему не добавляется символ подчеркивания. Этот идентификатор может использоваться для именования в программе на языке Си глобального объекта, который используется также в программе на языке Паскаль. В объектном коде, сгенерированном компилятором языка Си, и в объектном

*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 (или другом языке программирования с подобными правилами написания имен внешних переменных). 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**

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

*Примечание.* Все функции в стандартных включаемых файлах (например, stdio.h) 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

**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** (структурированная переменная, содержащая информацию о файле) 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.

абота с файлами включает в себя открытие, чтение, запись и закрытие файлов. Для этих целей в C++ используется библиотека <fstream>, которая предоставляет классы для работы с файловыми потоками.

Основные классы для работы с файлами

std::ifstream (input file stream) — для чтения из файлов.

std::ofstream (output file stream) — для записи в файлы.

std::fstream (file stream) — для чтения и записи в файлы.

Открытие файлов

Чтобы открыть файл, нужно создать объект соответствующего класса и связать его с файлом с помощью метода open() или конструктора. Пример:

#include <iostream>

#include <fstream>

int main() {

// Открытие файла для чтения

std::ifstream inputFile("example.txt");

// Открытие файла для записи

std::ofstream outputFile("output.txt");

// Закрытие потоков

inputFile.close();

outputFile.close();

file.close();

return 0;

}

После завершения работы с файлом, необходимо закрыть поток, чтобы освободить ресурсы, связанные с файлом. Это делается с помощью метода close(). В большинстве случаев потоки автоматически закрываются при завершении программы или выходе из области видимости, но явное закрытие потока является хорошей практикой.

Чтение из файла может быть выполнено с использованием операторов >> или методов getline() и read(). Запись в файл осуществляется с использованием операторов << или методов write().

Пример чтения и записи:

#include <iostream>

#include <fstream>

#include <string>

int main() {

std::ifstream inputFile("example.txt");

std::ofstream outputFile("output.txt");

std::string line;

while (std::getline(inputFile, line)) {

outputFile << line << '\n'; // Запись строки в выходной файл

}

inputFile.close();

outputFile.close();

return 0;

}

**Проверка состояния потоков**

Для проверки успешного открытия файла и состояния потоков используются методы и флаги:

is\_open() — проверяет, открыт ли файл.

Флаги состояния: eof() (конец файла), fail() (ошибка при операции), bad() (неустранимая ошибка), good() (все хорошо).

Пример проверки состояния:

if (inputFile.is\_open()) {

// Файл открыт успешно

}

if (inputFile.eof()) {

// Достигнут конец файла

}

if (inputFile.fail()) {

// Ошибка при выполнении операции

}

**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.

И если при попытке открытия файла по указанному пути его не окажется, то консоль выведет следующую ошибку:

Error occured while opening data28.txt: No such file or directory

**Переменная errno и коды ошибок**

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

**errno** – переменная, хранящая целочисленный код последней ошибки. В каждом потоке существует своя локальная версия **errno**, чем и обусловливается её безопасность в многопоточной среде. 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.

Прочие коды ошибок (несколько десятков) и их описания определены в стандарте POSIX. Кроме того, в спецификациях стандартных функций обычно указываются используемые ими коды ошибок и их описания.

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:

«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. 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. Запись символа в файл:

fputc(символ,поток);

Аргументами функции являются символ и указатель на поток типа FILE. Функция возвращает код считанного символа. Функции fscanf()и fprintf()аналогичны функциям scanf() и printf(), но 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, они являются аналогами функций gets() и puts() для работы с файлами. 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);

Копирует строку в поток с текущей позиции. Завершающий нуль- символ не копируется.

Пример Ввести число и сохранить его в файле s1.txt. Считать число из файла s1.txt, увеличить его на 3 и сохранить в файле s2.txt.

#include <stdio.h>

#include <stdlib.h>

int main()

{

FILE \*S1, \*S2;

int x, y;

system(«chcp 1251»);

system(«cls»);

printf(«Введите число: «);

scanf(«%d», &x);

S1 = fopen(«S1.txt», «w»);

fprintf(S1, «%d», x);

fclose(S1);

S1 = fopen(«S1.txt», «r»);

S2 = fopen(«S2.txt», «w»);

fscanf(S1, «%d», &y); y += 3;

fclose(S1);

fprintf(S2, «%d\n», y);

fclose(S2); return0;

}

**Результат выполнения — 2 файла**

ПЕРЕКЛЮЧЕНИЕ И РАБОТА С ФАЙЛАМИ

Понятие ввода вывода включает в себя функции данные и устройства Рассмотрим - , . , например нашу программу , ввод вывод - 4. В ней используется функция getchar( ), осуществляющая ввод причем устройство ввода клавиатура в соответствии с нашим , - ( предположением а входные данные отдельные символы Нам бы хотелось сохранить функции ), - . ввода и тип данных но изменить источник их поступления в программу Зададимся вопросом , . : откуда программа узнает где искать входные данные ?

По умолчанию Си программа рассматривает стандартный ввод как источник - " " поступления данных Стандартным вводом называется устройство принятое в качестве . " , обычного средства ввода данных в машину Это может быть устройство чтения данных с . магнитной ленты или перфокарт телетайп или как мы продолжаем считать терминал , ( ) . Современная машина это послушный инструмент и мы можем воздействовать на нее так - , , чтобы она вводила данные из любого источника В частности мы можем сообщить программе . , , что источник входных данных файл а не клавиатура .

Существуют два способа написания программ работающих с файлами Первый способ , . заключается в явном использовании специальных функций которые открывают и закрывают , файлы организуют чтение и запись данных и т п мы не хотим пока касаться этого вопроса , . .; . Второй способ состоит в том чтобы использовать программу спроектированную первоначально , , в предположении что данные в нее вводятся с клавиатуры и выводятся на экран переключить , , ввод и вывод на другие информационные каналы например из файла в файл Этот способ в : , . некоторых отношениях обладает меньшими возможностями чем первый но зато гораздо проще , , в использовании Мы изучим понятие переключения в данном разделе . .

Переключение ввода

Теперь предположим, , , , вам хочется чтобы слова вводимые с клавиатуры попадали в файл с именем mywords. Для этого вы должны 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 , , . ( ОС однако вам предоставляется возможность защитить существующие файлы На экране UNIX, , .) 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 (sic!). 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.

#include <conio.h>

#include <stdio.h>

#include <stdlib.h>

#define ERROR\_FILE\_OPEN -3

void main()

{

FILE \*output = NULL;

int number;

output = fopen("D:/c/output.bin", "wb");

if (output == NULL)

{

printf("Error opening file");

getch();

exit(ERROR\_FILE\_OPEN);

}

scanf("%d", &number);

fwrite(&number, sizeof(int), 1, output);

fclose(output);

\_getch();

}

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. Вместо массив, конечно, может быть передан любой объект. Запись в бинарный файл объекта похожа на его отображение: берутся данные из оперативной памяти и пишутся как есть. 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.

К сожалению, стандартом не определено, что такое конец файла, поэтому полагаться на эту функцию нельзя. В случае удачной работы функция возвращает 0.

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

Для программиста открытый файл представляется как последовательность считываемых или записываемых 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:

* **«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/записи (проверяет поток на признак конца файла). 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.

Для дополнения файла новыми данными надо установить текущую позицию на конец файла и выполнить операцию записи.

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. Если текущая позиция не определена, функция возвращает  -1L.

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 положение (начало файла).

Эквивалентный вызов функции fseek для возвращения указателя на начало:

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

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

Не маловажным отличием данных функций является то, что, в отличие от функции fseek, rewind обнуляет индикатор ошибок. Для потоков, открытых в режиме обновления (чтения + записи), вызов функции rewind позволяет переключаться между режимами чтения и записи.

**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.
* В случае успеха, функция возвращает нулевое значение.  
  В противном случае она возвращает ненулевое значение и инициализирует  глобальную переменную ERRNO положительным значением, которое может быть интерпретировано [функцией perror](http://cppstudio.com/spravochnik/standartnye-zagolovochnye-fajly-iz-si-v-s/zagolovochnyj-fajl-cstdio-stdio-h/funkciya-perror/).

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, в будущем, 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.

**Блочный ввод-вывод**

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.

В случае успешного считывания информации функция возвращает **число** прочитанных элементов (а не байтов), иначе – **EOF**.

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.

В случае успешной записи информации функция возвращает число записанных элементов, иначе –  **EOF**.

Примеры:

**Записать** в файл переменную типа **long** начиная с позиции **20**

*long d;*

*fseek (fd, 20L, SEEK\_SET);*

*fwrite (&d, sizeof(long), 1, fd);*

**Добавить** в файл переменную типа **student**

*struct student*

*{ char \*name;*

*int id, age;*

*char pol;*

*} ;*

*struct student sb;*

*fseek (fd, 0L, SEEK\_END);*

*fwrite (&sb, sizeof b, 1, fd);*

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

**Использование функций getc(), putc(), fgetc() и fputc().**

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.

И одно последнее замечание: на самом деле **getc()** и **putc()** являются не истинными функциями, а макросами. Функции **fgetc()** и **fputс()** идентичны их аналогам макросам **getc()** и **putc().**

* 1. **Прототип функции putc:**

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

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

Символ записывается в текущую позицию в потоке, на которую указывает внутренний индикатор положения, после чего,  индикатор положения инкрементируется на один символ.

* 1. **Прототип функции getc:**

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

Функция возвращает символ из потока filestream, на который ссылается внутренний индикатор позиции файла. Внутренний индикатор позиции в файле, после срабатывания этой функции сдвигается на один символ, таким образом, теперь он указывает на следующий символ.

* 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. Внутренний индикатор позиции файла уменьшается обратно, на предыдущее положение, так что этот символ возвращается при следующем вызове операции чтения для этого потока.

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.

Обратите внимание, что данная функция влияет только на следующую операцию чтения для данного потока, а не на содержание файла, связанного с потоком, который не изменяется при любом вызове этой функции.

Если внутренний показатель конца файла EOF был установлен, то после вызова этой функции он очищается.

* 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.

Функции fgets() и fputs() предназначены для ввода-вывода строк, они являются аналогами функций gets() и puts() для работы с файлами.

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

Возвращает строку из потока

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.

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

Записывает строку в поток

Копирует строку в поток с текущей позиции. Завершающий нуль- символ не копируется.

Функции **fscanf()** и **fprintf()** аналогичны функциям scanf() и printf(), но работают с файлами данных, и имеют первый аргумент — указатель на файл.

* 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. Для этого предназначены функции принудительной записи данных из буфера (прототипы хранятся в <stdio.h>:

**int fflush(указатель\_на\_поток);**- записывает информацию хранящуюся в буфере в файл, поток при этом не закрывается. Возвращает 0 в случае успеха и EOF если ошибка. Не работает с небуферизированными потоками. **fflush(stdin)**– очистить входной поток.

Для управления размерами буфера можно использовать следующие функции (вместо автоматической установки буфера):

**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. Он должен иметь длину не менее BUFSIZ байт. BUFSIZ — макро-константа, определенная в <cstdio> специально разработана для указания длинны массива. Кроме того, указатель NULL  может быть задан для отключения буферизации.
* **type**Режимы буферизации файлов:

|  |  |
| --- | --- |
| \_IOFBF | **Полная буферизация:**На выход, данные записываются после заполнения буфера. Входной буфер заполняется при открытии файла и, если буфер пуст. |
| \_IOLBF | **Строковая буферизация:** На выход, данные записываются, когда символ новой строки вставляется в поток или когда буфер полон. На входе, буфер заполняется до следующего символа новой строки, когда запрашивается операция ввода, при условии, что буфер пуст. |
| \_IONBF | **Без буферизации:** Нет буфер используется. Каждая операция ввода / вывода написано как можно скорее. В этом случае, thebuffer и размер параметры игнорируются. |

* **size**Размер буфера в байтах. Если аргумент параметра size - NULL, то автоматически выделяется минимальный размер буфера, в противном случае оно должно быть равным или меньше размера в байтах массива, заданного в качестве буфера.

**Стандартные потоки (putchar, getchar, puts, gets, perror).**

Когда программа начинает выполняться, автоматически открываются пять потоков, из которых основными являются:

−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 отдельных символов в стандартные потоки.**

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. При обращении она возвращает в вызвавшую ее функцию один введенный символ.

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**". Если при чтении из файла с помощью функции **getchar()** может быть достигнут конец файла. В этом случае операционная система в ответ на попытку чтения символа передает функции **getchar()** значение **EOF (End of File).** Константа **EOF** определена в заголовочном файле **stdio.h** и в разных операционных системах имеет значение **0** или -**1**.

Применение константы **EOF**вместо конкретных целых значений, возвращаемых при достижении конца файла, делает программу мобильной (не зависимой от конкретной операционной системы).

В случае ошибки при вводе функция **getchar()**также возвращает **EOF.**

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

Приведем в качестве примера применения функций **getchar() и putchar(**) программу копирования данных из стандартного ввода в стандартный вывод, которую можно найти практически в любом пособии по программированию на Си:

**#include <stdio.h>**

**int main() {**

**int c;**

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

**putchar(c);**

**return 0;**

**}**

Для завершения приведенной выше программы копирования необходимо ввести с клавиатуры сигнал прерывания Ctrl+C (одновременно нажать клавиши <Ctrl> и <С>).

**Ввод-вывод строк в стандартные потоки**

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

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

Обе функции имеют только один аргумент - указатель s на массив символов. Если строка прочитана удачно, функция **gets()**возвращает адрес того массива s, в который производился ввод строки. Если произошла ошибка, то возвращается **NULL.**

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

Пример использования этих функций:

**#include <stdio.h>**

**int main() {**

**char name[80];**

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

**puts(strl);**

**gets(name);**

**return 0; }**

Напомним, что любая строка символов в языке Си должна заканчиваться нуль-символом**'\0'**. В последний элемент массива **strl** нуль-символ будет записан автоматически во время трансляции при инициализации массива. Для функции **puts(**) наличие нуль-символа в конце строки является обязательным. В противном случае, т.е. при отсутствии в символьном массиве символа **'\0'**, программа может завершиться аварийно, так как функция **puts()**в поисках нуль-символа будет перебирать всю доступную память байт за байтом, начиная в нашем примере с адреса **strl**. Об этом необходимо помнить, если в программе происходит формирование строки для вывода ее на экран дисплея. Функция **gets()**завершает свою работу при вводе символа '**\n'**, который автоматически передается от клавиатуры в **ЭВМ** при нажатии на клавишу **<Enter>**. При этом сам символ '**\n'** во вводимую строку не записывается. Вместо него в строку помещается нуль-символ'**\0'**. Таким образом, функция **gets()** производит ввод "**правильной**" строки, а не просто последовательности символов.

Здесь следует обратить внимание на следующую особенность ввода данных с клавиатуры. 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 которой в **stdio.h** имеет вид:

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

Функция **perror()** выводит строку символов, адресуемую указателем **s**, за которой размещаются: **двоеточие, пробел и сообщение об ошибке**. Содержимое и формат сообщения определяются реализацией системы программирования. Текст сообщения об ошибке выбирается функцией **perror()** на основании номера ошибки. Номер ошибки заносится в переменную **int errno** (определенную в заголовочном файле **errno.h**) рядом функций библиотеки языка Си, в том числе и функциями ввода-вывода.

**Служебные функции для работы с файлами (remove, rename, tmpfile, tmpnam).**

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

Функция **remove** удаляет файл, имя которого указано в параметре **fname**. Эта операция производится непосредственно с файлом, нет потоков, участвующих в операции удаления.

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

Функция **rename**изменяет имя файла или каталога, указанного в **oldfilename** на новое имя, указанное в **newfilename**. Если в **oldfilename и newfilename** указать разные пути, и, если это поддерживается системой, файл будет перемещен в новое место.  
Эта операция производится непосредственно с файлом, нет потоков, участвующих в операции.

В языке Си имеется возможность работы с временными файлами, которые нужны только в процессе работы программы и должны быть удалены после выполнения некоторых вычислений. В этом случае используется функция

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

которая создает на диске временный файл с правами доступа **w+b**. После завершения работы программы или закрытия этого (временного) файла он автоматически удаляется.

**char \* tmpnam( char \* string );**

Функция **tmpnam** создает имя для файла. Генерируется строка, содержащая имя файла, которое отличается от любого существующего имени файла.  
Эта строка может быть использована для создания временных файлов без перезаписи любого другого существующего файла.  
Если **string** аргументом является нулевой указатель, то результирующая строка сохраняется во внутренний статический массив, который может быть доступен через возвращаемое значение. Содержание строки хранится до следующего вызова этой же функции.

Если **string**аргумент не является нулевым указателем, то он должен указывать на массив размером, по крайней мере, **L\_tmpnam байт**, который будет заполнен, предлагаемым функцией **tempname**, именем. **L\_tmpnam** является макро-константа, определенная в **<cstdio>.**

Имя файла, возвращаемое этой функцией может быть использовано для создания регулярного файла, используя функцию **fopen**. Файл, созданный таким образом, в отличие от того, который создан как временный файл, автоматически не удаляется, при закрытии. Чтобы удалить такой файл, вы должны вызвать функцию **remove**.

# **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. При чтении количество прочитанных байтов может оказаться меньше числа, указанного в третьем аргументе. Нуль означает конец файла, а –1 сигнализирует о какой-то ошибке. При записи функция возвращает количество записанных байтов, и если это число не совпадает с требуемым, следует считать, что запись не произошла. За один вызов можно прочитать или записать любое число байтов. Обычно это число равно или 1, что означает посимвольную передачу без буферизации, или чему-нибудь вроде 1024 или 4096, соответственно размеру физического блока внешнего устройства. Эффективнее обмениваться большим числом байтов, поскольку при этом требуется меньше системных вызовов. Используя полученные сведения, мы можем написать простую программу, копирующую свой ввод на свой вывод и эквивалентную программе копирования файла, описанной в главе1. С помощью этой программы можно копировать откуда угодно и куда угодно, поскольку всегда существует возможностьперенаправить ввод или вывод на любой файл или устройство.

#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;

}

Прототипы функций, обеспечивающие системные вызовы, мы собрали в файле syscalls.h, что позволяет нам включать его в программы этой главы. Однако имя данного файла не зафиксировано стандартом.

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.

**Форматированный вывод**

Сегодня мы рассмотрим две важные функции форматированного ввода и вывода. Устройство и работу этих функций полностью можно понять только после изучения работы с указателями и функций с переменным числом параметров. Но пользоваться этими функциями необходимо уже сейчас, так что некоторые моменты придётся пропустить. 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>

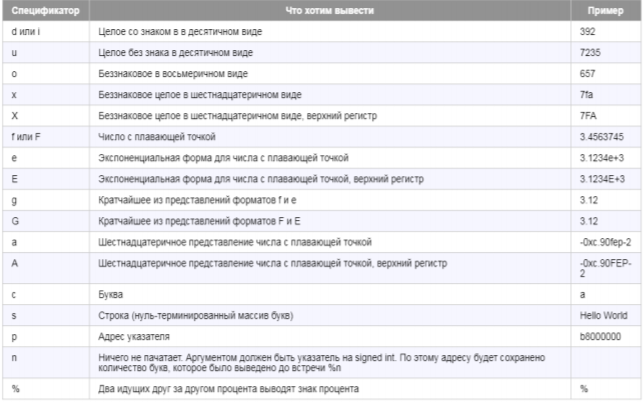
void main() {

//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. Общий синтаксис спецификатора формата %[ ][ ][. ][ ] флаги ширина точность длина спецификатор Спецификатор – это самый важный компонент. Он определяет тип переменной и способ её вывода.



**Форматированный ввод**

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();

} В си определён ряд символов, которые не выводятся на печать, но позволяют производить форматирование вывода. Эти символы можно задавать в виде численных значений, либо в виде эскейп-последовательностей: символа, экранированного обратным слешем. 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(). Если не принимать во внимание, что они перенаправляют вывод в файл, то можно считать, что работа данных функций полностью идентична printf() и scanf() соответственно. Хотя fprintf() u fscanf() часто являются наиболее простым способом записи и чтения данных из файла, они далеко не всегда являются наиболее эффективным решением. Поскольку отформатированные ASCII-данные сохраняются так же, как они выглядят на экране (а не в двоичном представлении), появляются дополнительные расходы при каждом вызове. Если скорость работы программы или размер создаваемых файлов играют важную роль, то, возможно, лучше ис- пользовать fread() и fwrite()

# **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.

// создаём объект для записи в файл

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

Для чего нам объект? Объект необходим, чтобы можно было выполнять запись в файл.

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

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

fout.close(); // закрываем файл

# **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.

Как и в языке С, в С++ существует несколько предопределенных потоков, открывающихся автома­тически вместе с началом выполнения программы. Ими служат cin, cout, cerr и clog. Как извест­но, *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.

Т.е. при запуске консольного приложения неявно открываются четыре потока:

|  |  |  |
| --- | --- | --- |
| 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) |

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 (соответственно для классов istream и ostream) можно перегрузить таким образом, чтобы можно было применять их для ввода или вывода объектов класса, определенного пользователем.

Давайте рассмотрим более подробно опрератор <<. В рассмотренном ранее случае синтаксис такой:

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;

В общем случае вторым параметром может быть любой стандартный тип. Это связано с тем, что этот ператор был перегружен для всех стандартных типов языка С++.

В случае оператора >> всё логично. Если хочется написать свой оператор <<, то нужно переопределить этот оператор в своём классе. Если же у нас есть класс комплексных чисел Complex, то вывод этих чисел через оператор << надо написать всего один раз на все случаи (экран, файл, принтер …)

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.

#include "stdafx.h"

#include <fstream>

using namespace std;

int main(int argc, char\* argv[])

{

    ofstream fout("file.txt"); // создаём объект класса ofstream для записи и связываем его с файлом cppstudio.txt

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

    fout.close(); // закрываем файл

    system("pause");

    return 0;

}

#include "stdafx.h"

#include <fstream>

#include <iostream>

using namespace std;

int main(int argc, char\* argv[])

{

    setlocale(LC\_ALL, "rus"); // корректное отображение Кириллицы

    char buff[50]; // буфер промежуточного хранения считываемого из файла текста

**ifstream fin("file.txt"); // открыли файл для чтения**

**fin >> buff; // считали первое слово из файла**

    cout << buff << endl; // напечатали это слово

    fin.getline(buff, 50); // считали строку из файла

    fin.close(); // закрываем файл

    cout << buff << endl; // напечатали эту строку

    system("pause");

    return 0;

}

Как и в языке С, в С++ существует несколько предопределенных потоков, открывающихся автома­тически вместе с началом выполнения программы. Ими служат cin, cout, cerr и clog.

|  |  |  |
| --- | --- | --- |
| 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) |

|  |  |  |
| --- | --- | --- |
| Класс | Метод | Описание |
| 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 результатами:  ......  cin.get();  return 0  }  Если указать третьим параметром символ-разделитель, cin.get() будет считывать либо сколько сказано символов, либо пока cin не встретит этот символ:   |  |  | | --- | --- | |  | cin.get(s,n,' '); |   (по сути, аналог getchar() из C) |
| 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.  Использование его переопределенной версии в хедере **string** выглядит так:  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;  }  В принципе, в write() можно передавать указатель на любой блок памяти, но для вывода в консоль характерны только массивы читаемых, понятных человеку символов.  Ð±Ð¸Ð±Ð»Ð¸Ð¾ÑÐµÐºÐ° iostream, ÐºÐ»Ð°ÑÑ cin, ÐºÐ»Ð°ÑÑ cout, cout.write |

|  |  |  |
| --- | --- | --- |
| Класс | Метод | Описание |
| cout | width() | Метод**width()** задает ширину выводимого, если необходимо выровнять до определенного количества символов. Как правило применяется при построении таблиц. Типичный пример: Вывод таблицы вычисления формулы. |
| 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++).*

Во всех компиляторах С имеются несколько описанных в стандарте ANSI С функций, предназначенных для ввода и вывода символов. Эти функции обеспечивают стандартный ввод и вывод и рассматриваются как высокоуровневые процедуры (в отличие из низкоуровневых функций, которые более непосредственно обращаются к аппаратному обеспечению компьютера). Ввод/вывод в С реализован не при помощи ключевых слов, являющихся частью языка, а с использованием функций, поставляемых разработчиками компилятора.

