Лабораторная работа №3

**Динамическая память**

**Цель работы:** изучить принципы работы с памятью и функции для работы с ней. Научиться разрабатывать программы с использованием динамических структур на смежной памяти.

Теория

**Понятие динамической памяти**

Память, которую использует программа делится на статическую память, стек и кучу.

**Статическая память** (static memory):

* хранит глобальные переменные и константы;
* размер определяется при компиляции.

**Стек** (stack):

* хранит локальные переменные, аргументы функций и промежуточные значения вычислений;
* размер определяется при запуске программы (обычно выделяется 4 Мб).

**Куча** (heap):

* динамически распределяемая память;
* ОС выделяет память по частям (по мере необходимости).

**Динамическая память** – это область (блок) памяти выделенный для нужд программы в процессе работы программы (а не заранее).

Динамически распределяемую память следует использовать в случае если мы заранее (на момент написания программы) не знаем сколько памяти нам понадобится (например, размер массива зависит от того, что введет пользователь во время работы программы) и при работе с большими объемами данных (например, массив из 1 000 000 int'ов не поместится на стеке). Основными действиями над динамической памятью являются: выделение и освобожде-

ние. В языке С++ функции для осуществления этих действий описаны в библиотеке cstdlib.

Стандартные функции динамического выделения памяти

Для использования функций динамического выделения памяти необходимо описать указатель, представляющий собой начальный адрес хранения элементов массива.

int \*p; // указатель на тип int

Начальный адрес статического массива определяется компилятором в момент его объявления и не может быть изменен.

Для динамического массива начальный адрес присваивается объявленному указателю на массив в процессе выполнения программы.

Основу системы динамического распределения в С++ составляют функции malloc() и free(). Эти функции работают совместно. Функция malloc() выделяет память, а free() – освобождает ее. Это значит, что при каждом запросе функция malloc() выделяет требуемый участок свободной памяти, a free() освобождает его, то есть возвращает системе.

Кроме того, есть еще 2 «модернизированные» функции работы с памятью – calloc() и realloc(). Функция calloc() работает аналогично malloc(), но отличается синтаксисом и тем, что выделенная память будет обнулена, а функция realloc() изменяет размер выделенной памяти (на которую указывает ptr, ранее полученный из вызова malloc, calloc или realloc).

В программу, использующую эти функции, должен быть включен заголовочный файл

<cstdlib>.

Работа с памятью также практически невозможна без унарного оператора sizeof(), возвращающего длину в байтах переменной или типа, помещенных в скобки. Рассмотрим эти функции более подробно.

Оператор sizeof

**sizeof** – это унарный оператор, возвращающий длину в байтах переменной или типа, помещенных в скобки.

float f;

cout << "Размер переменной f: " << sizeof f << endl; cout << "Размер типа double: " << sizeof(double) << endl;

Результат:



Помните, что для вычисления размера типа следует поместить имя типа в круглые скобки, а имена переменных можно писать и без скобок.

Использование sizeof помогает создавать переносимый код для тех случаев, когда код зависит от размера стандартных типов данных. Например, представим, что программе, работающей с базой данных, необходимо сохранять 6 целочисленных значений в записи. Для того, чтобы сделать эту программу переносимой, не следует предполагать, что размер целочисленного типа – 2 или 4 байта, следует самостоятельно определить настоящую длину, используя sizeof. Следующая подпрограмма может быть использована для сохранения записи в файле:

/\* Запись записи в файл \*/

void put\_rec(FILE \*fp, int rec[6]) { int size = sizeof(int) \* 6,

num = fwrite(rec, size, 1, fp);

if (num != 1) cout << "Ошибка записи";

}

Самое главное в этом примере то, что put\_rec() будет компилироваться и корректно запускаться на любом компьютере, включая компьютеры, использующие 2- и 4-байтовые целые.

Функция malloc()

Прототип функции **malloc()** следующий:

void \* malloc(size\_t количество\_байтов);

Здесь количество\_байтов – размер памяти, необходимой для размещения данных. (Тип size\_t определен в <cstdlib> как некоторый целый без знака.) Функция malloc() возвращает указатель типа void \*, поэтому его можно присвоить указателю любого типа. При успешном выполнении malloc() возвращает указатель на первый байт непрерывного участка памяти, выделенного в динамически распределяемой области памяти. Если в динамически распределяемой области памяти недостаточно свободной памяти для выполнения запроса, то память не выделяется и malloc() возвращает нуль.