В языке C++ форматированный вывод возможен при помощи вывода на экран специальной управляющей команды:   
  
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.setf(std::ios::hex);

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

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.

Для наглядности небольшой пример:

int apples = 5;

int oranges = 7;

std::string str = format("I have %d apples and %d oranges, so I have %d fruits", apples, oranges, apples + oranges);

std::cout << str << std::endl;

Здесь:  
*Строка-шаблон:* I have %d apples and %d oranges, so I have %d fruits  
*Местозаполнители:*  %d, %d, %d  
*Аргументы:* apples, oranges, apples + oranges

При выполнении примера, получаем результирующую строку:

I have 5 apples and 7 oranges, so I have 12 fruits

Теперь посмотрим, что же нам предоставляет C++ для строкового форматирования.

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

char buf[100];

int res = snprintf(buf, sizeof(buf), "I have %d apples and %d oranges, so I have %d fruits", apples, oranges, apples + oranges);

std::string str = "error!";

if (res >= 0 && res < sizeof(buf)) str = buf;

std::cout << str << std::endl;

Это довольно неплохой способ форматирования, несмотря на кажущуюся неуклюжесть:

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

Но, конечно, не обошлось и без недостатков:

* 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

Начиная с C++11 в стандартной библиотеке появилась функция std::to\_string(), которая позволяет преобразовать передаваемое значение в строку. Функция работает не со всеми типами аргументов, а только со следующими:

* int
* long
* long long
* unsinged int
* unsinged long
* unsigned long long
* float
* double
* long double

# **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.

Как и в языке С, в С++ существует несколько предопределенных потоков, открывающихся автома­тически вместе с началом выполнения программы. Ими служат cin, cout, cerr и clog. Как извест­но, *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.

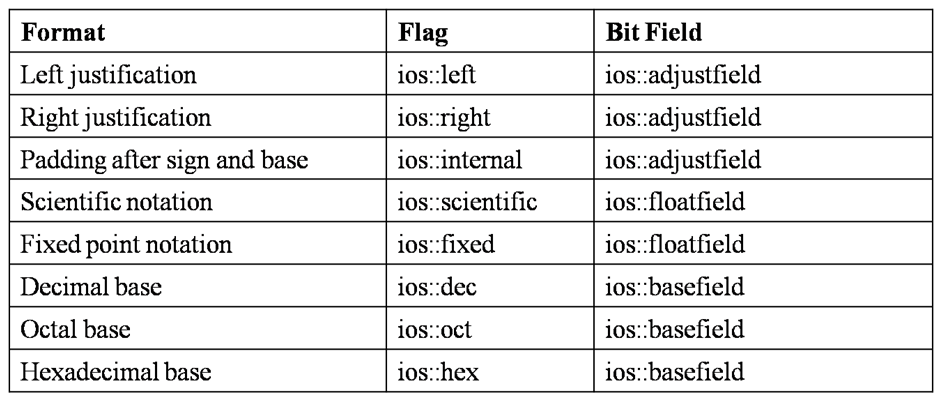
|  |  |
| --- | --- |
| ios\_base | *«InputOutputStream\_Base»*, базовый класс всей иерархии классов потоков. Содержит общие функции, типы и классы, в основном представляющие собой флаги (индикаторы). Эти флаги используются функциями подклассов fstream и могут быть определены с помощью функций ios\_base. |
| 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. В классе ios объявляется перечисление fmtflags, в котором определены следующие значения (Точнее, эти значения определены в классе ios\_base, который является базовым для ios):

|  |  |  |  |
| --- | --- | --- | --- |
| adjustfield | floatfield | showpoint | skipws |
| basefield | hex | oct | unitbuf |
| boolalpha | internal | showbase | uppercase |
| dec | left | right |  |
| fixed | showpos | scientific |  |

*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. (Если компилятор старый, то флаги форматирования fmtflags кодируются как целочисленные long-значения).

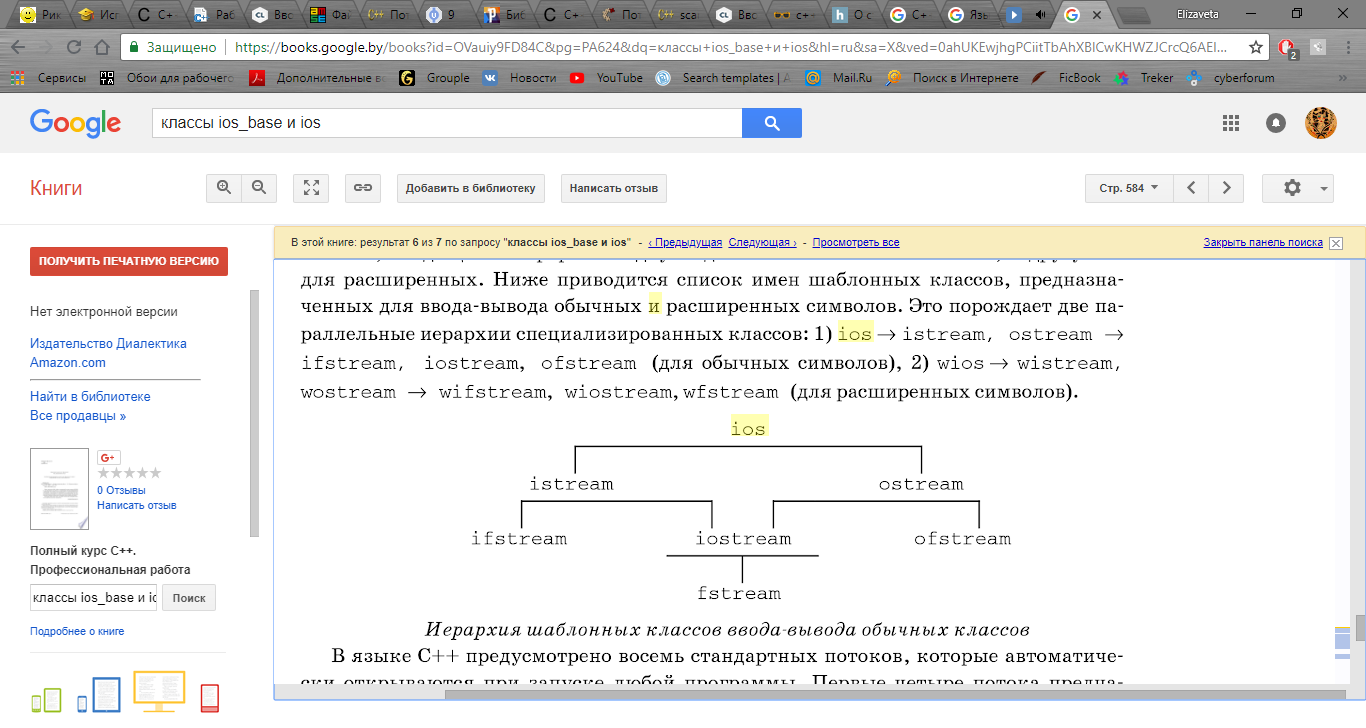
Для установки любого флага используется функция setf(), которая является членом класса ios. В самом общем виде её формат выглядит так:

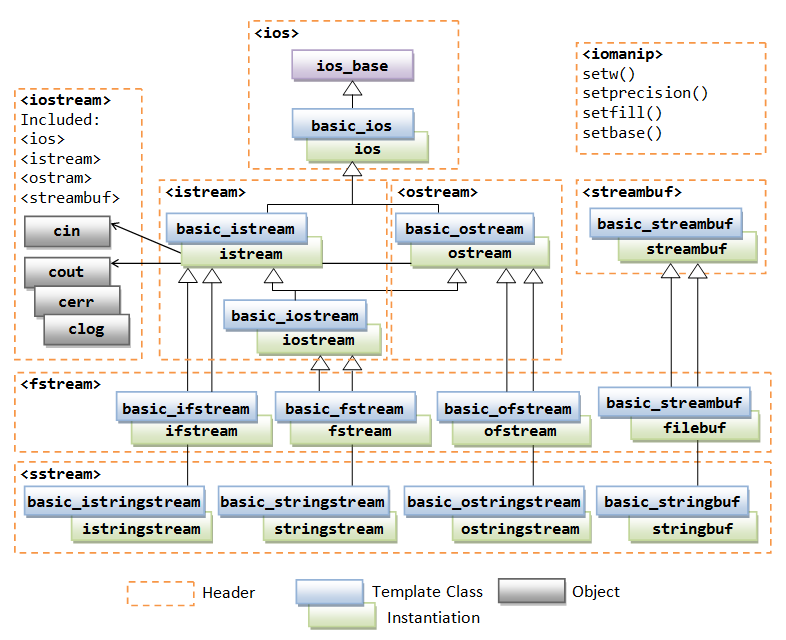
fmtflags setf (fmtflags flags);

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);

Здесь элемент stream значит поток, параметры форматирования которого вы хотите изменить. Обратите внимание на использование префикса ios:: для уточнения «классовой» принадлежности параметра showbase. Поскольку параметр showbase представляет собой перечислимую константу, определённую в классе ios, то при обращении к ней необходимо указывать имя класса ios. Этот принцип относится ко всем флагам форматирования.



Или немного более точная схема: 

* open(). Этим методом можно открыть заданный файл, сопоставив его с одним из объектов потока. В зависимости от передаваемых [аргументов](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), файл может быть открыт для чтения, для записи (либо для полной, либо для добавления данных), как [бинарный](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%BD%D0%B0%D1%80%D0%BD%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB), или как текстовый файл.
* is\_open(). Функция, определяющая, открыт ли в данный момент файл, которому сопоставлен определенный объект потока. Возвращает [булевское](https://ru.wikipedia.org/wiki/%D0%91%D1%83%D0%BB%D0%B5%D0%B2%D1%81%D0%BA%D0%B8%D0%B9_%D1%82%D0%B8%D0%BF) значение. Используется в основном для предотвращения ошибок доступа при попытке открыть уже использующийся файл. Без аргументов.
* 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.

*Простыми словами:*

Поток вывода как правило буфферизуется. Т.е. ты вызываешь printf ("abc"); но реально печати нет, далее пишешь printf ("def"); но реально печати нет, потому как текст копится в буффере. Затем по какому-то условию буффер печатается целиком (т.е. "abcdef"). Условия в каждой конкретной реализации могут быть разные: начиная от переполнения буффера и заканчивая наличием enter’а в печатаемой строке. Если вызвать fflush (stdout), то произойдёт принудительный сброс буфферов (т.е. напечатается "abcdef")

Функция put() выводит одиночный символ в стандартное устройство вывода (cout, если оно не было переназначено, то это экран).

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. При каждом выполнении операции ввода или вывода соответствующий указатель автоматически перемещается, последовательно изменяя свою позицию. Однако, используя функции seekg() и seekp(), можно получить доступ к файлу непоследовательным образом.

Версия функции seekp() с двумя параметрами перемещает указатель вывода на offset байт от позиции, заданной параметром origin. Параметр offset имеет тип streamoff, который определен в заголовке iostream.h. Объект типа streamoff способен содержать самое большое допустимое значение, которое может иметь параметр offset.

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 |

Версия функции seekp() с одним параметром перемещают указатели файлов в позиции, заданные параметром position. Это значение должно быть предварительно получено путем обращения либо к функции tellg(), либо к функции tellp() соответственно.

[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

Вставляет первые n символов массива, на которые указывает s в поток.   
Эта функция просто копирует блок данных, не проверяя его содержимое: массив может содержать нулевые символы, которые также копируются без остановки процесса копирования.

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 );
* istream& istream::get( char\* buffer, streamsize num, char delim );
* istream& istream::get( streambuf& buffer );
* istream& istream::get( streambuf& buffer, char delim );

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 (delim не читается до следующего вызова функции),
* считывает символы в буфер, пока не достигнет конца строки, или конца файла, или символа delim (который считается при следующем вызове функции).

Функция getline

Синтаксис:

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

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

Функция **getline()** используется с потоками ввода. Версия функции без ограничителя delim ставит ограничитель на символ новой строки. 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.

Если не удалось прочитать символ **delim** (**обычно это символ новой строки**), то поток ввода переходит в [**состояние ошибки**](http://ru.cppreference.com/w/cpp/io/fail).

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. Она считывает и выбрасывает символы, пока не считает **num** символов (**по умолчанию num равен 1**) или пока не считает символ-ограничитель **delim** (по умолчанию **delim равен** **EOF**). Функция **ignore()** иногда полезна при использовании вместе с функцией **getline()** и оператором **>>.** Например, если вы считываете некоторые данные, которые следуют за символом новой строки, с помощью **оператора >>,** символ новой строки останется не считаным и будет считан следующим. Так как **getline()** по умолчанию останавливает считывание по достижению символа новой строки, последующий вызов функции **getline()** возвратит пустую строку. По этой причине следует вызывать **ignore()** до **getline(),** чтобы выбросить символ новой строки.

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. Если достигнут конец файла **EOF**, **read()** останавливается, оставляя текущее количество байтов в буфере.

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. При каждом выполнении операции ввода или вывода соответствующий указатель автоматически перемещается, последовательно изменяя свою позицию. Однако, используя функции **seekg()** и **seekp()**, можно получить доступ к файлу непоследовательным образом.

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 (если не изменится позиция ввода).

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

Если символ не удается вернуть или функция **putback()** пытается вернуть другой символ, устанавливается флаг **badbit,** что может привести к выдаче соответствующего исключения.

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. Эти флаги унаследованы всеми потоками из базового класса **ios**. Флаги форматирования реализованы в виде отдельных фиксированных битов и хранятся в **protected** компоненте класса **long x\_flags.** Для доступа к ним имеются соответствующие **public** функции.

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. В стандартной библиотеке определено три класса строковых потоков:

**istringstream** – входные строковые потоки;

**ostringstream** – выходные строковые потоки;

**stringstream** – двунаправленные строковые потоки.

Эти классы определены в заголовочном файле **<sstream>** и являются производными от классов **istream**,**ostream** и **iostream** соответственно.

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

**strstream.h** — резидентные 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**

**заканчивается символом '\0', то размер буфера указывать не обязательно. При записи в буфер вывода в конце нужно добавить символ '\0' для правильной работы метода ostrstream::str.**

**Если при создании выходного резидентного потока не указывать буфер, то автоматически будет создан динамический буфер, изменяющий размер во время исполнения:**

**ostream os1; // резидентный поток с динамическим буфером**

**char Buf[100];**

**ostream os2(Buf, siseof(Buf)); // резидентный поток со статическим буфером**

Буферы и синхронизация

Потоки выполняют обмен данными между прикладной программой и файлом через буферы, которые являются объектами классов **filebuf** и **stringbuf** соответственно. Эти классы являются производными от абстрактного класса **streambuf**. 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. Неявно синхронизация выполняется в случае закрытия файла или заполнения буфера. Для явной синхронизации используют манипуляторы flush и endl или функцию sync.

# **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.

Чтобы выяснить, в каком состоянии находится поток, достаточно проверить его значение на истину:

**if ( !cin )**

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

Для чтения заранее неизвестного количества элементов мы обычно пишем цикл while:

**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.

*\* У любого потока есть набор флагов, с помощью которых можно следить за состоянием потока. Имеются четыре предикатные функции-члена:* eof() *возвращает* true*, если достигнут конец файла:*

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

**// отлично: все прочитано ...**

\* bad() возвращает true при попытке выполнения некорректной операции, например при установке позиции за концом файла. Обычно это свидетельствует о том, что поток находится в состоянии ошибки;

\* 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. Например, в коде оператора ввода для класса WordCount при обнаружении неверного формата мы используем setstate() для установки флага fail в состоянии объекта istream:

**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 );**

При тестировании оператора ввода в классе WordCount (см. раздел 20.5) мы писали:

**if ( !cin ) {**

**cerr "Ошибка ввода WordCount" endl;**

**return -1;**

**}**

Возможно, вместо этого мы предпочли бы продолжить выполнение программы, предупредив пользователя об ошибке и попросив повторить ввод. Но перед чтением нового значения из потока cin необходимо перевести его в нормальное состояние. Это можно сделать с помощью функции-члена clear():

**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. (Оба вызова эквивалентны, поскольку goodbit является для clear() аргументом по умолчанию.)

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. По существу, двоичный файл является аналогом внутренней (оперативной, физической) памяти – неограниченным массивом байтов с возможностью непосредственного обращения (произвольного доступа) к любой его части.

Такая модель файла полностью совпадает с системой представлений, принятой в Си для работы с памятью на низком (физическом уровне).

* 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. Преобразование типа указателя к void\* позволяет интерпретировать его как «чистый» адрес, а преобразование к char\* - как указатель на массив байтов (физическое представление 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. Функции используются для перенесения данных из файла в память программы (чтение) и обратно (запись).

**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 (неизвестен) характер структуры данных в той области памяти, в которую осуществляется ввод-вывод (указатель void\* buf).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.

Чтобы воспользоваться этими функциями, необходимо обеспечить преобразования переменных к «массиву байтов», используя указатели для задания адресов и операцию sizeof для вычисления размерности:

**// Прочитать  целую переменную и следующий за ней**

**// динамический массив из n переменных типа double**

**int n;                                         // в целой переменной – размерность массива**

**fread(&n, sizeof(int),1,fd);            // указатель на переменную int**

**double   \*pd = new double[n];**

**fread(pd, sizeof(double),n,fd);      // преобразование к****void\* - неявное**

Дальнейшее изложение приходится начинать с банальности: при использовании исключительно функций fread/fwrite данные, записанные в определенной последовательности в файл, хранятся в нем и читаются в том же самом порядке. Этот неизменный порядок извлечения данных называется последовательным доступом, а файл -  последовательным двоичным файлом. Естественно, что нас при этом не интересуют адреса размещения данных в файле. Однако существует и другой способ, позволяющий извлекать данные в любом произвольном порядке – прямой (или произвольный) доступ.

Произвольный доступ базируется на понятии адреса в двоичном файле. Поскольку на физическом уровне двоичный файл представляется как «неограниченно растущий» массив байтов, то под адресом понимается порядковый номер байта, начиная с 0.

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

**extern**

Поскольку С позволяет выполнять раздельную компиляцию модулей для большой программы в целях ускорения компиляции и помощи управлению большими проектами, должны быть способы передачи информации о глобальных переменных файлам программы. Решение заключается в объявлении всех глобальных переменных в одном файле и использовании при объявлении в других файлах слова extern, как показано в таблице

| **Файл 1** | **Файл 2** |
| --- | --- |
| int х, у; char ch;  main(void) { ... }  void func1 (void) { x =  23; } | extern int x, y; extern char ch;  void func22(void) { х= у/ 10; }  void func23(void) { y=10; } |
| Таблица. Использование глобальных переменных в раздельно компилируемых файлах | |

В файле 2 список глобальных переменных копируется из файла 1 и при объявлении добавляется спецификатор extern. Спецификатор extern сообщает компилятору, что следующие за ним типы и имена переменных объявляются где-то в другом месте. Другими словами, extern позволяет компилятору знать о типах и именах глобальных переменных без действительного создания этих переменных. Когда два модуля объединяются, все ссылки на внешние переменные пересматриваются.

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. Следующий фрагмент программы демонстрирует, как использовать данный вариант.

**int first, last; /\*глобальное определение first и last \*/  
int main (void)  
{  
extern int first; /\* необязательное использование extern объявления \*/  
......  
}**

Хотя объявление переменной с extern может иметь место в одном файле с объявлением глобальной переменной, в этом на самом деле нет необходимости. Если компилятор С встречает переменную, которая не была объявлена, то компилятор проверяет, соответствует ли она какой-либо глобальной переменной. Если это так, то компилятор предполагает, что эта переменная ссылается на глобальную.

# **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 (без параметров) и вызовы (с параметрами) можно использовать в качестве правого операнда для операции обмена** (**<<** или **>>**). В качестве левого операнда в этом выражении, как обычно, используется поток (ссылка на поток), и именно на этот поток оказывает влияние манипулятор.

    К примеру, напомним, что манипулятор **hex** позволяет устанавливать шестнадцатеричное основание счисления выводимых в поток **cout** числовых значений. Выполнив последовательность операторов:

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

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

получим на экране:

Десятичное число: 15

Шестнадцатеричное представление: 0xF

    Как наглядно показывает результат, применение манипулятора **hex** изменило одно из свойств потока **cout**. Десятичная целая константа 15 воспринимается и выводится на экран в шестнадцатеричном виде.

    В качестве параметра каждый манипулятор автоматически (без явного участия программиста) получает ссылку на тот поток, с которым он используется в выражении. После выполнения манипулятора он возвращает ссылку на тот же поток. Поэтому манипуляторы можно использовать в цепочке включений в поток или извлечений из потока. При выполнении манипулятора никаких обменов данными с потоком не происходит, но манипулятор изменяет состояние потока. Например, выполнив оператор с манипуляторами **hex, oct, dec**:

cout << 15 << hex << 15 << oct << 15 << dec << 15;

в качестве результата получим:

150xF01715

    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** - действует только при вводе и предусматривает извлечение из входного потока пробельных символов (пробел, знаки табуляции **'\t'** и **'\v'**, символ перевода строки **'\n'**, символ возврата каретки **'\r'**, символ перевода страницы **'\f'**);
* **endl** - действует только при выводе, обеспечивает включение в выходной поток символа новой строки и сбрасывает буфер (выгружает содержимое) этого потока;
* **ends** - действует только при выводе и обеспечивает включение в поток нулевого признака конца строки;
* **flush** - действует только при выводе и очищает выходной поток, т.е. сбрасывает его буфер (выгружает содержимое буфера).

    Обратите внимание, что не все перечисленные манипуляторы действуют как на входные, так и на выходные потоки. 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. Например, его использование просто необходимо в таком операторе:

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.