При выполнении следующего фрагмента программы выделяется непрерывный участок памяти объемом 1000 байтов:

char \*p;

p = (char \*) malloc(1000); // выделение 1000 байтов

После присвоения указатель p ссылается на первый из 1000 байтов выделенного участка памяти. В следующем примере выделяется память для 50 целых. Для повышения мобильности (переносимости программы с одной машины на другую) используется оператор sizeof.

int \*p = (int \*) malloc(50 \* sizeof(int));

Поскольку динамически распределяемая область памяти не бесконечна, при каждом размещении данных необходимо проверять, состоялось ли оно. Если malloc() не смогла по какой-либо причине выделить требуемый участок памяти, то она возвращает нуль. В следующем примере показано, как выполняется проверка успешности размещения:

int \*p = (int \*) malloc(100);

if (!p) { cout << "Нехватка памяти." << endl; exit(1); }

Конечно, вместо выхода из программы exit() можно поставить какой-либо обработчик ошибки. Обязательным здесь можно назвать лишь требование не использовать указатель р, если он равен нулю.

Функция сalloc()

void \* calloc(size\_t num, size\_t size);

Функция calloc() выделяет память, размер которой равен значению выражения num \* size, т.е. память, достаточную для размещения массива, содержащего num объектов размером size. Все биты распределенной памяти инициализируются нулями.

Функция calloc() возвращает указатель на первый байт выделенной области памяти. Если для удовлетворения запроса нет достаточного объема памяти, возвращается нулевой указатель. Перед попыткой использовать распределенную память важно проверить, что возвращаемое значение не равно нулю.

**Пример.** Эта функция возвращает указатель на динамически распределенный блок памяти для массива из 100 чисел типа float:

float \* get\_mem(void) {

float \*p = (float \*) calloc(100, sizeof(float));

if (!p) { cout << "Ошибка при выделении памяти" << std; exit(1); }

return p;

}

Функция realloc()

void \*realloc(void \*ptr, size\_t size);

Функция realloc() возвращает указатель на блок памяти указанного размера, в который копируется содержимое блока ptr (до size байтов). Причем допускается, чтобы новый и старый блоки начинались с одинакового адреса (т.е. указатель, возвращаемый функцией realloc(), может совпадать с указателем, переданным в параметре ptr).

Если указатель ptr нулевой, функция realloc() просто выделяет size байтов памяти и возвращает указатель на эту память. Если значение параметра size равно нулю, память, адресуемая параметром ptr, освобождается.

Если в динамически распределяемой области памяти нет достаточного объема свободной памяти для выделения size байтов, возвращается нулевой указатель, а исходный блок памяти остается неизменным.

**Пример.** Эта программа сначала выделяет блок памяти для 17 символов, копирует в них строку "Это - 17 символов", а затем использует realloc() для увеличения размера блока до 18 символов, чтобы разместить в конце точку.

int main(void) {

char \*p = (char \*) malloc(18);

if (!p) { cout << "Ошибка!" << cout; exit(1); } strcpy(p, "Это - 17 символов");

p = (char \*) realloc(p, 19);

if (!p) { cout << "Ошибка!" << endl; exit(1); }

strcat(p, "."); printf(p); free(p);

return 0;

}

Функция free()

void free(void \*ptr);

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

В качестве входного параметра в free() нужно передать указатель, значение которого получено из функции malloc. Вызов free на указателях, полученных не из malloc (например, free(p+10)), приведет к неопределенному поведению. Это связанно с тем, что при выделении памяти при помощи malloc в ячейки перед той, на которую указывает возвращаемый функцией указатель операционная система записывает служебную информацию. При вызове free(p+10) информация, находящаяся перед ячейкой (p+10), будет трактоваться как служебная.

**Пример.** Эта программа распределяет блок памяти для вводимых пользователем строк, а затем освобождает блок памяти:

int main(void) {

char \*str[5]; int i;

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

if ((str[i] = (char \*) malloc(128)) == NULL) {

cout << "Ошибка при распределении памяти" << endl; exit(1);

}

cin >> str[i];

}

/\* Освобождение блока памяти \*/

for (i = 0; i < 5; i++) free(str[i]);

return 0;

}

Динамическое выделение памяти для одномерных массивов

Форма обращения к элементам массива с помощью указателей имеет следующий вид:

int a[10], \*p; // описываем статический массив и указатель int b;

p = a; // присваиваем указателю начальный адрес массива

... // ввод элементов массива

b = \*p; // b = a[0]; b = \*(p + i) // b = a[i]