    Рекомендуется с помощью манипулятора **flush** сбрасывать буфер входного потока при выводе на экран подсказки до последующего ввода информации:

cout << "Введите название файла: " << flush;

cin >> fileName; **// Здесь fileName - символьный массив.**

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

**setbase(int n)**

устанавливает основание (**n**) системы счисления. Значениями параметра **n** могут быть: **0**, **8, 10** или **16**. При использовании параметра 0 основание счисления при выводе выбирается десятичным. При вводе параметр 0 означает, что целые десятичные цифры из входного потока должны обрабатываться по правилам стандарта **ANSI** языка **С**;

**resetiosflags(long L)**

сбрасывает (очищает) отдельные флаги состояния потоков ввода и вывода на основе битового представления значения параметра **L** (сбрасывает флаги в соответствии с единичными битами);

**setiosflags(long L)**

устанавливает отдельные флаги состояния (форматные биты) потоков ввода-вывода на основе битового представления значения параметра **L** (устанавливаются флаги в соответствии с единичными битами параметра);

**setfill(int n)**

значение параметра **n** в дальнейшем используется в качестве кода символа-заполнителя, который помещается в незанятых позициях поля при вводе значения (компонент **х\_fill** класса [**ios**](http://it.kgsu.ru/C_STREAM/oop_0006.html#1));

**setprecision(int n)**

определяет с помощью значения параметра **n** точность представления вещественных чисел, т.е. максимальное количество цифр дробной части числа при вводе и выводе (компонент **x\_precision** класса [**ios**](http://it.kgsu.ru/C_STREAM/oop_0006.html#1));

**setw(int n)**

значение параметра **n** задает минимальную ширину поля вывода (компонент **x\_width** класса [**ios**](http://it.kgsu.ru/C_STREAM/oop_0006.html#1)).

    С помощью манипуляторов можно управлять представлением информации в выходном потоке. Например, манипулятор **setw(int n)** позволит выделить для числового значения поле фиксированной ширины, что удобно при печати таблиц.

    Итак, для управления форматом вывода (включения в поток) класс **ios** имеет следующие компоненты:

* компонент (атрибут) [**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 (вставляемые в цепочки обмена).

# 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. Вызов fclose() освобождает блок управления файлом, связанный с потоком, и делает его доступным для повторного использования.

Функция fclose() имеет прототип:

**int fclose(FILE \*fp);**

где fp - это указатель на файл, возвращенный fopen(). Если возвращен 0, то это означает, что операция закрытия выполнена успешно, а если EOF, то, значит, была ошибка.

**int fcloseall(void)**

закрывает все открытые потоки ввода-вывода. за исключением stdin, stdout, stderr, stdaux, stdprn. Возвращает общее число потоков, закрытых функцией, либо EOF, если обнаружены какие-либо ошибки при закрытии файлов.

**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.

При успешном завершении fdopen возвращает значение заново открытый поток stream. В случае ошибки, функция возвращают ноль (NULL).

**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.

Функция ferror возвращает ненулевое значение, если в потоке stream была обнаружена ошибка.

**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.

**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.

При успешном завершении fgetpos возвращает 0. При ошибке возвращает ненулевое значение и устанавливает errno в EBADF или EINVAL.

# 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. Символы, формирующие режим, приведены в табл.3.1. В случае успеха функция возвращает указатель на описание открытого потока, которое будет совпадать с stream. В противном случае функция возвращает NULL. Повторное открытие того же самого файла изменяет права доступа, очищает внутренние буферы и позиционирует указатель записи-чтения либо на начало файла (r, w, r+, w+), либо на его конец (а, а+). Если файлы разные, переоткрытие приводит к переадресации файлового ввода-вывода. Эта функция обычно применяется для переадресации предоткрытых потоков stdin, stdout, stderr, stdaux, stdprn в файлы, определяемые пользователем.

**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(). Ниже приведены адаптированные прототипы системного вызова open().

**int open (const char \* filename, int flags, mode\_t mode);**

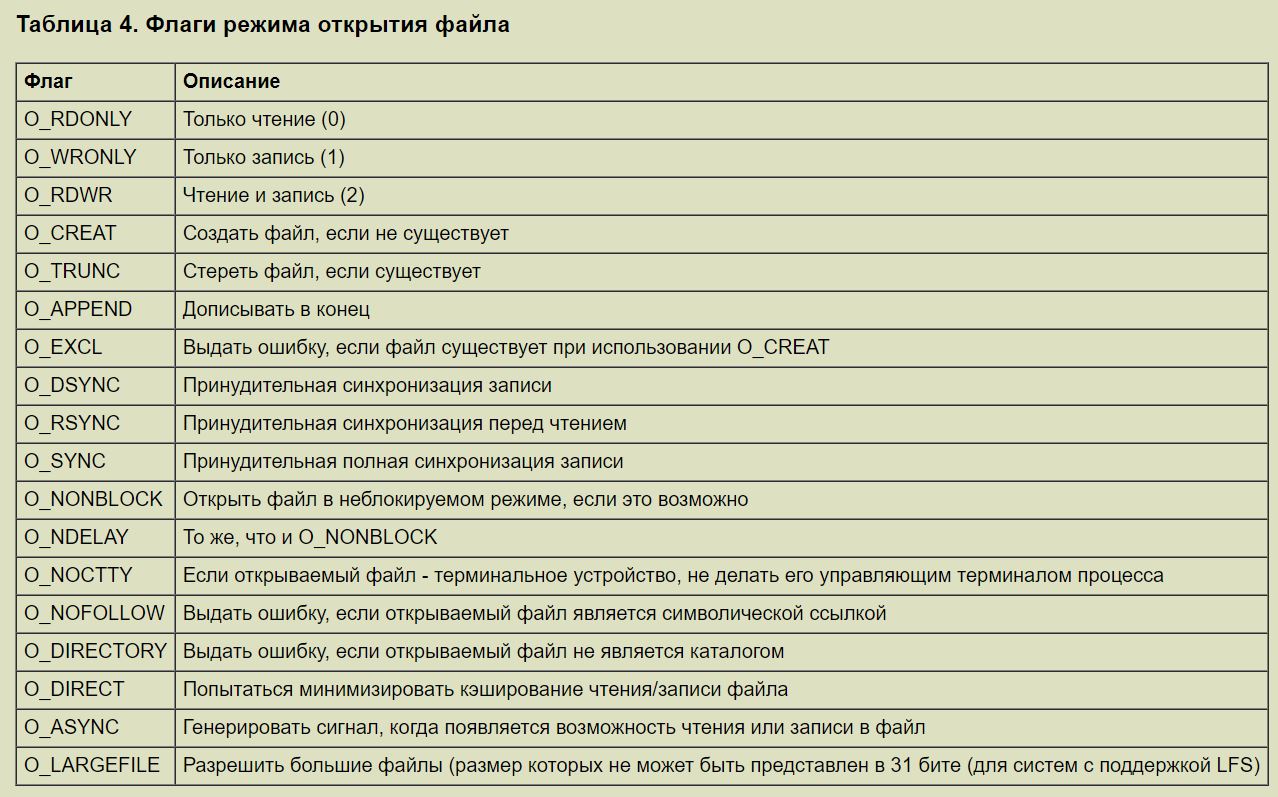
**int open (const char \* filename, int flags);**

Системный вызов open() объявлен в заголовочном файле fcntl.h. Ниже приведен общий 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.



Если вы хотите, например, открыть файл в режиме чтения и записи, и при этом автоматически создать файл, если такового не существует, то второй аргумент open() будет выглядеть примерно так: **O\_RDWR|O\_CREAT**.

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.

Чтобы, например, созданный файл был доступен в режиме "чтение-запись" пользователем и группой и "только чтение" остальными пользователями, - в третьем аргументе open() надо указать примерно следующее: **S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IWGRP|S\_IROTH** или **0664**.

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. Попытка открыть несуществующий файл является ошибкой Если произошла ошибка, то open() возвращает -1.

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. Системный вызов close() объявлен в файле unistd.h. Ниже приведен его адаптированный прототип. Завершение программы с помощью *exit* или *return* в главной программе закрывает все открытые файлы.

**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(), объявленный в файле unistd.h, позволяет читать данные из файла. Системный вызов read() читает данные в "сыром" виде, то есть как последовательность байт, без какой-либо интерпретации. Ниже представлен 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.

Хочу сказать несколько слов о типах. Тип size\_t используется для хранения размеров блоков памяти. Какой тип реально скрывается за size\_t, зависит от архитектуры; как правило это unsigned long int или unsigned int. Тип ssize\_t (Signed SIZE Type) - это тот же size\_t, только знаковый. Используется, например, в тех случаях, когда нужно сообщить об ошибке, вернув отрицательный размер блока памяти.

Теперь напишем программу, которая просто читает файл и выводит его содержимое на экран.

**#include <stdlib.h>**

**#include <stdio.h>**

**#include <unistd.h>**

**#include <fcntl.h>**

**#include <sys/stat.h>**

**#include <sys/types.h>**

**int main (int argc, char \*\* argv)**

**{**

**int fd;**

**ssize\_t ret;**

**char ch;**

**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 ((ret = read (fd, &ch, 1)) > 0)**

**{**

**putchar (ch);**

**}**

**if (ret < 0)**

**{**

**fprintf (stderr, "myread: Cannot read file\n");**

**exit (1);**

**}**

**close (fd);**

**exit (0);**

**}**

В этом примере используется укороченная версия open(), так как файл открывается только для чтения. В качестве буфера (второй аргумент read()) мы передаем адрес переменной типа char. По этому адресу будут считываться данные из файла (по одному байту за раз) и передаваться на стандартный вывод. Цикл чтения файла заканчивается, когда read() возвращает нуль (нечего больше читать) или -1 (ошибка). Системный вызов close() закрывает файл.

Ниже приведен исходный код программы, которая делает то же самое, что и предыдущий пример, но с использованием блочного чтения файла. Размер блока установлен в 64 байта.

**/\* 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);**

Как видите, прототип write() отличается от read() только спецификатором const во втором аргументе. В принципе write() выполняет процедуру, обратную read(): записывает count байтов из буфера buffer в файл с дескриптором fd, возвращая количество записанных байтов или -1 в случае ошибки. Так просто, что можно сразу переходить к примеру.

**#include <stdlib.h>**

**#include <stdio.h>**

**#include <unistd.h> /\* read(), write(), close() \*/**

**#include <fcntl.h> /\* open(), O\_RDONLY \*/**

**#include <sys/stat.h> /\* S\_IRUSR \*/**

**#include <sys/types.h> /\* mode\_t \*/**

**#define BUFFER\_SIZE 64**

**int main (int argc, char \*\* argv)**

**{**

**int fd;**

**ssize\_t read\_bytes;**

**ssize\_t written\_bytes;**

**char buffer[BUFFER\_SIZE];**

**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)**

**{**

**/\* 1 == stdout \*/**

**written\_bytes = write (1, buffer, read\_bytes);**

**if (written\_bytes != read\_bytes)**

**{**

**fprintf (stderr, "Cannot write\n");**

**exit (1);**

**}**

**}**

**if (read\_bytes < 0)**

**{**

**fprintf (stderr, "myread: Cannot read file\n");**

**exit (1);**

**}**

**close (fd);**

**exit (0);**

**}**

В этом примере нам уже не надо изощеряться в попытках вставить нуль-терминатор в строку для записи, поскольку системный вызов write() не запишет большее количество байт, чем мы ему указали.

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.

Как уже говорилось, с каждым открытым файлом связано число, указывающее на текущую позицию чтения-записи. При открытии файла позиция равна нулю. Каждый вызов read() или write() увеличивает текущую позицию на значение, равное числу прочитанных или записанных байт. Благодаря этому механизму, каждый повторный вызов read() читает **следующие** данные, и каждый повторный write() записывает данные **в продолжение** предыдущих, а не затирает старые. Такой механизм последовательного доступа очень удобен, однако иногда требуется получить произвольный доступ к содержимому файла, чтобы, например, прочитать или записать файл заново.

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);**

Первый вызов устанавливает текущую позицию в начало файла. Второй вызов смещает позицию вперед на 20 байт. В третьем случае текущая позиция перемещается на 10 байт назад относительно конца файла.

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. Для описания типа данных struct обязательно задание шаблонов. 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).

Допустим синтаксиси: struct book libry;

Язык С менее типизированный, чем С++, то для С++ можно использовать book libry;

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.

Если же нужно передать адрес отдельного члена структуры, то перед именем структуры должен находиться оператор &. Например, чтобы передать адреса членов структуры mike, можно написать следующее:

func(&mike.x); /\* передается адрес символа x \*/

func2(&mike.y); /\* передается адрес целого y \*/

func(&mike.s[2]); /\* передается адрес символа в s[2] \*/

Обратите внимание, что оператор & стоит непосредственно перед именем структуры, а не перед именем отдельного члена. И еще заметьте, что s уже обозначает адрес, поэтому & не требуется.

Когда в качестве аргумента функции используется структура, то для передачи целой структуры используется обычный способ вызова по значению. Это, конечно, означает, что любые изменения в содержимом параметра внутри функции не отразятся на той структуре, которая передана в качестве аргумента.

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);

}

Как видно из этой программы, при объявлении параметров, являющихся структурами, объявление типа структуры должно быть глобальным, чтобы структурный тип можно было использовать во всей программе. Например, если бы struct\_type был бы объявлен внутри main(), то этот тип не был бы виден в f1().

## 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]

}  
  
В результате получаем набор из 10 переменных, устроенных, как объявлено в типе структуры MyStruct.

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. Синтаксис прототипа функции при передаче структур посредством указателей и ссылок идентичен синтаксису обычной передачи параметров через указатели и ссылки:

// 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.

## 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 зачетки. 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).

Для хранения члена **group** выделено 4 бита, для **department** –  3 бита и для **course**  – 3 бита. Колич. битов определяется ожидаемым диапазоном значений для каждого члена структуры. Член структуры **group** хранит значения от 0 до 12  в области памяти размером 4 бита (4 бита, выдел. для элемента **group**, м. хранить значения от 0 до 15). Член структуры **department** м. хранить значения от 0 до 8 (факультеты). Область памяти размером 3 бита, выделенная для члена **course**, б. хранить значения от 0 до 4 (диапазон от 0 до 8).

Сгенерируем номера зачеток:

**number Fit[90];**

**……………………………….**

**void FNumber(number \* const doc)**

**{ for (int i = 0; i <= 99; i++)**

**{ doc[i].group = i % 3 + 9;**

**doc[i].department = 5;**

**doc[i].course = 1;**

**}**

Допустимы **неименованные** поля; они не влияют на смысл именованных полей, но с ними можно улучшить размещение.  Если Имя пропущено, то соответствующее количество бит не используется (пропускается ).

**struct  { unsigned a1 : 4;**

**unsigned  : 2;**

**unsigned a3 : 5;**

**unsigned a4 : 2;**

**} prim1;**

Длина структуры всегда **кратна восьми**.

Поле нулевой длины обозначает выравнивание на границу следующего слова:

**struct  { unsigned b1 : 1;**

**unsigned  : 0;**

**unsigned b3 : 5;**

**unsigned b4 : 2;**

**} prim2;**

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.

Хотя битовые поля сокращают требования к памяти, их использование может привести к тому, что компилятор будет генерировать машинный код, выполняющийся с **низкой скоростью**, т.к. придется использовать дополнительные операции машинного языка для получения доступа к отдельным частям адресуемых элементов памяти.

Пример. Ввод и печать информации о сотрудниках

**# include <iostream>**

**using namespace std;**

**enum Dep** // список отделов

**{ Manag = 1 , Law, Des, Prog } deps;**

**struct Date**

**{ unsigned short nDay : 5;** // 5 битов - 32

**unsigned short nMonth : 4;** // 4 бита - 16

**unsigned short nYear : 11;** // 11 бит - 2048

**} ;**

**struct Sotr** // структура для сотрудника

**{ char Name[30];** // ФИО

**int Dep;**  // Отдел

**Date Priem;** // Дата найма

**};**

**Sotr S[50];** // массив сотрудников

**char \*NameDep(int n)** //название отдела по номеру

**{ char\* dep;**

**switch(n)**

**{ case 1: dep="Менеджеры"; break;**

**case 2: dep="Юристы"; break;**

**case 3: dep="Дизайнеры"; break;**

**case 4: dep="Программисты"; break;**

**}**

**return dep;**

**}**

**void main()**

**{ setlocale(LC\_CTYPE, "Russian");**

**int i, d, m, y;**

**for(i = 0; i < 2; i++)** //2 сотрудника

**{ cout<< endl<<"Фамилия:"<<endl;**

**cin>>S[i].Name;**

**cout<<"Номер отдела: "<<endl;**

**cin>>S[i].Dep;**

**cout<<"День найма:"<<endl;**

**cin>>d;**

**S[i].Priem.nDay = d;**

**cout<<"Месяц найма:"<<endl;**

**cin>>m;**

**S[i].Priem.nMonth = m;**

**cout<<"Год найма:"<<endl;**

**cin>>y;**

**S[i].Priem.nYear = y;**

**cout<<endl;**

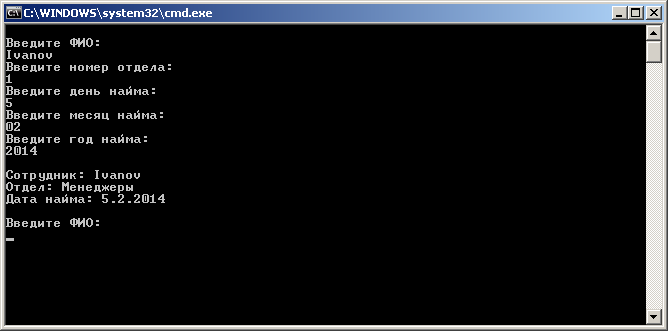
**cout<<"Сотрудник: "<<S[i].Name <<endl;**

**int n = S[i].Dep;**

**cout<<"Отдел:"<<NameDep(n)<<endl;**

**cout<<"Дата найма: "<<S[i].Priem.nDay << "."<<S[i].Priem.nMonth<<"."<<**

**S[i].Priem.nYear<<endl;**

 **}**

**}**

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.

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

В разные отрезки времени выполнения программы некоторые объекты могут быть не нужны. 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 –>

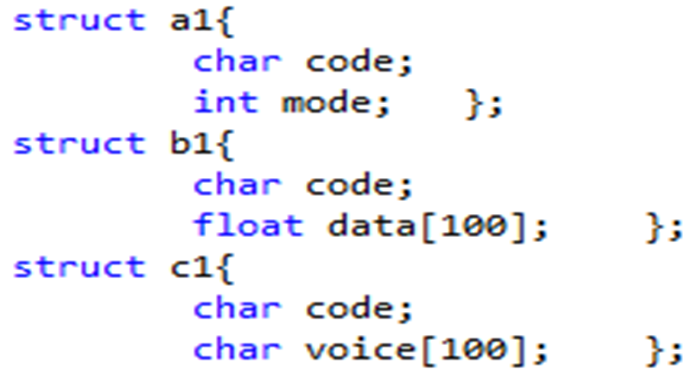
Над объединениями нельзя выполнять операции сравнения.

Свойства объединений:

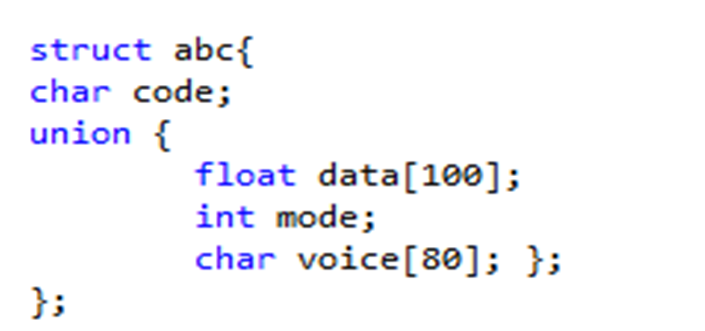
* размещение всех членов объединения начинается с одного и того же адреса;
* в любой момент времени можно присвоить значение лишь одному члену;
* не бывает статических членов;
* не бывает членов, являющихся ссылками.

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)**

И так, что же такое перечисление или, правильней сказать, перечисляемые типы в С++? Тема не из самых сложных. Само название — перевод с английского enumeration. А перечисляются целые константы. Простым для понимания сути вопроса будет такой пример: объявим перечисляемый тип годовщин свадьбы.

*#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 с Ольгой отмечают\t" << chintz << "-ю годовщину со дня свадьбы!!!";*

*cout << "\n";*

*cout << "Андрей с Анной отмечают\t" << paper << "-ю годовщину со дня свадьбы!!!";*

*cout << "\n";*

*cout << "Марк с Ириной отмечают\t" << leather << "-ю годовщину со дня свадьбы!!!";*

*cout << "\n";*

*cout << "Игорь с Юлией отмечают\t" << linen << "-ю годовщину со дня свадьбы!!!";*

*cout << "\n";*

*cout << "Олег с Аллой отмечают\t" << wooden << "-ю годовщину со дня свадьбы!!!";*

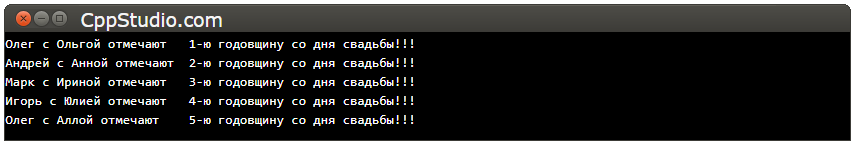
*cout << "\n\n";*

*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.

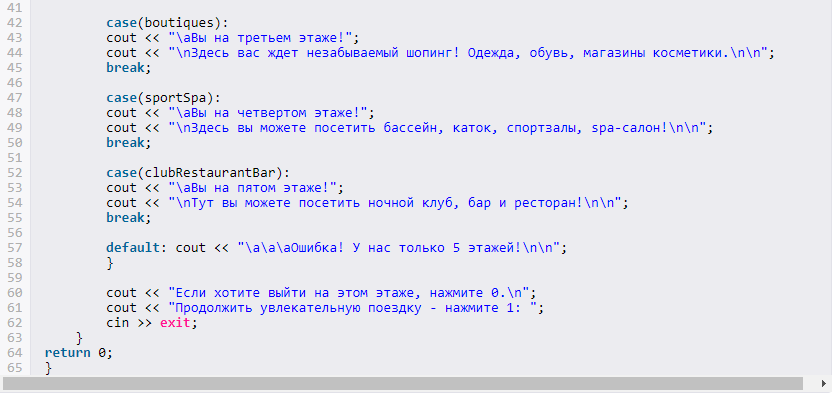
Определяя перечисление, объявление типа и переменной необязательно.

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.





## 

Структура tm имеет вид

struct tm

{

int tm\_sec; // секунды после минут [0,59]

int tm\_min; // минуты после часов [0,59]

int tm\_hour; // часы после полуночи [0,23]

int tm\_mday; // день месяца [1,31]

int tm\_mon; // месяц года (январь = 0) [0,11]

int tm\_year; // год (1900 год = 0)

int tm\_wday; // день недели (вс = 0) [0,6]

int tm\_yday; // день года (1 января = 0) [0,365]

int tm\_isdst; // флаг перехода на летнее время (>0- вкл.)

};

Функция

struct tm \*localtime(const time\_t \*ptm);

преобразует календарное время, указанное ptm, сохраняет его в структуре tm и возвращает указатель на нее.

Функция

time\_t mktime(struct tm \*timeptr);

осуществляет обратное преобразование.

Функция

clock\_t clock (void);

возвращает наилучшее приближение процессорного времени, прошедшего с момента запуска программы; для получения времени в секундах значение необходимо разделить на константу, определенную в библиотеке time.h:

#define CLOCKS\_PER\_SEC 1000

Если время не доступно или не может быть представлено, возвращает (clock\_t) (-1).

Функция

size\_t strftime(

char \*restrict s, // указатель на выходную строку

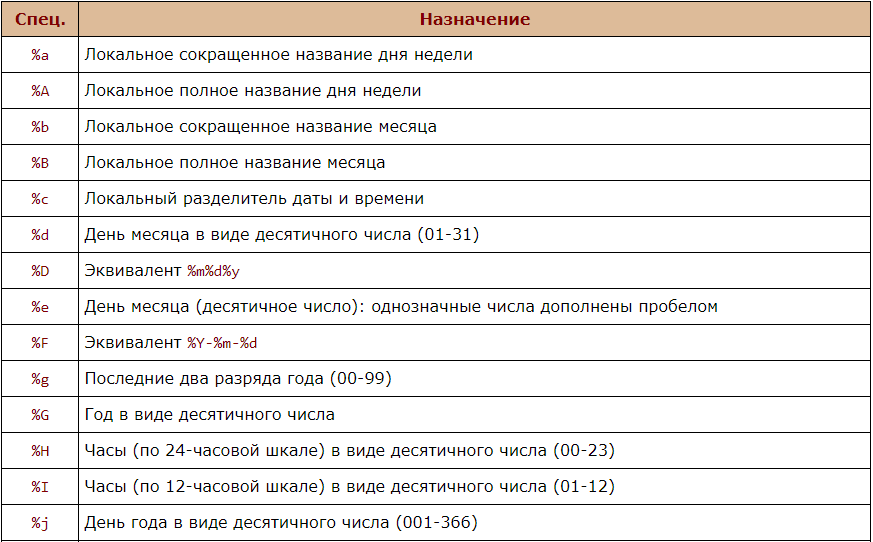
size\_t max, // максимальный размер выходной строки

const char \*restrict fmt, // указатель на строку формата

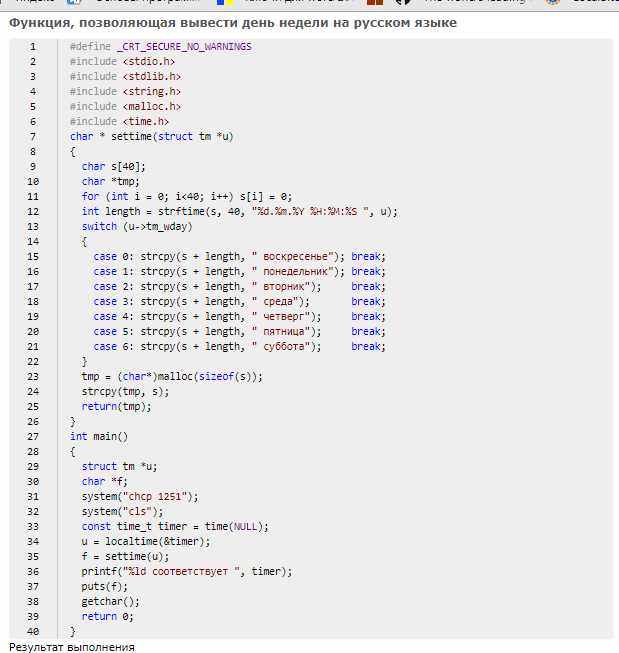
const struct tm \*restrict tmpt); // указатель на структуру tm

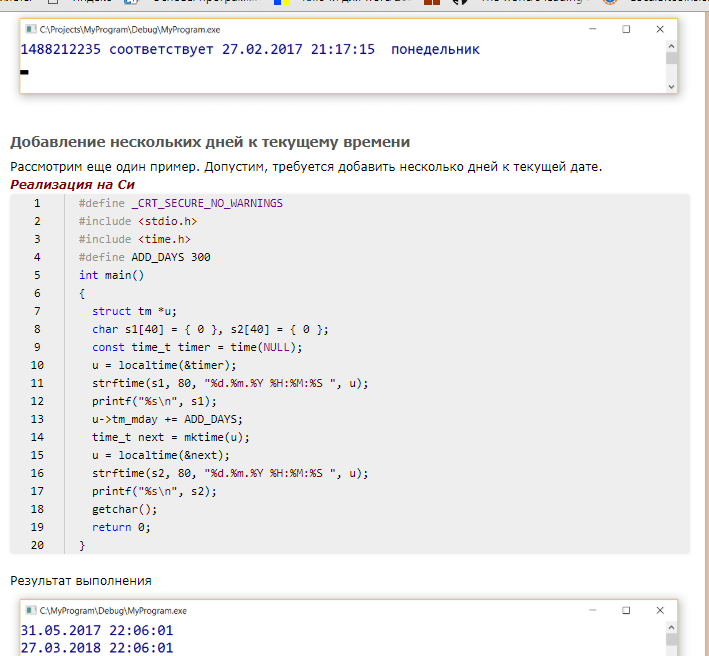
копирует строку fmt в строку s, заменяя спецификаторы формата в fmt соответствующими данными, взятыми из содержимого структуры времени, на которое указывает tmpt; в строку s помещается не более max символов.

Функция возвращает количество символов (исключая нулевой) в результирующей строке. Если результирующая строка (включая нулевой символ) содержит больше, чем max символов, функция возвращает 0, а содержимое s не определено.









55-61

# 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.

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

***Пример***. Сформировать список, содержащий целые числа 3, 5, 1, 9.

***Решение***. При работе с динамическими структурами данных можно рекомендовать следующий порядок действий.

а) Прежде всего необходимо определить две структуры:

1. структура, содержащая характеристики данных, то есть все те поля с данными, которые необходимы для решения поставленной задачи (в нашем случае имеется всего одно поле целого типа). Назовём эту структуру Data;
2. структура, содержащая поле типа Data и поле — адрес последующего элемента next. Вторую структуру назовём List.

Тексты этих структур необходимо расположить в начале программы (до main() и других функций). Вот возможная реализация структур:

struct Data

{

   int a;

};

struct List

{

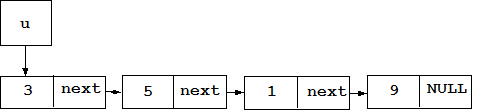
   Data d;

   List \*next;

};

Такой подход позволит в дальнейшем изменять в широких пределах структуру с собственно данными, никак не затрагивая при этом основную структуру List.

Итак, мы описали структурный тип, с помощью которого можно создать наш односвязный список. Графически создаваемый список можно изобразить так, как это показано на рисунке ниже:



б) В программе (обычно в функции main()) определяем указатель на начало будущего списка:

List \*u = NULL;

Пока список пуст, и указатель явно задан равным константе NULL.

в) Выполняем первоначальное заполнение списка.

Создадим первый элемент:

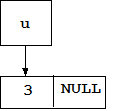
u = new List;  // Выделяем память под элемент списка

u->d.a = 3;    // Заполняем поля с данными

               // (здесь это всего одно поле)

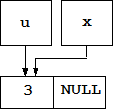
u->next = NULL;// Указатель на следующий элемент пуст

После включения первого элемента список можно изобразить так:

  
Продолжим формирование списка, добавляя новые элементы в его конец. Для удобства заведём вспомогательную переменную-указатель, которая будет хранить адрес последнего элемента списка:

List \*x;

x = u; // Сейчас последний элемент списка совпадает с его началом

  
Таким образом, к области памяти можно обратиться через два указателя.

Выделяем место в памяти для следующего элемента списка и перенаправляем указатель x на выделенную область памяти:

x->next = new List;

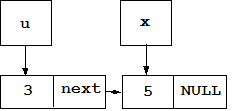
x = x->next;

Затем определяем значение этого элемента списка:

x->d.a = 5;

x->next = NULL;

Получилась такая схема:

  
Этот процесс продолжаем до тех пор, пока не будет сформирован весь список.

**Действия со сформированным списком**

Сформировав начальный список, можно выполнять с ним различные действия. Настоятельно рекомендуется каждую операцию со списком оформлять в виде отдельной функции. Такой подход заметно упрощает разработку программы и возможную её дальнейшую модификацию. Понятно, что и только что рассмотренное формирование начального списка также лучше записать в виде функции.

1. Просмотр списка — осуществляется последовательно, начиная с его начала. Указатель последовательно ссылается на первый, второй, и т.д. элементы списка до тех пор, пока весь список не будет пройден. Приведём пример реализации просмотра, например, для вывода списка на экран монитора:

void Print(List \*u)

{

   List \*p = u;

   cout << "Spisok:" << endl;

   while(p)

   {

      cout << p->d.a << endl;

      p = p->next;

   }

}

Обратиться к функции можно так:

Print(u);

Здесь и далее в примерах u — это указатель на начало списка.

2. Поиск первого вхождения в список элемента, соответствующего заданным требованиям:

List \* Find(List \*u, Data &x)

{

   List \*p = u;

   while(p)

   {

      if(p->d.a == x.a) // условие для поиска

         return p;

      p = p->next;

   }

   return 0;

}

Возможный вызов функции:

List \* v = Find(u, x);

где x — объект типа Data.

3. Добавление нового элемента в начало списка:

void Add\_Beg(List \*\*u, Data &x)

{

   List \*t = new List;

   t->d.a = x.a;

   t->next = \*u;

   \*u = t;

}

Возможный вызов функции:

Add\_Beg(&u, x);

где x — объект типа Data.

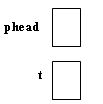
## Алгоритм построения списка с заглавным звеном.

    Чтобы сделать действия, выполняемые при включении элемента в список (исключении элемента из списка), единообразными, обычно применяется следующий прием. В начало каждого списка добавляется ***заглавное звено (заголовок списка)***. Оно никогда не исключается из списка и перед ним в список никогда не включаются новые элементы.

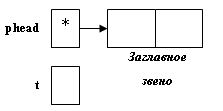
    Информационная часть заглавного звена или не используется вовсе, или используется для специальных целей. Например, в случае списка целых чисел она может содержать число, равное количеству звеньев в списке. Добавление заглавного звена в список приводит к тому, что теперь у всех элементов, в том числе и у первого, имеется предшественник, и действия по включению новых элементов в список (или исключение элементов из списка) проводятся единым способом.

    Приведем алгоритм построения однонаправленного списка с заглавным звеном с сохранением порядка поступления звеньев. Шаги построения алгоритма будем иллюстрировать с помощью схем "до и после" Д.Кнута.

1. Вначале отведем место для указателей в статической памяти:
2. struct node \*phead;
3. struct node \*t;

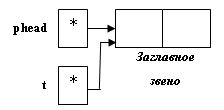
  
Рис.1. Резервирование места в памяти

1. В куче зарезервируем место для динамического объекта, на который указывает **phead**:
2. phead = new(node);

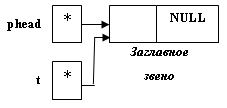
  
Рис.2. Выделили место для элемента списка

    Построено ***заглавное звено*** будущего однонаправленного списка.

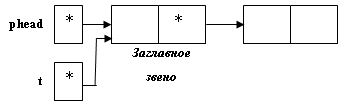
1. Выполним еще ряд подготовительных действий:
2. t = phead;

  
Рис.3. Присвоим значение **t**

(\*t).sled = NULL;

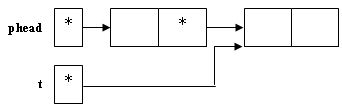
  
Рис.4. Присвоим указателю заглавного звена **NULL**

1. Теперь можно приступить к циклическому процессу построения списка. Идентификатор ***Число*** обозначает объект языка **C++**, вводимый с клавиатуры.
2. cin>>*Число*;
3. while (*Число* != *Числу, определяющему окончание ввода*)
4. {
5. (\*t).sled = new (node); //Резервируем место для нового объекта.

  
Рис.5. Создаем новый объект

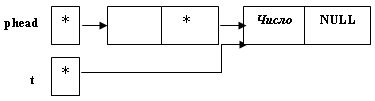
t = (\*t).sled; //Указатель t содержит адрес

//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;

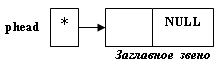
(\*t).elem = el; (\*t).sled = NULL;

cin>>el;

}

}

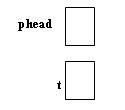
    Ясно, что ***пустой однонаправленный список с заглавным звеном*** можно схематически представить так:

  
Рис.8. Пустой однонаправленный список с заглавным звеном

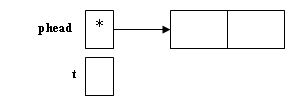
## Построение списка без заглавного звена.

    Приведем алгоритм построения однонаправленного списка без заглавного звена с помощью схем "до и после" [1]. В нежеприведенной схеме переменная **phead** - указатель на первый элемент списка, а **t** - указатель, содержащий адрес последнего (текущего) элемента списка.

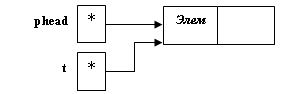
1. Отведем место для указателей в статической памяти:
2. struct node \*phead;
3. struct node \*t;

  
Рис.1. Размещение указателей

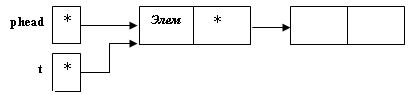
1. В куче резервируем место для динамического объекта:
2. phead = new (node);

  
Рис.2. Размещение объекта в куче

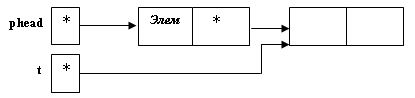
1. Присвоим значение переменной **t**, и поместим в информационное поле значение элемента:
2. t = phead;
3. (\*t).value = *Элем*;

  
Рис.3. Определение **t**

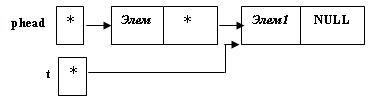
1. Поместим в поле звена адрес нового динамического объекта, зарезервированного в куче:
2. (\*t).next = new (node);

  
Рис.4. Добавление нового элемента

1. Переменная **t** должна содержать адрес последнего добавленного элемента:
2. t = (\*t).next;

  
Рис.5. **t** содержит адрес последнего элемента

1. Если требуется завершить построение списка, то в поле указателя последнего элемента нужно поместить **NULL**:
2. (\*t).next = NULL;
3. (\*t).value = *Элем1*;

  
Рис.6. Результат выполнения операций

    В результате построен линейный однонаправленный список без заглавного звена, содержащий два узла:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris2_7.jpg  
Рис.7. Конечный результат

    Оформим алгоритм в виде программы на языке **C++**.

#include<iostream.h>

struct node

{

int value;

node \*next;

};

void main ()

{

int i;

node \*phead, \*t;

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 << " ";

}

## Общие сведения о списках.

***Список*** - это совокупность объектов, называемых элементами списка, в которой каждый объект содержит информацию о местоположении связанного с ним объекта [1, с. 214].

    Если список располагается в оперативной памяти, то, как правило, информация для поиска следующего объекта - это ***адрес (указатель)*** в памяти. Если список хранится в файле на диске, то информация о следующем элементе может включать смещение элемента от начала файла, положение указателя записи/считывания файла, ключ записи и любую другую информацию, позволяющую однозначно отыскать следующий элемент списка.

    Каждый элемент списка представим структурой языка **C++** с двумя полями:

* ***информационное поле***, которое в общем случае может содержать произвольное количество полей разных типов. Ясно, что если значением переменной **p** является ссылка на элемент списка, то присоединяя к обозначению **(\*p)** с помощью точки имя соответствующего поля, можно манипулировать со значением любого поля информационной части;
* ***ссылка на следующий элемент списка***.

    Каждую пару будем называть ***звеном***, а ссылки, содержащиеся в каждом из звеньев, будем использовать для соединения звеньев в список. Такой способ представления упорядоченной последовательности звеньев называется ***сцеплением***.

    С учетом сказанного, мы можем описать звено списка так:

struct node

{

int elem; //Информационный элемент звена списка

node \*sled; // Указатель на следующее звено списка

};

    Чтобы иметь возможность оперировать со списком как с единым объектом, введем в употребление статическую ссылочную переменную **phead**, которая указывает на первое звено списка и описывается следующим образом:

struct node \*phead;

## Алгоритм удаления списка из памяти.

    Остановимся теперь на удалении однонаправленного линейного списка из динамической 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".

    Соберем рассмотренные функции в объектно-ориентированную программу.

    Пример. Построение однонаправленного списка с заглавным звеном и его просмотр.

#include<iostream.h>

class Spisok

{

private:

struct node

{

int elem;

node \*sled;

} \*phead; //Указатель на начало списка.

public:

Spisok() {phead = new (node); (\*phead).sled=NULL;} //Конструктор.

~Spisok() { delete phead; } //Деструктор.

void POSTROENIE ();

void VYVOD ();

void OCHISTKA ();

};

void main ()

{

Spisok A;

A.POSTROENIE ();

A.VYVOD ();

A.OCHISTKA ();

}

void Spisok::POSTROENIE ()

//Построение однонаправленного списка с заглавным звеном.

// phead - указатель на заглавное звено списка.

{

node \*t;

int el;

t = phead;

cout<< "Вводите элементы списка: ";

cin>>el;

while (el!=0)

{

(\*t).sled = new (node);

t = (\*t).sled; (\*t).elem = el; (\*t).sled = NULL;

cin>>el;

}

}

void Spisok::VYVOD ()

//Вывод содержимого однонаправленного линейного списка

//с заглавным звеном.

// phead - указатель на заглавное звено списка.

{

node \*t;

t = (\*phead).sled;

cout<<"Список: ";

while (t!=NULL)

{

cout<<(\*t).elem<<" ";

t = (\*t).sled;

}

cout<<endl;

}

void Spisok::OCHISTKA ()

//Удаление однонаправленного списка из памяти.

// phead - указатель на заглавное звено списка.

{

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;}

}

# 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.

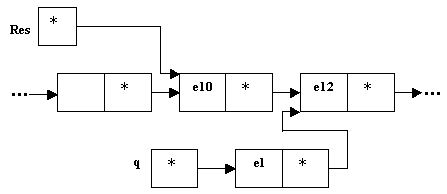
1. В куче резервируется место для динамического объекта:
2. q = new (node);

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;

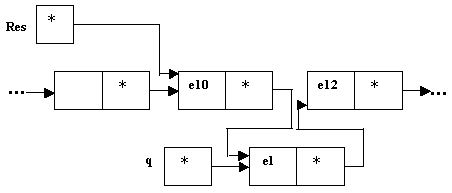
http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris6_2.jpg  
Рис.2. Заполнение информационного поля

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);

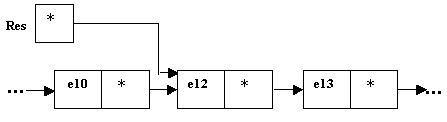
(\*q).elem = el; (\*q).sled = (\*\*Res).sled;

(\*\*Res).sled = q;

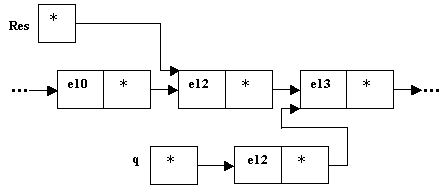
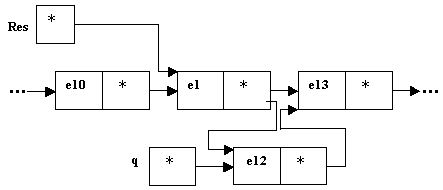
}

## 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;

}

    Пример. Включение звена с заданным элементом в однонаправленный список с заглавным звеном.

#include<iostream.h>

class Spisok

{

private:

struct node

{

int elem;

node \*sled;

};

node \*phead; //Указатель на заглавное звено списка.

node \*Res; //Указатель на найденное звено списка.

public:

Spisok() {phead=new(node); (\*phead).sled=NULL; Res=NULL;}

~Spisok() { delete phead; }

void POSTROENIE();

void VYVOD();

node \*POISK(int);

void VSTAV(int);

void VSTAV1(int);

void OCHISTKA();

};

void main ()

{

Spisok A;

int el,el1;

A.POSTROENIE ();

A.VYVOD ();

cout<<"\nВведите элемент звена, после которого ";

cout<<"осуществляется вставка:\n";

cin>>el;

cout<<"Введите элемент вставляемого звена:\n";

cin>>el1;

if (A.POISK(el)!=NULL)

{

A.VSTAV (el1); A.VYVOD ();

}

else cout<<"Звена с заданным элементом в списке нет!";

cout<<"\nВведите элемент звена, перед которым ";

cout<<"осуществляется вставка:\n";

cin>>el;

cout<<" Введите элемент вставляемого звена:\n";

cin>>el1;

if (A.POISK (el)!=NULL)

{

A.VSTAV1 (el1); A.VYVOD ();

}

else cout<<" Звена с заданным элементом в списке нет!";

A.OCHISTKA();

}

void Spisok::POSTROENIE ()

// Построение однонаправленного списка с заглавным звеном.

// phead - указатель на заглавное звено списка

{

node \*t;

int el;

t = phead;

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;

}

}

void Spisok::VYVOD ()

// Вывод содержимого однонаправленного списка с

// заглавным звеном. phead - указатель на заглавное звено списка.

{

node \*t;

t =(\*phead).sled;

cout<<"Список: ";

while (t!=NULL)

{

cout<<(\*t).elem<<" "; t = (\*t).sled;

}

cout<<endl;

}

node \*Spisok::POISK (int el)

//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).sled;

while (t!=NULL && Res==NULL)

if ((\*t).elem==el)

Res = t;

else t = (\*t).sled;

return Res;

}

void Spisok::VSTAV (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;

}

void Spisok::VSTAV1 (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;

}

void Spisok::OCHISTKA ()

//Удаление однонаправленного списка из памяти.

// phead - указатель на заглавное звено списка.

{

node \*q,\*q1;// Рабочие указатели.

q = phead;

q1 = (\*q).sled; // Указатель q1 "опережает" указатель q.

while (q1!=NULL)

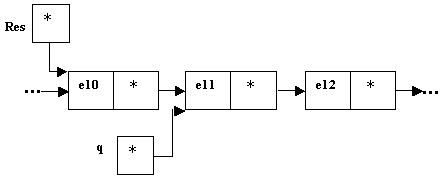
{ q = q1; q1 = (\*q1).sled; delete q;}

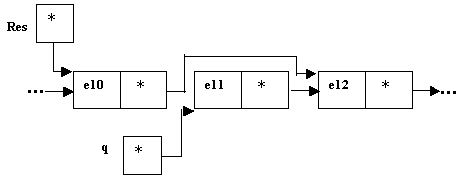
}

## Алгоритм удаления звена(1).

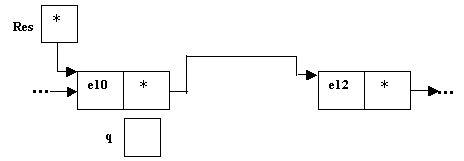
    На этом шаге мы рассмотрим ***алгоритм удаления звена, расположенного после звена, на которое указывает ссылка* Res**.

    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";

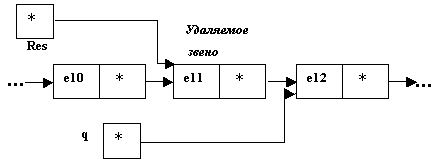
}

## Алгоритм удаления звена(2)

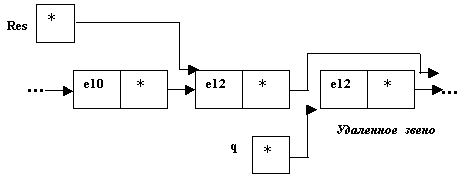
На которое указывает ссылка Res.

Воспользуемся схемами "до и после":

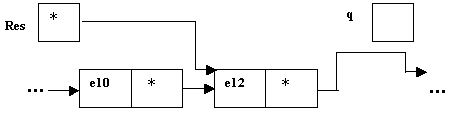
1. Определяем местоположение следующего за удаляемым элемента.
2. q = (\*Res).sled;

  
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;

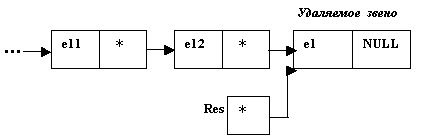
  
Рис.2. "Переписываем" следующий элемент в удаляемый

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

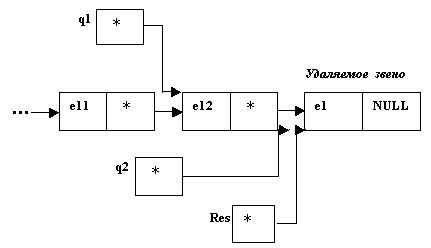
    Найдем указатель на предпоследнее звено линейного списка с помощью двух вспомогательных указателей **q1** и **q2**, перемещающихся по списку "параллельно друг другу", причем указатель **q2** "опережает" указатель **q1** на один шаг.

q1 = phead; q2 = (\*q1).sled; //Инициализация указателей.

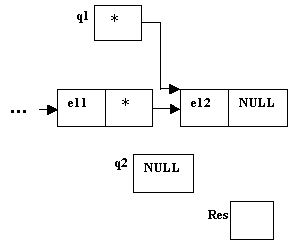
while (q2!=Res)

{ 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;

}

}

    Приведем пример программы, осуществляющей удаление звена из однонаправленного списка с заглавным звеном.

#include<iostream.h>

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 YDALE ();

void YDALE1();

void OCHISTKA();

};

void main ()

{

Spisok A;

int el;

node \*Res\_Zn;

A.POSTROENIE ();

A.VYVOD ();

cout<<"\nВведите элемент звена, после которого ";

cout<<"осуществляется удаление:\n";

cin>>el;

Res\_Zn=A.POISK (el);

if (Res\_Zn!=NULL && (\*Res\_Zn).sled!=NULL)

{A.YDALE (); A.VYVOD ();}

else cout<<"Звена с заданным элементом в списке нет!";

cout<<"\nВведите удаляемый элемент звена:\n";

cin>>el;

Res\_Zn=A.POISK (el);

if (A.POISK (el)!=NULL)

{

A.YDALE1 (); A.VYVOD (); cout<<endl;

}

else cout<<"Звена с заданным элементом в списке нет!";

A. OCHISTKA();

}

void Spisok::POSTROENIE ()

//Построение однонаправленного списка с заглавным звеном.

//phead -указатель на заглавное звено

{

node \*t;

int el;

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;

}

}

void Spisok::VYVOD ()

//Вывод содержимого однонаправленного списка.

//phead - указатель на заглавное звено.

{

node \*t;

t = phead; t = (\*t).sled;

cout<<"Список: ";

while (t!=NULL)

{

cout<<(\*t).elem <<" ";

t = (\*t).sled;

}

}

node \*Spisok::POISK (int el)

//Поиск звена с элементом el в списке, заданном указателем phead.

//В случае успешного поиска в Res находится адрес искомого звена

//списка. В противном случае Res содержит 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;

return Res;

}

void Spisok::YDALE ()

//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";

}

void Spisok::YDALE1 ()

//Удаление звена, на которое указывает ссылка Res.

{

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;

while (q2!=Res)

{

q1 = q2; q2 = (\*q2).sled;

}

(\*q1).sled = NULL; q2 = NULL; delete Res;

}

}

void Spisok::OCHISTKA ()

//Удаление однонаправленного списка из памяти.

// phead - указатель на заглавное звено списка.

{

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;}

}

# 57.Динамические структуры данных C++. Ортогональные списки. Реализация операций над ортогональными списками.

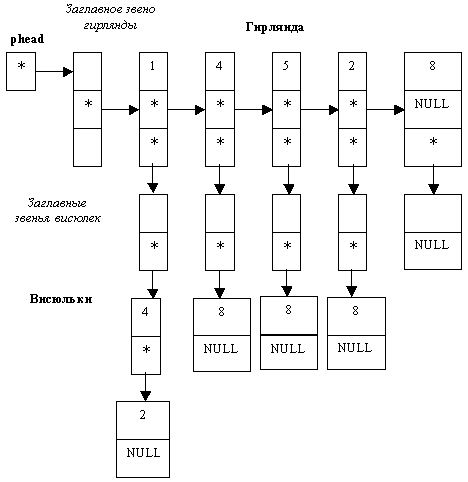
## Ортогональные списки.

    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.

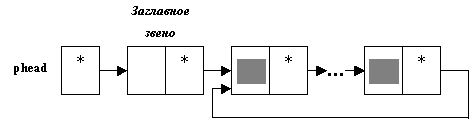
    Один из недостатков линейных списков заключается в том, что, зная указатель **p** на звено списка, мы не имеем доступа к ***предшествующим*** ему звеньям. Если производится просмотр списка, то для повторного обращения к нему исходный указатель на начало списка должен быть сохранен.

    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.

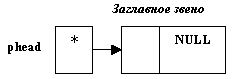
***Кольцевым списком (кольцом)*** на базе линейного однонаправленного списка называется линейный список, в котором указатель из некоторого звена направлен на такое звено в списке, из которого данное звено может быть достигнуто вновь.

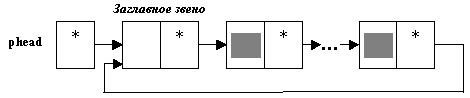
    Опишем два способа представления однонаправленного кольцевого списка с заглавным звеном:

  
Рис.1. Кольцо с удаленным заглавным звеном

    Такой кольцевой список будем называть ***кольцевым списком с удаленным заглавным звеном***.

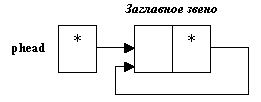
***Пустой кольцевой список с удаленным заглавным звеном*** представим так:

  
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.

    На этом шаге мы приведем лишь обобщающий пример. Приведем текст программы, осуществляющий включение звена в кольцо с удаленным заглавным звеном, а также удаление звена с заданным значением информационного поля из кольца с удаленным заглавным звеном.

#include<iostream.h>

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 ()

{

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();

}

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;

}

else if ((\*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;

}

else if ((\*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 .

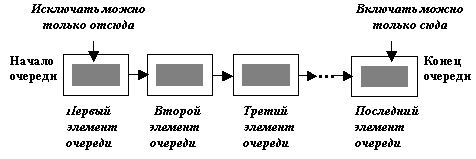
    В практике информационного моделирования часто используются линейные списки, в которых включение, исключение или доступ к звеньям почти всегда производится в первом или последнем звеньях.

    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"***).

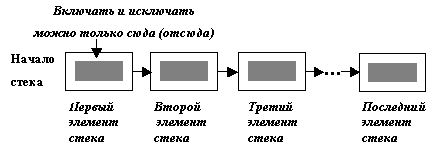
    Мы часто будем использовать термины ***начало*** и ***конец очереди***: информация помещается в конец очереди и удаляется в момент, когда, наконец, достигает ее начала. Изобразим это схематически:

  
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

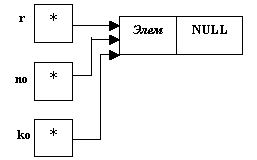
    Отметим, что очередь - динамическая структура данных, так как с течением времени длина очереди (количество входящих в нее звеньев) изменяется. Например, все мы знакомы с очередью людей у кассы в магазине самообслуживания или очередью автомобилей у бензозаправочной станции. Вновь прибывшие становятся в один конец очереди и покидают ее после оплаты покупок или заправки с другого.

    Другой, возможно более уместный, пример очереди может быть обнаружен в вычислительной системе с разделением времени, с которой одновременно работает несколько пользователей. Поскольку такая система обычно имеет единственный центральный процессор и одну основную память, то эти ресурсы должны разделяться среди пользователей путем выделения короткого интервала времени на выполнение программы одного пользователя, за которым следует выполнение программы другого пользователя и так далее до тех пор, пока не будет вновь выполняться программа первого пользователя. Программы пользователей, ожидающие своего выполнения, образуют ***очередь ожидания***. Управление такой очередью необязательно должно основываться на принципе "первым пришел" - "первым вышел", а можно использовать некоторую сложную приоритетную схему, учитывающую такие факторы, как используемый компилятор, требуемое время выполнения, желаемое число строк, выводимых на печать и так далее.

## Алгоритм формирования очереди.

***Очередь на базе линейного однонаправленного списка*** - линейный список, в котором включения элементов производятся на одном конце списка, а все исключения делаются на другом его конце. Обычно ***удаление звеньев из очереди*** происходит из начала линейного списка, а ***помещение звеньев*** осуществляется в конец линейного списка.

    Запишем алгоритм формирования очереди:

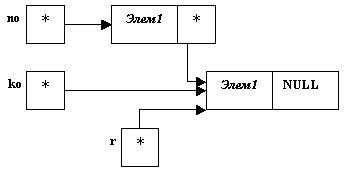
1. r = new (node);
2. (\*r).elem = *Элем*;
3. (\*r).sled = NULL;
4. http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris16_1.jpg  
   Рис.1. Первый элемент в очереди
5. no = r; ko = r;
6.   
   Рис.2. Настройка указателей начала и конца очереди

    Итак, мы образовали очередь, состоящую из одного звена.

1. Продолжим заполнение очереди:
2. r = new (node);
3. (\*r).elem = *Элем1*;
4. (\*r).sled = NULL;

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris16_3.jpg  
Рис.3. Создали новый элемент очереди

1. Настроим указатель на конец очереди:
2. (\*ko).sled = r;
3. 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;

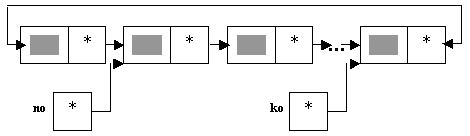
while (r!=NULL)

{ cout<<(\*r).elem<<" "; r = (\*r).sled; }

cout<<endl;

}

*Замечание[1, с.52]. Очереди целесообразно хранить в памяти ЭВМ в виде кольцевого списка с двумя указателями (один - на начало, другой - на конец очереди):*

*  
Рис.5. Представление очереди*

*При исключении из очереди первого звена в первый указатель записывается ссылка на следующее звено списка, которая содержалась в поле указателя исключаемого звена, а при включении в очередь нового звена во второй указатель записывается ссылка на новое звено.*

## Алгоритм добавления звена к очереди.

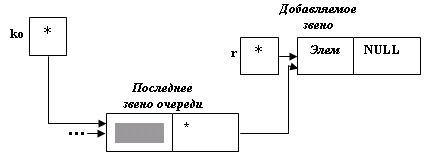
    Приступим к описанию алгоритма добавления звена к очереди. Напомним, что звено добавляется ***в конец очереди***.

  
Рис.1. Исходная очередь

1. Вначале построим добавляемое звено:
2. r = new (node);
3. (\*r).elem = *Элем*;
4. (\*r).sled = NULL;

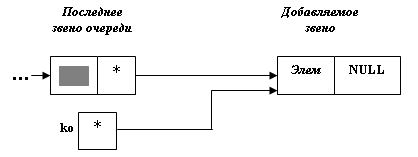
  
Рис.1. Заполнение добавляемого звена

1. Присоединяем звено к очереди:
2. (\*ko).sled = r;

  
Рис.2. Результат присоединения звена

1. "Настраиваем" указатель **ko** на конец очереди:
2. 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;}

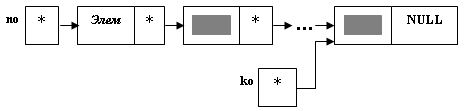
else

{ \*no = r; \*ko = r;}

}

## Алгоритм удаления звена из очереди.

    Пусть очередь не пуста (**no!=NULL**). Изобразим ее схематически:

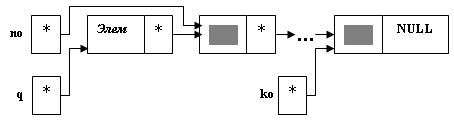
  
Рис.1. Очередь

    Приступим к удалению звена. Напомним, что звено удаляется из очереди из ее начала.

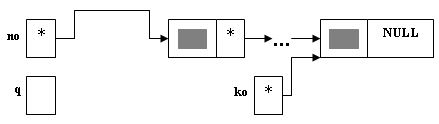
1. Сохраним удаляемый элемент:
2. klad = (\*no).elem;

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris18_2.jpg  
Рис.2. Сохранение удаляемого элемента

1. Сохраним указатель на удаляемый элемент и "перенастроим" указатель на начало очереди:
2. q = no;
3. no = (\*no).sled;

  
Рис.3. "Перенастройка" указателя на начало очереди

1. Теперь необходимо включить в список свободной памяти удаленное из очереди звено с помощью вызова функции:
2. 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;}

}

    Пример. Построение и просмотр очереди, добавление и удаление звеньев из очереди.

#include<iostream.h>

struct node

{

int elem;

node \*sled;

};

class Spisok {

private:

node \*no,\*ko;

int klad;

public:

Spisok () {no=ko=NULL;}

void POSTROENIE ();

void VYVOD ();

void DOBAVLENIE (int);

int Set\_Udal () { return klad; }

void YDALENIE ();

void OCHISTKA();

};

void main ()

{

Spisok A;

int el;

A.POSTROENIE ();

A.VYVOD ();

cout<<"Введите добавляемый элемент: ";

cin>>el;

A.DOBAVLENIE (el); A.VYVOD ();

cout<<"Удалим элемент из очереди.\n";

A.YDALENIE (); A.VYVOD ();

el=A.Set\_Udal();

cout<<"Информационное поле удаленного звена: "<<el<<endl;

A.OCHISTKA();

}

void Spisok::POSTROENIE ()

//Построение очереди на базе однонаправленного списка

//без заглавного звена.

//no - указатель на начало очереди.

//ko - указатель на конец очереди.

{

node \*r;

int el;

cout<<"Вводите элементы очереди:\n";

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;}

}

void Spisok::VYVOD ()

//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;

}

void Spisok::DOBAVLENIE (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;}

}

void Spisok::YDALENIE ()

//Удаление звена из очереди, определенной указателями

//no и ko, с помещением его информационного поля в

//параметр 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;

}

}

void Spisok::OCHISTKA()

//Возврат выделенной памяти в "кучу".

{

node \*q;

q=no;

if (no!=NULL)

{

while (no!=ko)

{

no=(\*q).sled;

delete q; q=no; }

delete no;

no=ko=NULL;

}

}

# 60.Динамические структуры данных C++. Стек. Формирование стека. Включение звена в стек. Удаление звена из стека.

## Стек. Формирование стека.

Напомним, что ***стек*** - это специально организованная память, выборка и занесение данных в которую подчиняется дисциплине **LIFO** (***"последним вошел - первым обслужен"***).

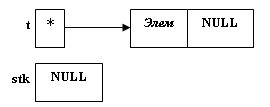
***Стек на базе линейного однонаправленного списка*** - линейный однонаправленный список, в котором все включения и исключения звеньев делаются в одном (выбранном нами) конце списка.

    Опишем алгоритм помещения в стек информации.

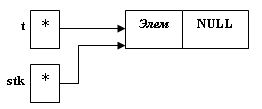
1. Вначале стек пуст:
2. stk = NULL;

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:
2. cin>>*Элем*;
3. t = new (node);
4. (\*t).elem = Элем; (\*t).sled = stk;

  
Рис.2. Новый элемент

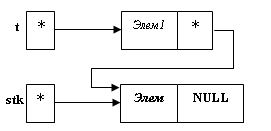
1. "Настраиваем" указатель стека на созданный элемент:
2. stk = t;

  
Рис.3. "Настройка" указателя стека

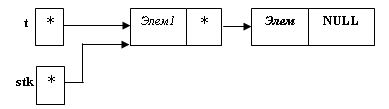
1. В результате в стек будет помещено первое звено:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris19_4.jpg  
Рис.4. Первый элемент в стеке

1. Продолжим заполнение стека:
2. cin>>*Элем1*;
3. t = new (node);
4. (\*t).elem = Элем1;
5. (\*t).sled = stk;

  
Рис.5. Размещение в стеке второго элемента

1. "Настраиваем" указатель стека на созданный элемент:
2. 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.

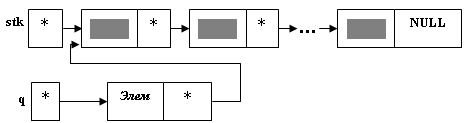
1. Исходное состояние стека:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris20_1.jpg  
Рис.1. Исходное состояние стека

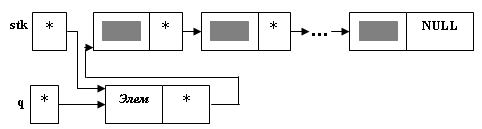
1. Создаем новый элемент:
2. q = new (node);
3. (\*q).elem = *Элем*;

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris20_2.jpg  
Рис.2. Новый элемент

1. Включаем элемент в начало стека:
2. (\*q).sled = stk;

  
Рис.3. Включение элемента в стек

1. "Настроим" указатель вершины стека:
2. 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.

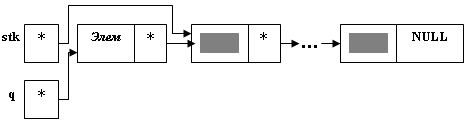
http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris21_1.jpg  
Рис.1. Исходный стек

    Тогда спокойно приступаем к удалению.

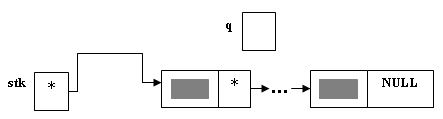
1. Сохраняем удаляемый элемент:
2. klad = (\*stk).elem;

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:
2. q = stk; stk = (\*stk).sled;

  
Рис.3. "Перенастройка" указателя стека

1. Возвращаем память в кучу:
2. 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;}

}

    Приведем пример программы, реализующей действия со стеком. Формирование и вывод содержимого стека на экран дисплея. Удаление и вставка звена в стек.

#include<iostream.h>

struct node

{

int elem;

node \*sled;

};

class Spisok

{

private:

node \*stk;

int klad;

public:

Spisok () { stk=NULL; }

int Set\_Stack () {return klad;}

void POSTROENIE();

void VYVOD();

void W\_S(int);

void YDALENIE ();

void OCHISTKA();

};

void main ()

{

Spisok A;

int el;

int t; //Содержимое информационного поля верхушки стека.

A.POSTROENIE (); A.VYVOD ();

cout<<"Введите вставляемый элемент: ";

cin>>el;

A.W\_S (el); A.VYVOD ();

cout<<"Удалим элемент из стека.\n";

A.YDALENIE ();

t=A.Set\_Stack();

cout<<"Из стека было извлечено число... "<<t<<endl;

A.VYVOD ();

A.OCHISTKA();

}

void Spisok::POSTROENIE ()

//Построение стека, заданного указателем stk.

{

node \*t;

int el;

cout<<"Вводите элементы стека: ";

cin>>el;

while (el!=0)

{

t = new (node);

(\*t).elem = el; (\*t).sled = stk;

stk = t; cin>>el;

}

}

void Spisok::VYVOD ()

//Вывод содержимого стека, заданного указателем stk.

{

node \*t;

cout<<"Содержимое стека: "; t = stk;

while (t!=NULL)

{cout<<(\*t).elem<<" "; t = (\*t).sled;}

cout<<endl;

}

void Spisok::W\_S (int el)

//Помещение элемента el в стек stk.

{

node \*q;

q = new (node);

(\*q).elem = el; (\*q).sled = stk; stk = q;

}

void Spisok::YDALENIE ()

//Удаление элемента из стека, заданного указателем stk.

//Значение информационного поля удаляемого элемента

//помещается в параметр klad.

{

node \*q;

if (stk==NULL)

cout<<"Стек пуст!\n";

else

{

klad = (\*stk).elem; q = stk;

stk = (\*stk).sled; delete q;

}

}

void Spisok::OCHISTKA()

//Возврат выделенной памяти в "кучу".

{

node \*t,\*q;

t = stk;

if (t!=NULL)

{

q=(\*t).sled;

while (q!=NULL)

{

delete t; t = q; q =(\*q).sled;

}

delete t;

}

}

# 61.Динамические структуры данных C++. Дек.

***создание и основные операции над деками***.

***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.h>

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 ()

{

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();

}

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).**

 Рассмотрим ***алгоритм формирования списка***:

1. Приступим к построению алгоритма формирования двунаправленного списка с заглавным звеном. Опишем необходимые переменные:

node \*nsp; **// Указатель на заглавное звено списка.**

node \*ksp; **// Указатель на последнее звено списка.**

node \*rsp; **// Рабочий указатель для перемещения по списку.**

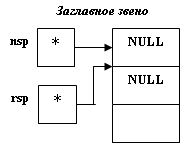
1. Построим заглавное звено:

nsp = **new**(node);

rsp = nsp;

(\*nsp).pred = **NULL**;

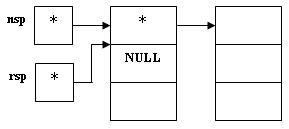
(\*nsp).sled = **NULL**;

  
Рис.1. Заглавное звено

1. Создаем элемент списка:

cin>>el;

(\*rsp).sled = **new**(node);

  
Рис.2. Элемент списка

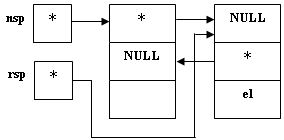
1. Заполняем поля элемента:

(\*((\*rsp).sled)).pred = rsp;

rsp = (\*rsp).sled;

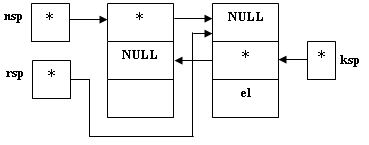
(\*rsp).sled = **NULL**;

(\*rsp).elem = el;

  
Рис.3. Заполнили поля элемента списка

1. "Настраиваем" указатель на последний элемент списка:

ksp = rsp;

  
Рис.4. "Настроили" указатель

1. Построен двунаправленный список с заглавным звеном, содержащий одно звено.
2. Запишем функцию построения двунаправленного списка с заглавным звеном:

**void** Postroenie (node \*\*nsp, node \*\*ksp)

**// Построение двунаправленного списка с заглавным звеном:**

**// \*nsp - указатель на начало списка,**

**// \*ksp - указатель на конец списка.**

{

node \*rsp;

**int** el;

\*nsp = **new**(node);

rsp = \*nsp;

(\*\*nsp).pred = (\*\*nsp).sled = **NULL**;

cout<<"Вводите последовательность:\n"; cin>>el;

**while** (el!=0)

{ (\*rsp).sled = **new**(node); (\*((\*rsp).sled)).pred = rsp;

rsp = (\*rsp).sled; (\*rsp).sled = **NULL**; (\*rsp).elem = el;

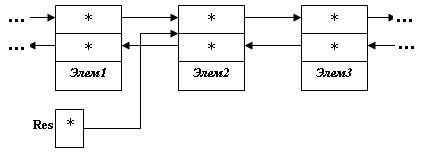
cin>>el; }

\*ksp = rsp;

}

  На этом шаге мы рассмотрим ***алгоритм вставки нового звена после звена, на которое указывает ссылка (1-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"

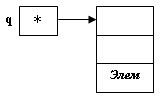
    Проверим, является ли звено, на которое указывает ссылка Res, последним в списке. Это осуществляется путем анализа значения операции отношения (\*Res).sled != NULL.

    Пусть звено, на которое указывает Res, не является последним.

1. В heap-области (в куче) 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);

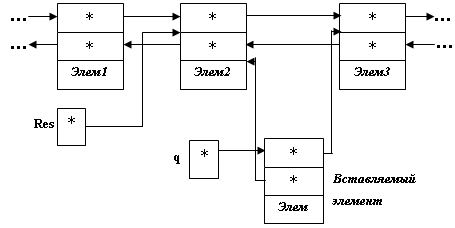
(\*q).elem = *Элем*;

  
Рис.2. Вставляемый элемент

1. "Настраиваем" указатели вставляемого элемента:

(\*q).sled = (\*Res).sled;

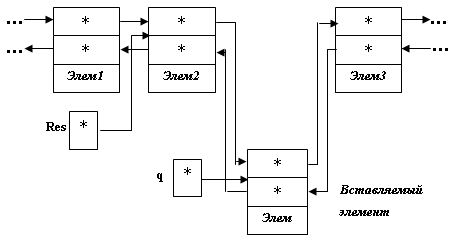
(\*q).pred = (\*\*Res.sled).pred;

  
Рис.3. "Настройка" указателей вставляемого элемента

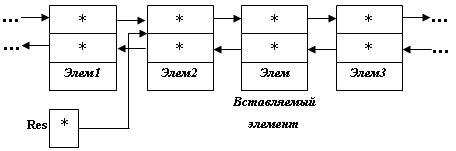
1. "Настраиваем" указатели списка на вставляемый элемент:

(\*\*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

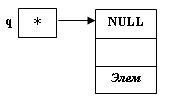
    Если элемент, после которого размещается новый элемент, окажется *последним в двунаправленном списке*, тогда алгоритм изменится следующим образом:

1. Создадим новый элемент и заполним его поля:

q = **new**(node);

(\*q).elem = *Элем*;

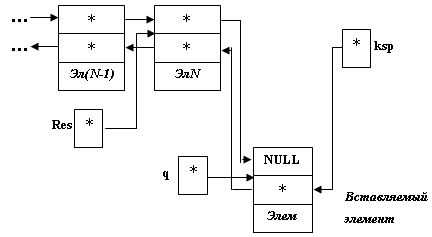
(\*q).sled = **NULL**;

  
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;}

}

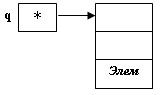
    На этом шаге мы рассмотрим ***другой способ вставки звена в двуноправленный список (2-й случай)***:

    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);

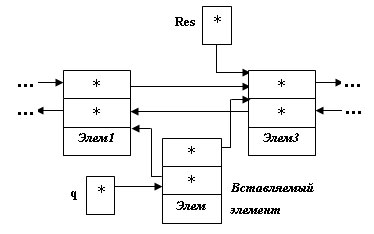
(\*q).elem = *Элем*;

  
Рис.1. Создание элемента

1. "Настраиваем" его указатели:

(\*q).sled = (\*(\*Res).pred).sled;

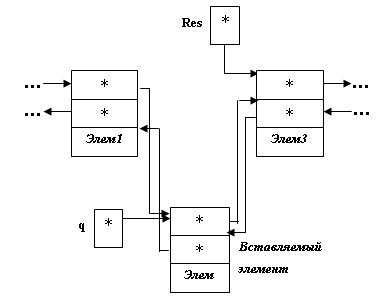
(\*q).pred = (\*Res).pred;

  
Рис.2. "Настройка" указателей созданного элемента

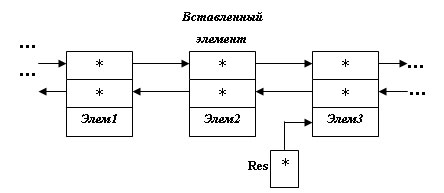
1. "Настраиваем" указатели элементов списка на вставляемый элемент:

(\*(\*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;

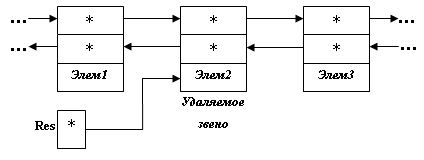
}

На этом шаге мы 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.

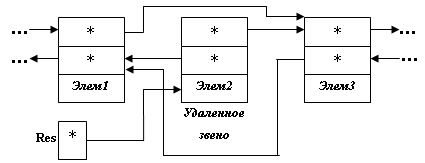
    Вначале рассмотрим ситуацию, когда он *не последний*, то есть (\*Res).sled != NULL. Изобразим схему "до", соответствующую этому случаю:

  
Рис.1. "Начальная" позиция

1. Исключим звено из списка, "перенастроив" указатели предыдущего и последующего звеньев:

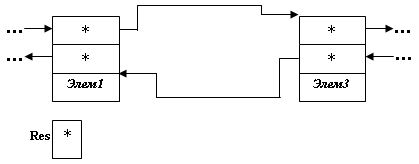
(\*(\*Res).sled).pred = (\*Res).pred;

(\*(\*Res).pred).sled = (\*Res).sled;

  
Рис.2. "Перенастройка" указателей

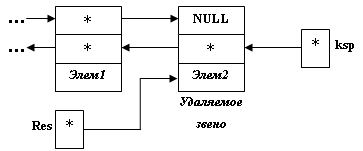
1. Осталось только удалить звено, на которое указывает Res, из heap-области:

**delete** Res;

  
Рис.3. Освобождение памяти

    Пусть теперь удаляемый элемент оказался *последним*.

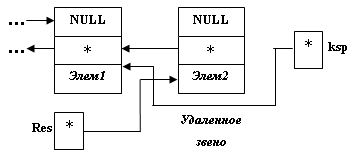
    Взгляните на схему "до":

  
Рис.4. "Начальная" позиция

1. "Настроим" указатели предпоследнего элемента и конца списка:

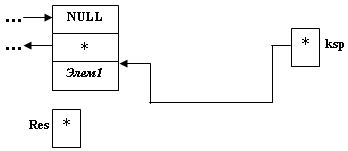
(\*(\*Res).pred).sled = **NULL**;

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;}

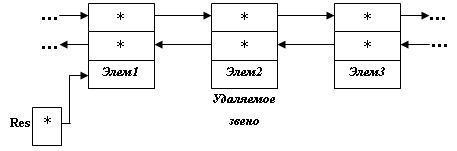
**else**

{ (\*(\*Res).pred).sled = **NULL**; \*ksp = (\*\*ksp).pred; **delete** Res;}

}

 На этом шаге мы рассмотрим другой ***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":

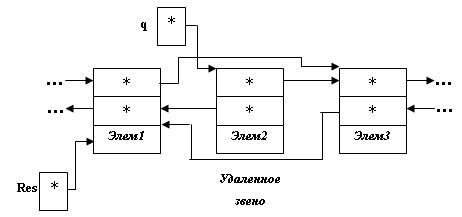
  
Рис.1. "Начальная" позиция

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;

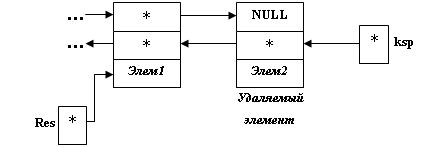
(\*Res).sled = (\*(\*Res).sled).sled;

  
Рис.2. "Перенастройка" указателей

1. Осталось только удалить звено, на которое указывает q, из heap-области (кучи):

**delete** q;

    Если же удаляемый элемент является *последним* в двунаправленном списке,

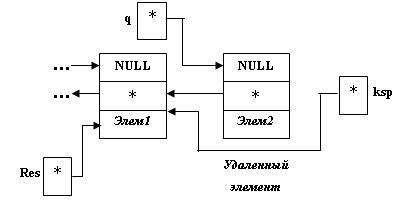
  
Рис.3. "Начальная" позиция

то алгоритм становится тривиальным:

1. "Настроим" указатели предпоследнего элемента и конца списка:

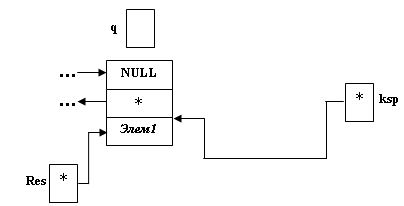
q = (\*Res).sled; (\*Res).sled = **NULL**;

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; }

}

Приведем пример программы, иллюстрирующей применение рассмотренных функций:

    Пример: построение двунаправленного списка. Помещение звена в двунаправленный список. Удаление звена из двунаправленного списка. Очистка двунаправленного списка.

**#include**<iostream.h>

**struct** node

{

**int** elem;

node \*sled;

node \*pred;

};

**class** Spisok

{

**private**:

node \*nsp,\*ksp;

**public**:

Spisok() {nsp=ksp=**NULL**;}

**void** Postroenie ();

**void** VyvodForward ();

**void** VyvodBack ();

**void** Ochistka ();

**void** InsAfter (**int**,node\*);

**void** InsBefore (**int**,node\*);

**void** Delete (node\*);

**void** DelAfter (node\*);

node \*PoiskForward (**int**);

node \*PoiskBack (**int**);

};

**void** main ()

{

Spisok A;

node \*Res;

**int** el,el1;

A.Postroenie ();

A.VyvodForward (); A.VyvodBack ();

cout<<"Введите элемент звена, после которого ";

cout<<"осуществляется вставка: ";

cin>>el;

cout<<"Введите элемент вставляемого звена: ";

cin>>el1;

Res=A.PoiskForward (el);

**if** (Res!=**NULL**)

{

A.InsAfter (el1,Res);

A.VyvodForward (); A.VyvodBack ();

}

**else** cout<<"Звена с заданным элементом в списке нет!\n";

cout<<"Введите элемент звена, перед которым ";

cout<<"осуществляется вставка: ";

cin>>el;

cout<<"Введите элемент вставляемого звена: ";

cin>>el1;

Res = A.PoiskBack (el);

**if** (Res!=**NULL**)

{

A.InsBefore (el1,Res);

A.VyvodForward (); A.VyvodBack ();

}

**else** cout<<"Звена с заданным элементом в списке нет!\n";

cout<<"Введите элемент звена, после которого ";

cout<<"осуществляется удаление: ";

cin>>el;

Res = A.PoiskForward (el);

**if** (Res!=**NULL**)

{

A.DelAfter (Res);

A.VyvodForward (); A.VyvodBack (); }

**else** cout<<"Звена с заданным элементом в списке нет!\n";

cout<<"Введите элемент звена, которое ";

cout<<"надо удалить: ";

cin>>el;

Res = A.PoiskForward (el);

**if** (Res!=**NULL**)

{

A.Delete (Res);

A.VyvodForward (); A.VyvodBack ();

}

**else** cout<<"Звена с заданным элементом в списке нет!\n";

A.Ochistka ();

}

**void** Spisok::Postroenie ()

**//Построение двунаправленного списка с заглавным звеном:**

**// nsp - указатель на начало списка,**

**// ksp - указатель на конец списка.**

{

node \*rsp;

**int** el;

nsp = **new**(node);

rsp = nsp;

(\*nsp).pred = **NULL**; (\*nsp).sled = **NULL**;

cout<<"Вводите последовательность:\n";

cin>>el;

**while** (el!=0)

{

(\*rsp).sled = **new**(node);

(\*((\*rsp).sled)).pred = rsp; rsp = (\*rsp).sled;

(\*rsp).sled = **NULL**; (\*rsp).elem = el;

cin>>el;

}

ksp = rsp;

}

**void** Spisok::VyvodForward ()

**//Вывод содержимого двунаправленного списка от его начала.**

**// nsp - указатель на начало списка, ksp - указатель на конец списка.**

{

node \*rsp;

rsp = (\*nsp).sled;

cout<<"Двунаправленный список содержит: ";

**while** (rsp!=**NULL**)

{

cout<<(\*rsp).elem<<" "; rsp = (\*rsp).sled;

}

cout<<endl;

}

**void** Spisok::VyvodBack ()

**//Вывод содержимого двунаправленного списка от его конца.**

**// nsp - указатель на начало списка, ksp - указатель на конец списка.**

{

node \*rsp;

rsp = ksp;

cout<<"Двунаправленный список в обратном порядке: ";

**while** ((\*rsp).pred!=**NULL**)

{

cout<<(\*rsp).elem<<" "; rsp = (\*rsp).pred;

}

cout<<endl;

}

node \*Spisok::PoiskForward (**int** el)

**//Функция возвращает указатель на найденный элемент el**

**//двунаправленного списка, заданного указателями nsp**

**// и ksp, или NULL, если элемент в списке не найден.**

{

node \*q;

node \*Res;

Res = **NULL**; q = (\*nsp).sled;

**while** (q!=**NULL** && Res==**NULL**)

**if** ((\*q).elem==el) Res = q;

**else** q = (\*q).sled;

**return** Res;

}

node \*Spisok::PoiskBack (**int** el)

**//Функция возвращает указатель на найденный элемент el**

**//двунаправленного списка, заданного указателями nsp**

**// и ksp, или NULL, если элемент в списке не найден.**

{

node \*q;

node \*Res;

Res = **NULL**; q = ksp;

**while** (q!=**NULL** && Res==**NULL**)

**if** ((\*q).elem==el) Res = q;

**else** q = (\*q).pred;

**return** Res;

}

**void** Spisok::InsAfter (**int** el,node \*Res)

**//Вставка звена с информационным полем el в**

**//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;

}

}

**void** Spisok::InsBefore (**int** el,node \*Res)

**//Вставка звена с информационным полем el в**

**//в двунаправленный 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;

}

**void** Spisok::Delete (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;

}

}

**void** Spisok::DelAfter (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<<"Указано последнее звено!\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; }

}

**void** Spisok::Ochistka ()

**//Удаление двунаправленного списка из памяти.**

**// nsp - указатель на заглавное звено списка,**

**// ksp - указатель на последнее звено списка.**

{

node \*q,\*q1;

q = nsp; q1 = (\*q).sled;

**while** (q1!=**NULL**)

{

q = q1; q1 = (\*q1).sled; **delete** q;

}

**delete** nsp; nsp = ksp = **NULL**;

}

# **64. Динамические структуры данных C++. Двунаправленные кольцевые списки.**

Познакомимся с двунаправленными кольцевыми списками:

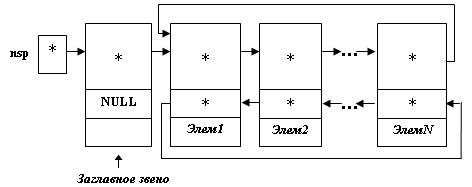
    В программировании двунаправленные списки часто преобразовывают следующим образом: "обычный" линейный двунаправленный список замыкают в своеобразное "кольцо": при движении по списку ***в прямом направлении***можно от последнего звена переходить к звену, следующему прямо за заглавным звеном, а при движении ***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.

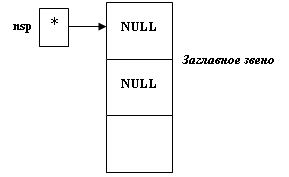
    Мы приведем два варианта структуры кольцевых двунаправленных списков.

    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!";

}

    Пример. Построение двунаправленного кольца с удаленным заглавным звеном, вывод содержимого кольцевого списка, вставка звена, удаление звена.

**#include**<iostream.h>

**struct** node

{

**int** elem;

node \*sled;

node \*pred;

};

**class** Spisok

{

**private**:

node \*nsp;

**public**:

Spisok() {nsp=**NULL**;}

**void** BuiltRing ();

**void** VyvodLeftRight ();

**void** VyvodRightLeft ();

**void** InsAfter (node\*,**int**);

**void** InsBefore (node\*,**int**);

**void** Delete (node\*);

**void** DelAfter (node\*);

node \*SearchRing (**int**);

**void** Ochistka();

};

**void** main ()

{

Spisok A;

node \*Res;

**int** el,el1;

A.BuiltRing ();

cout<<"Содержимое кольца 'по часовой стрелке': \n";

A.VyvodLeftRight ();

cout<<"Содержимое кольца'против часовой стрелки': \n";

A.VyvodRightLeft ();

cout<<"Введите элемент звена, после которого ";

cout<<"осуществляется вставка: ";

cin>>el;

cout<<"Введите элемент вставляемого звена: ";

cin>>el1;

Res = A.SearchRing (el);

**if** (Res!=**NULL**)

{A.InsAfter (Res,el1); A.VyvodLeftRight ();}

**else** cout<<"Звена с таким элементом в списке нет!\n";

cout<<"Введите элемент звена, перед которым ";

cout<<"осуществляется вставка: ";

cin>>el;

cout<<"Введите элемент вставляемого звена: ";

cin>>el1;

Res = A.SearchRing (el);

**if** (Res!=**NULL**)

{ A.InsBefore (Res,el1); A.VyvodLeftRight (); }

**else** cout<<"Звена с таким элементом в списке нет!\n";

cout<<"Введите элемент звена, который ";

cout<<"надо удалить: ";

cin>>el;

Res = A.SearchRing (el);

**if** (Res!=**NULL**)

{ A.Delete (Res); A.VyvodLeftRight (); }

**else** cout<<"Звена с таким элементом в списке нет!\n";

cout<<"Введите элемент звена, после которого ";

cout<<"осуществляется удаление: ";

cin>>el;

Res = A.SearchRing (el);

**if** (Res!=**NULL**)

{ A.DelAfter (Res); A.VyvodLeftRight (); }

**else** cout<<"Звена с таким элементом в списке нет!\n";

A.Ochistka();

}

**void** Spisok::BuiltRing ()

**// Построение двунаправленного кольцевого списка nsp**

**// с удаленным заглавным звеном.**

**// nsp - указатель на заглавное звено списка.**

{

node \*r;

**int** el;

**//Построим заглавное звено кольцевого списка.**

nsp = **new**(node);

r = nsp; (\*nsp).pred = **NULL**; (\*nsp).sled = **NULL**;

cout<<"Вводите элементы 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;

}

**//А теперь - образуем кольцевой список!**

**if** ((\*nsp).sled!=**NULL**)

{ (\*((\*nsp).sled)).pred = r; (\*r).sled = (\*nsp).sled; }

**else**

cout<<"Кольцевой список пуст!\n";

}

**void** Spisok::VyvodLeftRight ()

**// 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!";

}

**void** Spisok::VyvodRightLeft ()

**// Вывод содержимого двунаправленного кольцевого списка**

**// с удаленным заглавным звеном "против часовой стрелки".**

**// nsp - указатель на заглавное звено списка.**

{

node \*r;

cout<<"Кольцевой список: ";

**if** ((\*nsp).sled!=**NULL**)

{

cout<<(\*((\*((\*nsp).sled)).pred)).elem<<" ";

r = (\*((\*((\*nsp).sled)).pred)).pred;

**while** (r!=(\*((\*nsp).sled)).pred)

{ cout<<(\*r).elem<<" "; r = (\*r).pred; }

cout<<endl;

}

**else** cout<<"пуст!";

}

node \*Spisok::SearchRing (**int** el)

**// Поиск элемента el в кольцевом двунаправленном списке**

**// с удаленным заглавным звеном.**

**// nsp - указатель на заглавное звено списка.**

{

node \*q;

node \*p;

node \*Res;

Res = **NULL**; p = nsp;

**if** ((\*((\*p).sled)).elem==el) Res = (\*p).sled;

**else**

{

q = (\*((\*p).sled)).sled;

**while** (q!=(\*p).sled && Res==**NULL**)

**if** ((\*q).elem==el) Res = q;

**else** q = (\*q).sled;

}

**return** Res;

}

**void** Spisok::InsAfter (node \*Res,**int** el)

**// Вставление в кольцевой двунаправленный список звена**

**// с информационным полем el после звена, на которое**

**// указывает ссылка Res.**

{

node \*q;

q = **new**(node);

(\*q).elem = el; (\*q).sled = (\*Res).sled;

(\*q).pred = (\*(\*Res).sled).pred;

(\*(\*Res).sled).pred = q; (\*Res).sled = q;

}

**void** Spisok::InsBefore (node \*Res,**int** el)

**// Вставка в кольцевой двунаправленный список звена**

**// с информационным полем el перед звеном, на которое**

**// указывает ссылка Res.**

**// nsp - указатель на заглавное звено списка.**

{

node \*q;

q = **new**(node);

(\*q).elem = el;

(\*q).sled = (\*(\*Res).pred).sled; (\*q).pred = (\*Res).pred;

(\*(\*Res).pred).sled = q; (\*Res).pred = q;

**if** (Res==(\*nsp).sled) (\*nsp).sled = q;

}

**void** Spisok::Delete (node \*Res)

**// Удаление из кольцевого двунаправленного списка**

**// звена, на которое указывает ссылка Res.**

**// nsp - указатель на заглавное звено списка.**

{

**if** ((\*Res).sled==Res)

{ (\*nsp).sled = **NULL**; **delete** Res; }

**else**

{

(\*(\*Res).sled).pred = (\*Res).pred;

(\*(\*Res).pred).sled = (\*Res).sled;

**if** ((\*nsp).sled==Res)

**// Удаляем "первое" звено кольца.**

(\*nsp).sled = (\*Res).sled;

**delete** Res;

}

}

**void** Spisok::DelAfter (node \*Res)

**// Удаление из кольцевого двунаправленного списка звена,**

**// расположенного после звена, на которое указывает**

**// ссылка Res.**

**// nsp - указатель на заглавное звено списка.**

{

node \*q;

**if** ((\*Res).sled==Res)

{ (\*nsp).sled = **NULL**; **delete** Res;}

**else**

{

q = (\*Res).sled;

(\*(\*(\*Res).sled).sled).pred = (\*(\*Res).sled).pred;

(\*Res).sled = (\*(\*Res).sled).sled;

**if** ((\*(\*nsp).sled).pred==Res)

**// Удаляем "последнее" звено кольца.**

(\*nsp).sled = (\*Res).sled;

**delete** q;

}

}

**void** Spisok::Ochistka()

{

node \*q,\*q1;

q = (\*((\*nsp).sled)).sled;

q1 = (\*q).sled;

**while** (q1!=(\*((\*nsp).sled)).sled)

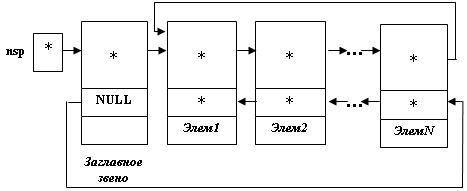
{ **delete** q; q=q1; q1=(\*q1).sled; }

**delete** q;

**delete** nsp;

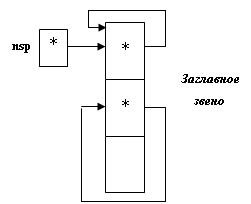
}

    Следует заметить, что предложенный выше способ образования двунаправленного кольцевого списка не является единственно возможным. Имеется еще один вариант представления кольцевых двунаправленных списков, который схематически можно представить так:

  
Рис.3. Кольцевой двунаправленный список с включенным заглавным звеном

    Мы будем называть подобную структуру ***кольцевым двунаправленным списком с включенным заглавным звеном.***

    Опишем структуру ***пустого кольцевого списка с включенным заглавным звеном:***

  
Рис.4. Пустой кольцевой двунаправленный список с включенным заглавным звеном

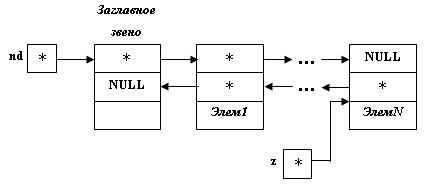
    Каждый из способов образования кольцевого списка имеет как положительные, так и отрицательные стороны. Например, во втором варианте образования списка очень просто реализуется вставка нового звена как в начало списка (после заглавного звена), так и в его конец, ибо вставка нового звена в конец списка эквивалентна его вставке перед заглавным звеном. Однако здесь при циклической обработке элементов списка придется каждый раз проверять, не является ли очередное звено заглавным звеном списка.

    Этого недостатка лишен первый способ организации списка, но в этом случае труднее реализуется добавление звена в конец списка.

# ***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.**

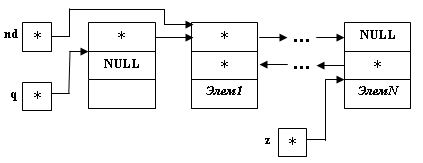
1. Вначале опишем алгоритм формирования двунаправленного списка без заглавного звена из двунаправленного списка с заглавным звеном, изображенного ниже:

"Начальная" позиция:

1. Исключим заглавное звено из списка:

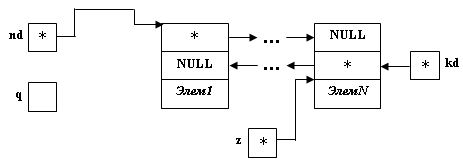
q = nd;

nd = (\*nd).sled;

 (\*nd).pred = NULL;

1. Возврат памяти в кучу и "настройка" указателя на конец дека:

delete q;

kd = z;

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**;}

}

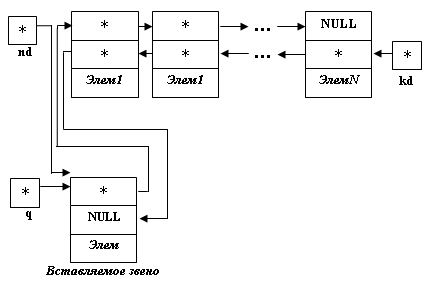
**Алгоритм добавления звена в начало дека**.

    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;

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;}

}

**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:

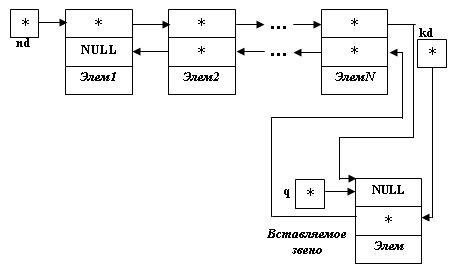
q = new(node);

(\*q).elem = *Элем*;

(\*q).sled = NULL;

(\*q).pred = \*kd;

(\*kd).sled = q;

 \*kd = q;

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...**

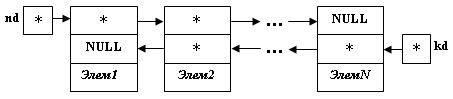
\*nd = q; (\*q).sled = (\*q).pred = **NULL**; \*kd = q;}

**else**

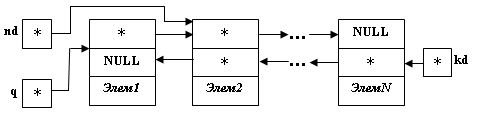
{ (\*q).sled = **NULL**; (\*q).pred = \*kd; (\*\*kd).sled = q; \*kd = q;}

}

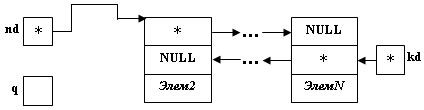
**Алгоритм удаления звена слева**.

1. Пусть уже построен дек на базе двунаправленного списка без заглавного звена:
2. Сохраним адрес удаляемого элемента и "настроим" указатель начала дека:

q = nd;

 nd = (\*nd).sled;

1. Возвратим память в кучу:

 delete q;

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;}

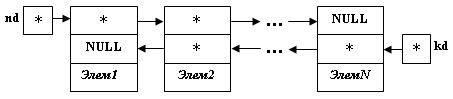
**else**

{ **//В деке находится один элемент.**

q = \*nd; \*el =(\*q).elem; \*nd = \*kd = **NULL**;

**delete** q; cout<<"Дек пуст!\n";}

}

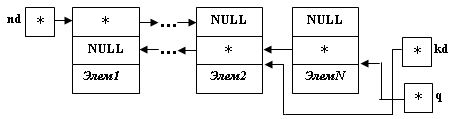
**Удаление звена справа.** Пусть уже построен дек на базе двунаправленного списка без заглавного звена:

 Приступим к пошаговому выполнению алгоритма:

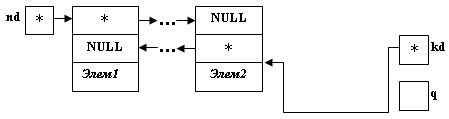
1. Сохраним адрес удаляемого элемента и "настроим" указатель конца дека:

q = kd;

kd = (\*kd).pred;

 (\*kd).sled = NULL;

1. Возвратим память в кучу:

 delete q;

 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";}

}

**Приведем обобщающий пример. Построение дека, вставка и удаление элементов из дека.**

**#include**<iostream.h>

**struct** node

{

**int** elem;

node \*sled;

node \*pred;

};

**class** Spisok

{

**private**:

node \*nd;**//Указатель на начало дека.**

node \*kd;**//Указатель на конец дека.**

**int** klad;**//Информационное поле удаленного элемента.**

**public**:

**void** BuiltDeck ();

**void** VyvodDeck ();

**void** InsLeft (**int**);

**void** InsRight (**int**);

**void** DelLeft ();

**void** DelRight ();

**int** Get\_Klad () {**return** klad;}

**void** Ochistka();

};

**void** main ()

{

Spisok A;

**int** el;

A.BuiltDeck ();

A.VyvodDeck ();

cout<<"Введите элемент звена, вставляемого справа: ";

cin>>el; A.InsRight (el); A.VyvodDeck ();

cout<<"Введите элемент звена, вставляемого слева: ";

cin>>el; A.InsLeft (el); A.VyvodDeck ();

cout<<"Удалим звено справа: \n";

A.DelRight (); A.VyvodDeck ();

cout<<"Был удален элемент: "<<A.Get\_Klad()<<endl;

cout<<"Удалим звено слева: \n";

A.DelLeft (); A.VyvodDeck ();

cout<<"Был удален элемент: "<<A.Get\_Klad()<<endl;

A.Ochistka();

}

**void** Spisok::BuiltDeck ()

**// Построение дека на базе двунаправленного**

**// списка с заглавным звеном.**

**// nd - указатель на начало дека,**

**// \*kd - указатель на конец дека.**

{

node \*q;

node \*z;

**int** el;

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**; }

}

**void** Spisok::VyvodDeck ()

**// Вывод содержимого дека.**

**// nd - указатель на начало дека.**

{

node \*z;

z = nd; cout<<"Содержимое дека: ";

**if** (z!=**NULL**)

**while** (z!=**NULL**)

{ cout<<(\*z).elem<<" "; z = (\*z).sled; }

**else** cout<<"он пуст!\n";

cout<<endl;

}

**void** Spisok::InsLeft (**int** el)

**// Добавление 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**)

{ nd = q; (\*q).sled = (\*q).pred = **NULL**; kd = q; }

**else**

{ (\*q).sled = nd; (\*q).pred = **NULL**;

(\*nd).pred = q; nd = q; }

}

**void** Spisok::InsRight (**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**)

{ nd = q; (\*q).sled = (\*q).pred = **NULL**; kd = q; }

**else**

{ (\*q).sled = **NULL**; (\*q).pred = kd;

(\*kd).sled = q; kd = q; }

}

**void** Spisok::DelLeft ()

**// Удаление звена из дека слева с помещением**

**// элемента удаляемого звена в переменную klad.**

**// nd - указатель на начало дека,**

**// kd - указатель на конец дека.**

{

node \*q;

**if** ((\*nd).sled!=**NULL**)

{ q = nd; klad =(\*q).elem;

nd = (\*nd).sled; (\*nd).pred = **NULL**; **delete** q;}

**else**

{ **// В деке находится один элемент.**

q = nd; klad =(\*q).elem;

nd = kd = **NULL**; **delete** q;cout<<"Дек пуст!\n"; }

}

**void** Spisok::DelRight ()

**// Удаление звена из дека справа с помещением**

**// элемента удаляемого звена в переменную klad.**

**// nd - указатель на начало дека,**

**// kd - указатель на конец дека.**

{

node \*q;

**if** ((\*kd).pred!=**NULL**)

{ q = kd; klad =(\*q).elem;

kd = (\*kd).pred; (\*kd).sled = **NULL**; **delete** q; }

**else**

{**// В деке находится один элемент.**

q = kd; klad =(\*q).elem;

nd = kd = **NULL**; **delete** q; cout<<"Дек пуст!\n"; }

}

**void** Spisok::Ochistka()

{

node \*q,\*q1;

q = nd;

q1 = (\*q).sled;

**while** (q1!=**NULL**)

{ **delete** q; q = q1; q1 = (\*q).sled;}

**delete** q;

nd = kd = **NULL**;

}

# ***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 (узлы 1, 4, 7, 13).
* *Внутренний узел* — любой *узел* дерева, имеющий *потомков*, и таким образом, не являющийся *листовым узлом* (3, 6, 10, 14).
* 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).

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

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

***Структура вершины бинарного дерева поиска***.

    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.

};

***Алгоритмы построения бинарного дерева поиска***.

**void** BuildTree (node \*\*Tree)

**// Построение бинарного дерева.**

**// \*Tree - указатель на корень дерева.**

{

**int** el;

\*Tree = **NULL**; **// Построено пустое бинарное дерево.**

cout<<"Вводите ключи вершин дерева...\n";

cin>>el;

**while** (el!=0)

{ Search (el,Tree); cin>>el;}

}

    В функции **BuildTree()** используется функция поиска вершины с данным ключом **x**:

**void** 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 с ключом x в дереве нет; включить ее.**

\*p = **new**(node);(\*\*p).Key = x; (\*\*p).Count = 1;

(\*\*p).Left = (\*\*p).Right = **NULL**;

}

**else** **//Поиск места включения вершины.**

**if** (x<(\*\*p).Key) **//Включение в левое поддерево.**

Search (x,&((\*\*p).Left));

**else** **if** (x>(\*\*p).Key) **//Включение в правое поддерево.**

Search (x,&((\*\*p).Right));

**else** (\*\*p).Count = (\*\*p).Count + 1;

}

**Дерево отрезков** — структура данных, позволяющая быстро изменять значения в массиве и находить некоторые функции от элементов a[i],a[i+1], … ,a[j] массива.

Построение дерева отрезков:

void build(int v, int l, int r)

{

if (l == r){

d[v] = a[l];

}

else {

int mid = (l + r) / 2;

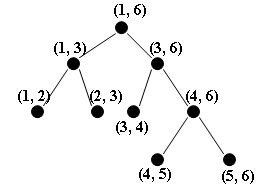
build(v \* 2, l, mid);

build(v \* 2 + 1, mid + 1, r);

d[v] = d[v \* 2] + d[v \* 2 + 1];

}

}

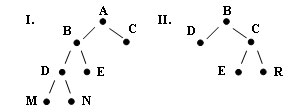
Пример дерева отрезков.

# ***67. Динамические структуры данных C++. Дерево. Обход бинарного дерева. Левосторонний обход бинарного дерева поиска. Концевой обход бинарного дерева поиска. Обратный обход бинарного дерева поиска.***

Для того, чтобы просмотреть информационные поля всех вершин построенного дерева, необходимо совершить его ***обход (посетить каждую его вершину)***.

**Левосторонний обход дерева:**

* ***посетите корень дерева;***
* ***обойдите левое поддерево;***
* ***обойдите правое поддерево.***



A B D M N E C

B D C E R

**void** ObhodLeft (node \*\*w)

**// Левосторонний обход дерева.**

**// \*w - указатель на корень дерева.**

{

**if** (\*w!=**NULL**)

{ cout<<(\*\*w).Key<<" ";

ObhodLeft (&((\*\*w).Left));

ObhodLeft (&((\*\*w).Right)); }

}

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

D E R C B

**void** ObhodEnd (node \*\*w)

**// Концевой обход дерева.**

**// \*w - указатель на корень дерева.**

{

**if** (\*w!=**NULL**)

{ ObhodEnd (&((\*\*w).Left));

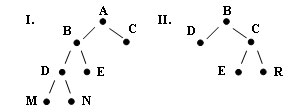
ObhodEnd (&((\*\*w).Right));

cout<<(\*\*w).Key<<" ";}

}

И, наконец, приведем ***алгорим обратного обхода дерева***, который заключается в следующем:

* ***обойдите левое поддерево***;
* ***посетите корень дерева***;
* ***обойдите правое поддерево***.

 M D N B E A C

D B E C R

**void** ObhodBack (node \*\*w)

**// Обратный обход бинарного дерева.**

**// \*w - указатель на корень дерева.**

{

**if** (\*w!=**NULL**)

{ ObhodBack (&((\*\*w).Left));

cout<<(\*\*w).Key<<" ";

ObhodBack (&((\*\*w).Right)); }

}

# **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

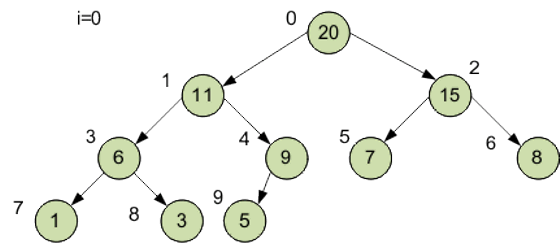
**Двоичная куча** представляет собой полное бинарное дерево, для которого выполняется основное свойство кучи: **приоритет каждой вершины больше приоритетов её потомков**.

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 двоичной кучи**

#include *<vector>*

#include *<exception>*

**using** **namespace** std;

*// здесь и далее:*

*// А - вектор элементов кучи*

*// length - количество элементов кучи*

*// (то есть куча содержит первые length элементов массива A)*

*// i - индекс, с которым мы работаем*

void Heapify(vector<int>& A, size\_t length, size\_t i) {

size\_t l = 2 \* i + 1;

size\_t r = 2 \* i + 2;

size\_t largest = i;

**if** (l < length && A[l] > A[largest])

largest = l;

**if** (r < length && A[r] > A[largest])

largest = r;

**if** (largest != i) {

swap(A[i], A[largest]);

Heapify(A, length, largest);

}

}

void Build\_Heap(vector<int>& A, size\_t length) {

**for** (size\_t i = length / 2 - 1; i != (size\_t)(-1); i--)

Heapify(A, length, i);

}

void Heapsort(vector<int>& A) {

Build\_Heap(A, A.size());

**for** (size\_t i = A.size() - 1; i > 0; i--) {

swap(A[0], A[i]);

Heapify(A, i, 0);

}

}

void Heap\_Increase\_Key(vector<int>& A, size\_t length, size\_t i, int key) {

**if** (key < A[i])

**throw** exception("Новый ключ меньше предыдущего");

A[i] = key;

**while** (i > 0 && A[(i-1)/2] < A[i]) {

swap(A[(i-1)/2], A[i]);

i = (i-1)/2;

}

}

void Heap\_Insert(vector<int>& A, size\_t length, int key) {

length++;

**if** (length > A.size())

A.resize(length);

Heap\_Increase\_Key(A, length, length - 1, key);

}

int Heap\_Extract(vector<int>& A, size\_t length) {

**if** (length == 0)

**throw** exception("Куча пуста");

int max\_element = A[0];

A[0] = A[length - 1];

length--;

Heapify(A, length, 0);

**return** max\_element;

}

**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];

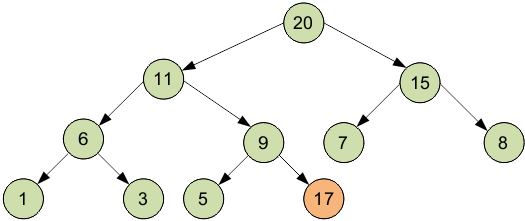
  HeapSize = 0;  
}

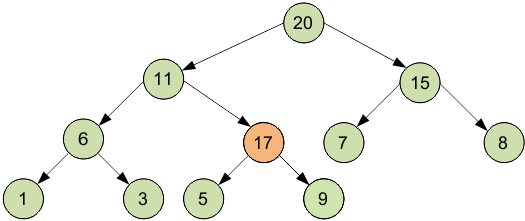
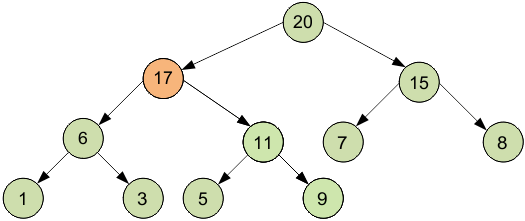
# **73 Структуры данных: двоичная куча (binary heap). Добавление элемента кучи. Вывод элементов кучи. Упорядочение кучи. Удаление вершины кучи (максимального элемента).**

**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.**

С точки зрения **IT** рекурсия используется для **1)алгоритмов, 2)функций**.

**Рекурсия** – определение части функции (метода) через саму себя, то есть это функция, которая вызывает саму себя, непосредственно (в своём теле) или косвенно (через другую функцию).

Войдя в рекурсивную функцию (программу), процесс движется по тексту программы до тех пор, пока не встретит ее вызов, после чего опять начнет выполняться та же самая функция сначала. При этом **копируются формальные, фактические параметры, локальные переменные и точка возврата**. Каждый новый рекурсивный вызов порождает новый "**экземпляр**" формальных параметров и локальных переменных, причем старый "экземпляр" не уничтожается, а сохраняется. Программа рекурсивной функции не меняется, она присутствует в памяти компьютера в единственном экземпляре. Рекурсия **должна иметь** внутри **себя условие завершения**.

**Достоинство рекурсии**: более простое создание версии некоторых алгоритмов по сравнению с итеративными эквивалентами. Недостатки: требуется **большой объем памяти** для многократного размещения в памяти формальных параметров и локальных переменных рекурсивной функции; **большой расход времени** на многократное выполнение команд вызова функции

**Виды рекурсии:**

**Прямая рекурсия (простая)**

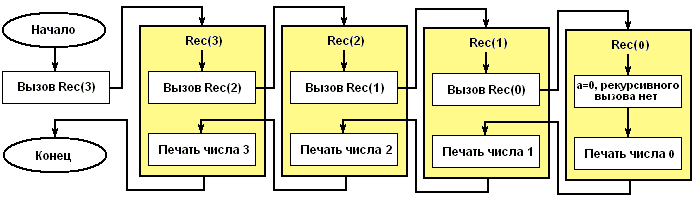
В случае прямой рекурсии вызов функцией самой себя делается непосредственно в этой же функции.

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**

Функция содержит **единственный** вызов самой себя. В таком случае рекурсия становится эквивалентной циклу. Любой циклический алгоритм можно преобразовать в линейно-рекурсивный и наоборот. Пример. **n! = 1∙2∙3… n** , т.е. **n! = n∙(n - 1)! (факториал)**

*Циклический* алгоритм вычисления факториала:

**int fact(int n)**

**{ for (int s = 1; n != 0; n--)**

**s \*= n;**

**return s;**

**}**

*Рекурсивный* алгоритм вычисления факториала:

**int fact(int n)**

**{ if (n == 1) return 1;**

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

**}**

**Ветвящаяся рекурсия**

Функция вызывается более одного раза внутри рекурсивной функции. Рассмотрим на примере вычисления числа Фибоначчи. Последовательность Фибоначчи обладает свойством: каждый следующий член является суммой двух предыдущих.

int Fib (int n)

{ if (n == 0) return 0;

else if (n == 1) return 1;

else return Fib(n - 1) + Fib(n - 2); }

void main()

{ int i;

for(i = 0; i < 10; i++)

cout<<Fib(i)<<' ';

return 0; }

**Бесконечная рекурсия**

Бесконечная рекурсия всегда приводит к одному и тому же результату: аварийному завершению программы из-за переполнения стека, поскольку в неё отсутствует проверка какого-либо условия, в зависимости от которого осуществлялся бы выход из рекурсивной функции.

int main()

{cout << "Hello world!" << endl;

   main();

   return 0;}

# **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.

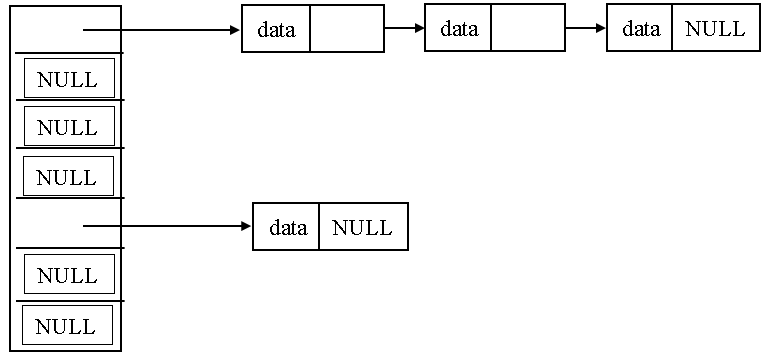
Такой подход не избавляет нас от рекурсии полностью. Однако он позволяет ограничиться только одним обращением к рекурсивной функции, что может быть достаточно, если мотивом является забота о максимальной производительности.

**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**

**Компилятор**, который использует хэш-таблицу для управления информацией о переменных в программе; **Web-бразуер** использует хэш-таблицу для хранения адресов страниц, которые недавно посещали для оперативного хранения( cache – кэширования) недавно использованных доменных имён и их IP – адресов. Хэш-таблицы часто применяются в базах данных. Один из наиболее эффективных способов реализации словаря - хэш-таблица.

Таким образом, хэширован полезно, когда широкий диапазон возможных значений должен быть сохранён в малом объёме памяти, и нужен способ быстрого, практически произвольного доступа.

В программировании хэш-таблица – динамическое множество, поддерживающее словарные операции: добавление, поиск и удаление элемента, использующее специальные методы адресации. Основное отличие таблицы от других динамических множеств – вычисление адреса элемента по значению ключа.

Алгоритмы поиска, которые используют хэширование, состоит из двух отдельных частей. Первый шаг – вычисление хэш-функции, которая преобразует ключ поиска в адрес в таблице. В идеале различные ключи должны были бы отображаться на различные адреса, но часто два и более различных ключа могут преобразовываться в один и тот же адрес в таблице. Поэтому вторая часть поиска методом хэширования – процесс разрешения конфликтов, которые обрабатывает такие ключи.

**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).**

***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. Затем записи вносятся в таблицу – каждая на свое место, определяемое ее ключом. При поиске ключ используется как адрес и по этому адресу выбирается запись. Если выбранная запись пустая, то записи с таким ключом вообще нет в таблице. Таблицы прямого доступа очень эффективны в использовании, но, к сожалению, область их применения весьма ограничена. В целях экономии памяти можно назначать размер пространства записей равным размеру фактического множества записей или превосходящим его незначительно. В этом случае необходимо иметь некоторую функцию, обеспечивающую отображение точки из пространства ключей в точку в пространстве записей, то есть, преобразование ключа в адрес записи: http://ok-t.ru/studopedia/baza5/117194492528.files/image041.png, где *a* – адрес, *k* – ключ.

Идеальной хеш-функцией является инъективная функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса.

***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 и сама по себе может быть применена лишь в очень ограниченном диапазоне реальных задач. Однако операция деления по модулю обычно применяется как последний шаг в более сложных функциях хеширования, обеспечивая приведение результата к размеру пространства записей. Если ключей меньше, чем элементов массива, то в качестве хеш-функции можно использовать деление по модулю, то есть остаток от деления целочисленного ключа Key на размерность массива **HashTableSize**, то есть: **Key % HashTableSize**

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.

Если сегменты примерно одинаковы по размеру, то в этом случае списки всех сегментов должны быть наиболее короткими при данном числе сегментов. Если исходное множество состоит из ***N*** элементов, тогда средняя длина списков будет http://ok-t.ru/studopedia/baza5/117194492528.files/image057.pngэлементов. Если можно оценить величину ***N*** и выбрать *В* как можно ближе к этой величине, то в каждом списке будет один или два элемента. Тогда время выполнения операторов словарей будет малой постоянной величиной, не зависящей от ***N***.

***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. Но если в хеш-таблице допускается удаление элементов, то при достижении пустого сегмента, не найдя элемента *х*, нельзя быть уверенным в том, что его вообще нет в таблице, так как сегмент может стать пустым уже после вставки элемента *х*. Поэтому, чтобы увеличить эффективность данной реализации, необходимо в сегмент, который освободился после операции удаления элемента, поместить специальную константу, которую назовем, например, **DEL**. В качестве альтернативы специальной константе можно использовать дополнительное поле таблицы, которое показывает состояние элемента. Важно различать константы **DEL** и **NULL** – последняя находится в сегментах, которые никогда не содержали элементов. При таком подходе выполнение поиска элемента не требует просмотра всей хеш-таблицы. Кроме того, при вставке элементов сегменты, помеченные константой **DEL**, можно трактовать как свободные, таким образом, пространство, освобожденное после удаления элементов, можно рано или поздно использовать повторно. Но если невозможно непосредственно сразу после удаления элементов пометить освободившиеся сегменты, то следует предпочесть закрытому хешированию схему открытого хеширования.

# **80. Хеш-таблицы. Основные, наиболее часто используемые функции хэширования.**

***Модульное хэширование*** Это простой, эффективный и наиболее часто используемый метод хэширования. Он заключается в выборе в качестве размера таблицы **hashTableSize** простого числа и вычислении хэша как остатка от деления Key на **hashTableSize:** **ize hashTableSkeykeyh %**)( **=** для любого целочисленного ключа **Key**. Такая функция называется модульной хэш-функцией и изменяется от **0** до (**hashTableSize – 1**). Это можно представить следующим образом:

typedef int HashIndexType; HashIndexType Hash(int Key) { return Key % hashTableSize; } // модульное хэширование

***Мультипликативный метод***

Размер таблицы **hashTableSize** выбирается равным степени два (**hashTableSize = 2p**). Значение key умножается на константу A из диапазона **от 0 ≤ A ≤ 1**. В качестве такой константы в **[8]** рекомендуется выбирать золотое сечение **A = (sqrt(5) – 1) / 2 = 0,6180339887499**. А затем выполнить деление по модулю **hashTableSize**. Другими словами, необходимо использовать **функцию ( ) ( mod1) h key hashTableSize key A =× ⎢⎥ ⎣ ⎦**.

***Аддитивный метод***

Для строк переменной длины (размер таблицы равен **256**) аддитивный метод дает вполне разумные результаты. В этом случае результат h заключен между **0 и 255**:

Исключающее **ИЛИ И**спользуется для строк переменной длины (размер таблицы равен 256). Этот метод аналогичен аддитивному, но успешно различает схожие

typedef unsigned char HashIndexType; HashIndexType Hash(char \*str) { HashIndexType h = 0; while (\*str) h += \*str++; return h; }

слова и анаграммы (аддитивный метод даст одно значение для **XY и YX**). Метод заключается в том, что к элементам строки последовательно применяется операция «исключающее или»..

***Универсальное хэширование***

Основано на выборе хэш-функции во время исполнения программы случайным образом из некоторого множества. Теоретически идеальная универсальная хэш-функция – это функция, для которой вероятность конфликта между двумя различными ключами в таблице размером **hashTableSize** равна в точности **1/hashTableSize**. Использование в качестве коэффициента А в мультипликативном методе последовательности различных случайных значений вместо фиксированного произвольного значения преобразует мультипликативное хэширование в универсальную хэш-функцию. Для аппроксимации вероятности возникновения конфликтов для двух несовпадающих ключей до значения **1/hashTableSize** вместо фиксированных оснований системы счисления применяются псевдослучайные значения коэффициентов. С целью минимизации нежелательных временных затрат при вычислении хэш-функции используется грубый генератор случайных чисел.

Типичная ошибка в реализациях хэширования заключается в том, что хэш-функция всегда возвращает одно и то же значение. Такая ошибка называется ошибкой производительности, поскольку использующая подобную хэш-функцию программа вполне может выполняться корректно, но крайне медленно (т.к. ее эффективная работа возможна только, если хэш-значения распределены равномерно). Однострочные реализации функций легко тестировать, поэтому рекомендуется проверять, насколько успешно они работают для типов ключей, которые могут встретиться в любой конкретной реализации таблицы символов. Для проверки гипотезы, что хэш-функция создает случайные значения, можно использовать функцию статистического распределения χ2, но на практике достаточно использовать проверку того, что значения распределены до такой степени, чтобы ни одно из них не доминировало.

# 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.

**Разрешение коллизиц** в [хеш-таблице](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0), задача, решаемая несколькими способами: метод цепочек, открытая адресация и т.д. Очень важно сводить количество коллизий к минимуму, так как это увеличивает время работы с хеш-таблицами.

Разрешение коллизий с помощью цепочек

Каждая ячейка ii массива HH содержит указатель на начало [списка](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA) всех элементов, хеш-код которых равен ii, либо указывает на их отсутствие. Коллизии приводят к тому, что появляются списки размером больше одного элемента

Линейное разрешение коллизий[[править](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A0%D0%B0%D0%B7%D1%80%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%BB%D0%BB%D0%B8%D0%B7%D0%B8%D0%B9&action=edit&section=2)]

Все элементы хранятся непосредственно в хеш-таблице, без использования связных списков. В отличие от хеширования с цепочками, при использовании этого метода может возникнуть ситуация, когда хеш-таблица окажется полностью заполненной, следовательно будет невозможно добавлять в неё новые элементы. Так что при возникновении такой ситуации решением может быть динамическое увеличение размера хеш-таблицы, с одновременной её перестройкой.

# 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, при каждом проходе по массиву с конца будет отрезаться самый большой элемент и он перестанет учавствовать в сравнении.  
**Реализация.**

|  |  |
| --- | --- |
| void bubbleSort(int arr[], int n) {  for (int i = 0; i < n - 1; ++i) {  for (int j = 0; j < n - i - 1; ++j) {  if (arr[j] > arr[j + 1]) {  // Обмен значениями  int temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }  } | int main() {  int n = 7;  int arr[7] = {64, 34, 25, 12, 22, 11, 90};  std::cout << "Исходный массив: ";  for (int i = 0; i < n; ++i) {  std::cout << arr[i] << " ";  }  std::cout << std::endl;  bubbleSort(arr, n);  std::cout << "Отсортированный массив: ";  for (int i = 0; i < n; ++i) {  std::cout << arr[i] << " ";  }  std::cout << std::endl;  return 0;  } |

**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.

**Реализация.**

|  |  |
| --- | --- |
| void insertionSort(int arr[], int n) {  for (int i = 1; i < n; ++i) {  int key = arr[i];  int j = i - 1;  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j = j - 1;  }  arr[j + 1] = key;  }  } | int main() {  int arr[] = { 64, 34, 25, 12, 22, 11, 90 };  int n = sizeof(arr) / sizeof(arr[0]);  std::cout << "Исходный массив: ";  for (int i = 0; i < n; ++i) {  std::cout << arr[i] << " ";  }  std::cout << std::endl;  insertionSort(arr, n);  std::cout << "Отсортированный массив: ";  for (int i = 0; i < n; ++i) {  std::cout << arr[i] << " ";  }  std::cout << std::endl;  return 0;  } |

**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.

**Реализация.**

|  |  |
| --- | --- |
| void selectionSort(int arr[], int n) {  for (int i = 0; i < n - 1; ++i) {  // Ищем минимальный элемент в неотсортированной части массива  int minIndex = i;  for (int j = i + 1; j < n; ++j) {  if (arr[j] < arr[minIndex]) {  minIndex = j;  }  }  // Обменяем найденный минимальный элемент с первым элементом неотсортированной части  int temp = arr[minIndex];  arr[minIndex] = arr[i];  arr[i] = temp;  }  } | int main() {  int arr[] = { 64, 34, 25, 12, 22, 11, 90 };  int n = sizeof(arr) / sizeof(arr[0]);  std::cout << "Исходный массив: ";  for (int i = 0; i < n; ++i) {  std::cout << arr[i] << " ";  }  std::cout << std::endl;  selectionSort(arr, n);  std::cout << "Отсортированный массив: ";  for (int i = 0; i < n; ++i) {  std::cout << arr[i] << " ";  }  std::cout << std::endl;  return 0;  } |

# **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.

**Реализация.**

|  |  |
| --- | --- |
| int main() {  int arr[] = { 64, 34, 25, 12, 22, 11, 90 };  int n = sizeof(arr) / sizeof(arr[0]);  std::cout << "Исходный массив: ";  for (int i = 0; i < n; ++i) {  std::cout << arr[i] << " ";  }  std::cout << std::endl;  mergeSort(arr, 0, 6);  std::cout << "Отсортированный массив: ";  for (int i = 0; i < n; ++i) {  std::cout << arr[i] << " ";  }  std::cout << std::endl;  return 0;  } | void merge(int A[], int low, int high, int mid)  {  int i, j, k, c[50];  i = low;  k = low;  j = mid + 1;  while (i <= mid && j <= high)  {  if (A[i] < A[j])  {  c[k] = A[i];  k++;  i++;  }  else  {  c[k] = A[j];  k++;  j++;  }  }  while (i <= mid)  {  c[k] = A[i];  k++;  i++;  }  while (j <= high)  {  c[k] = A[j];  k++;  j++;  }  for (i = low; i < k; i++)  {  A[i] = c[i];  }  } |
| void mergeSort(int A[], int low, int high) {  if (low < high) {  int mid;  mid = (low + high) / 2;  mergeSort(A, low, mid);  mergeSort(A, mid + 1, high);  merge(A, low, high, mid);  }  return;  } |

**Анализ.** Сортировка слиянием в среднем столь же эффективна, как и быстрая.

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

**Описание алгоритма** Быстрая сортировка (Quick Sort) — это популярный алгоритм сортировки, который использует подход "разделяй и властвуй". Она работает, рекурсивно разделяя массив на две части и сортируя их отдельно. Быстрая сортировка является одним из самых эффективных алгоритмов сортировки для больших массивов.

**Реализация.**

|  |  |
| --- | --- |
| // Вспомогательная функция для перестановки двух элементов  void swap(int\* a, int\* b) {  int t = \*a;  \*a = \*b;  \*b = t;  }  // Функция для разделения массива и нахождения опорного элемента  int partition(int arr[], int low, int high) {  int pivot = arr[high]; // Опорный элемент  int i = (low - 1); // Индекс меньшего элемента  for (int j = low; j <= high - 1; j++) {  // Если текущий элемент меньше или равен опорному  if (arr[j] <= pivot) {  i++; // Увеличиваем индекс меньшего элемента  swap(&arr[i], &arr[j]);  }  }  swap(&arr[i + 1], &arr[high]);  return (i + 1);  }  // Основная функция, реализующая быструю сортировку  void quickSort(int arr[], int low, int high) {  if (low < high) {  // pi - индекс опорного элемента, arr[pi] на правильной позиции  int pi = partition(arr, low, high);  // Рекурсивно сортируем элементы до и после разбиения  quickSort(arr, low, pi - 1);  quickSort(arr, pi + 1, high);  }  } | int main() {  int arr[] = { 64, 34, 25, 12, 22, 11, 90 };  int n = sizeof(arr) / sizeof(arr[0]);  std::cout << "Исходный массив: ";  for (int i = 0; i < n; ++i) {  std::cout << arr[i] << " ";  }  std::cout << std::endl;  quickSort(arr, 0, n - 1);  std::cout << "Отсортированный массив: ";  for (int i = 0; i < n; ++i) {  std::cout << arr[i] << " ";  }  std::cout << std::endl;  return 0;  } |

**Анализ.**

Быстрая сортировка имеет среднюю временную сложность O(n log n), что делает её эффективной для сортировки больших массивов. Однако в худшем случае (например, если массив уже отсортирован) её сложность может быть O(n^2), поэтому часто используются улучшенные версии алгоритма с оптимизацией выбора опорного элемента.

# **89. Сортировка данных. Сортировка подсчётом. Простой алгоритм. Алгоритм со списком. Алгоритм с двусвязным списком. Сортировка вставками.**

**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.

**#include <stdio.h>**

**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];**

**}**

**}**

**void main()**

**{ int A[] = {41, 7, 8, 40, 8};**

**int i; int n = sizeof(A) / sizeof(int);**

**int B[5] = {0};**

**CountSort(A, B, n);**

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

**printf("%d ", B[i]);**

**}**

Общее количество сравнений:

**(*n* –1 ) + (*n* – 2) +…+ 1 *= n* (*n* – 1) / 2 *≈ n*2 / 2**

Максимальное число перестановок − **(*n* – 1).**

Перестановки затратны по времени.

Трудоемкость алгоритма – **n2 / 2**.

Это **простейший** вариант алгоритма. Создать вспомогательный массив C[0..k - 1], состоящий из нулей, затем последовательно прочитать элементы входного массива A, для каждого A[i] увеличить C[A[i]] на единицу. Теперь достаточно пройти по массиву C, для каждого **{\displaystyle j\in \{0,...,k-1\}}j Э {0,…,k-1}** в массив A последовательно записать число j C[j] раз.

SimpleCountingSort:

for i = 0 to k - 1

C[i] = 0;

for i = 0 to n - 1

C[A[i]] = C[A[i]] + 1;

b = 0;

for j = 0 to k - 1

for i = 0 to C[j] - 1

A[b] = j;

b = b + 1;

**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;**

**}**

**}**

Внешний цикл выполняется **n - 1** раз, внутренний – не больше, чем **n - 1**.

**void main()**

**{ int A[] = {4, 7, 3, 4, 8, 2, 3, 6, 6, 4, 9, 2, 3};**

**int n = sizeof(A) / sizeof(int);**

**InsertSort(A, n);**

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

**cout<<A[i]<<' ';**

**}**

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.

Общее быстродействие - **O(n2),** но ввиду простоты метода является наиболее быстрым для малых (12-20 элементов) массивов.

Если использовать двоичный поиск для определения места элемента, то трудоемкость алгоритма составляет **n \* log(n)**. Однако эта трудоемкость не учитывает затрат на перемещение элементов при сдвиге.

# **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.

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).

 По виду данных:

сортировка массивов, массивов указателей, списков и других структур данных.

 По способу выбора элементов:

- **ОБМЕННАЯ** сортировка (“пузырек”, шейкер-сортировка): выполняется путем перестановки элементов по определенному правилу;

- сортировка **ВЫБОРОМ**  (прямая выборка, линейный выбор с подсчетом, квадратичная сортировка с выбором): выбирается очередной минимальный элемент и помещается в конец (начало) последовательности;

- сортировка **ВСТАВКАМИ** (простая вставка, вставка погружением, сортировка Шелла): очередной элемент помещается по месту своего расположения в выходную последовательность (массив);

- сортировка **РАЗДЕЛЕНИЕМ** (быстрая сортировка разделением, поразрядная сортировка разделением): основана на разделении элементов последовательности на части и сортировки каждой из частей независимо друг от друга итерационно или рекурсивно;

- сортировка **ПОДСЧЕТОМ**: определяется количество элементов, больших или меньших данного элемента;

- **ПИРАМИДАЛЬНАЯ** сортировка: строится пирамида, при сортировке выполняется просеивание пирамиды;

- сортировка **СЛИЯНИЕМ** (однократное слияние, циклическое слияние): последовательность (массив) регулярно распределяется в несколько последовательностей, которые затем объединяются;

**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.

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).

 По виду данных:

сортировка массивов, массивов указателей, списков и других структур данных.

 По способу выбора элементов:

- **ОБМЕННАЯ** сортировка (“пузырек”, шейкер-сортировка): выполняется путем перестановки элементов по определенному правилу;

- сортировка **ВЫБОРОМ**  (прямая выборка, линейный выбор с подсчетом, квадратичная сортировка с выбором): выбирается очередной минимальный элемент и помещается в конец (начало) последовательности;

- сортировка **ВСТАВКАМИ** (простая вставка, вставка погружением, сортировка Шелла): очередной элемент помещается по месту своего расположения в выходную последовательность (массив);

- сортировка **РАЗДЕЛЕНИЕМ** (быстрая сортировка разделением, поразрядная сортировка разделением): основана на разделении элементов последовательности на части и сортировки каждой из частей независимо друг от друга итерационно или рекурсивно;

- сортировка **ПОДСЧЕТОМ**: определяется количество элементов, больших или меньших данного элемента;

- **ПИРАМИДАЛЬНАЯ** сортировка: строится пирамида, при сортировке выполняется просеивание пирамиды;

- сортировка **СЛИЯНИЕМ** (однократное слияние, циклическое слияние): последовательность (массив) регулярно распределяется в несколько последовательностей, которые затем объединяются;

**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.

На каждом шаге выбирается минимальный из последующих узлов пирамиды и помещается в вершину. И т. д. по цепочке.

Всего выполняется **n - 1** шагов.

**#include <iostream>**

**using namespace std;**

**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);**

**}**

**}**

**void main()**

**{ int n; int A[100];**

**cout<<"Razmer "; cin>>n;**

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

**{ cout<<"Number= "; cin>> A[i]; }**

**PiramSort(A, n);**

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

**cout << A[i]<<' ';**

**}**

Преимущество ал­го­рит­ма в том, что пи­ра­ми­да хра­нит­ся пря­мо в ис­ход­ном мас­си­ве. По ме­ре то­го, как раз­мер пи­ра­ми­ды умень­ша­ет­ся, она за­ни­ма­ет всё мень­шую часть мас­си­ва, а ре­зуль­тат сор­ти­ров­ки за­пи­сы­ва­ет­ся, на­чи­ная с кон­ца мас­си­ва на осво­бо­див­шие­ся от пи­ра­ми­ды ме­ста.

Пирамидальная сортировка проигрывает быстрой сортировке по эффективности для больших массивов (в 1,5 раза медленнее).

Среднее число пересылок - **(n\*logn)/2**

Трудоемкость − **O(nlog(n)).**

**Другая** программа

**#include <iostream>**

**#include <conio.h>**

**using namespace std;**

**void iswap(int &n1, int &n2)**

**{ int temp = n1; n1 = n2; n2 = temp;}**

**int main()**

**{ int const n = 100; int a[n];**

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

**bool b = false; int sh = 0;** //смещение

**for(;;)**

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

**{ if( i \* 2 + 2 + sh < n )**

**{ if((a[i+sh]> a[i\*2+1+sh]) || (a[i+sh]>a[i \* 2 + 2 + sh]))**

**{ if ( a[i \* 2 + 1 + sh] < a[i \* 2 + 2 + sh] )**

**{ iswap( a[i + sh], a[i \* 2 + 1 + sh] ); b = true; }**

**Else if ( a[i \* 2 + 2 + sh] < a[ i \* 2 + 1 + sh])**

**{ iswap( a[ i + sh], a[i \* 2 + 2 + sh]); b = true; }**

**}**

//дополнительная проверка для последних двух элементов

//с помощью этой проверки можно отсортировать пирамиду

//состоящую всего лишь из трех элементов

**if( a[i\*2 + 2 + sh] < a[i\*2 + 1 + sh] )**

**{ iswap( a[i\*2+1+sh], a[i \* 2 +2+ sh] ); b = true; }**

**}**

**else if( i \* 2 + 1 + sh < n )**

**{ if( a[i + sh] > a[ i \* 2 + 1 + sh] )**

**{ iswap( a[i + sh], a[i \* 2 + 1 + sh] ); b = true; } }**

**}**

**if (!b) sh++;** //смещение увелич. когда на тек. этапе сортировать нечего

**if ( sh + 2 == n ) break; }** //конец сортировки

**cout << endl << endl;**

**for ( int i = 0; i < n; ++i ) cout << a[i] << " ";**

**return 0; }**

# **92.Сортировка данных. Поразрядная сортировка.**

**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).

 По виду данных:

сортировка массивов, массивов указателей, списков и других структур данных.

 По способу выбора элементов:

- **ОБМЕННАЯ** сортировка (“пузырек”, шейкер-сортировка): выполняется путем перестановки элементов по определенному правилу;

- сортировка **ВЫБОРОМ**  (прямая выборка, линейный выбор с подсчетом, квадратичная сортировка с выбором): выбирается очередной минимальный элемент и помещается в конец (начало) последовательности;

- сортировка **ВСТАВКАМИ** (простая вставка, вставка погружением, сортировка Шелла): очередной элемент помещается по месту своего расположения в выходную последовательность (массив);

- сортировка **РАЗДЕЛЕНИЕМ** (быстрая сортировка разделением, поразрядная сортировка разделением): основана на разделении элементов последовательности на части и сортировки каждой из частей независимо друг от друга итерационно или рекурсивно;

- сортировка **ПОДСЧЕТОМ**: определяется количество элементов, больших или меньших данного элемента;

- **ПИРАМИДАЛЬНАЯ** сортировка: строится пирамида, при сортировке выполняется просеивание пирамиды;

- сортировка **СЛИЯНИЕМ** (однократное слияние, циклическое слияние): последовательность (массив) регулярно распределяется в несколько последовательностей, которые затем объединяются;

**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.

**#include <iostream>**

**using namespace std;**

**const int n = 8;**

**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++;**

**}**

**}**

**}**

**void main()**

**{ int razr, i; int col\_razr=2;**

**int A[]={62, 54, 12, 23, 52, 61, 03, 54};**

**int B[n][n];**

**for(razr = 1; razr < col\_razr + 1; razr++)**

**PorazrSort(B, A, razr);**

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

**cout<<A[i]<<endl;**

**}**

Временная эффективность алгоритма – **Θ(T(n+k)),** где **T** – количество разрядов, **k** – диапазон значений разрядов.

**Недостаток** метода – использование дополнительной памяти.