**Пример.** Организация динамического одномерного массива и ввод его элементов.

int main() {

int \*a; // указатель на массив int i, n;

cout << " Введите размер массива: "; cin >> n;

// Выделение памяти

a = (int \*) malloc(n \* sizeof(int));

// Ввод элементов массива for (i = 0; i < n; i++) {

cout << " a[" << i << "] = "; cin >> a[i];

}

// Вывод элементов массива for (i = 0; i < n; i++) {

cout << " " << a[i];

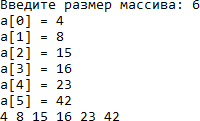
}

cout << endl; free(a);

return 0;

}

Результат:



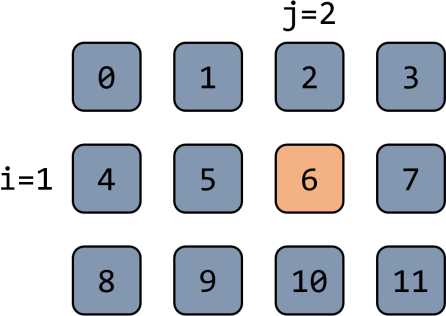
Динамическое выделение памяти для двумерных массивов

Пусть требуется разместить в динамической памяти матрицу, содержащую n строк и m столбцов. Двумерная матрица будет располагаться в оперативной памяти в форме ленты, состоящей из элементов строк. При этом индекс любого элемента двумерной матрицы можно получить по формуле:

𝑖𝑛𝑑𝑒𝑥 = 𝑖 ∙ 𝑚 + 𝑗,

где 𝑖 – номер текущей строки; 𝑗 – номер текущего столбца.

Рассмотрим матрицу 3 × 4:



Индекс выделенного элемента определится как: 𝑖𝑛𝑑𝑒𝑥 = 1 ∙ 4 + 2 = 6.

Объем памяти, требуемый для размещения двумерного массива, определится как:

𝑛 ∙ 𝑚 ∙ (размер элемента)

Однако поскольку при таком объявлении компилятору явно не указывается количество элементов в строке и столбце двумерного массива, традиционное обращение к элементу путем указания индекса строки и индекса столбца является некорректным:

a[i][j] – некорректно.

Правильное обращение к элементу с использованием указателя будет выглядеть как:

\*(p + i \* m + j),

где p – указатель на массив, m – количество столбцов, i – индекс строки, j – индекс столбца.

**Пример.** Ввод и вывод значений динамического двумерного массива.

int main() {

int \*a; // указатель на массив int i, j, n, m;

cout << " Введите количество строк: "; cin >> n; cout << " Введите количество столбцов: "; cin >> m;

// Выделение памяти

a = (int \*) calloc(n \* m, sizeof(int));

// Ввод элементов массива

for (i = 0; i < n; i++) { // цикл по строкам for (j = 0; j < m; j++) { // цикл по столбцам

cout << "a[" << i << "][" << j << "] = ";

cin >> \*(a + i \* m + j);

}

}

// Вывод элементов массива

for (i = 0; i < n; i++) { // цикл по строкам for (j = 0; j < m; j++) // цикл по столбцам

cout << \*(a + i \* m + j) << " "; cout << endl;

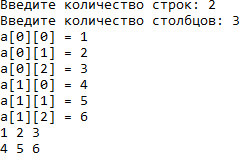
}

free(a);

return 0;

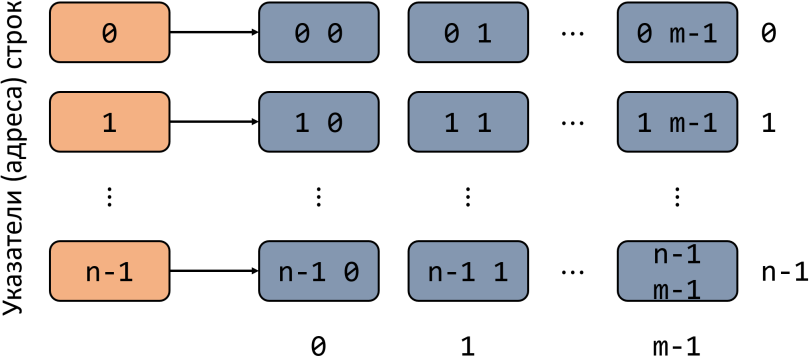
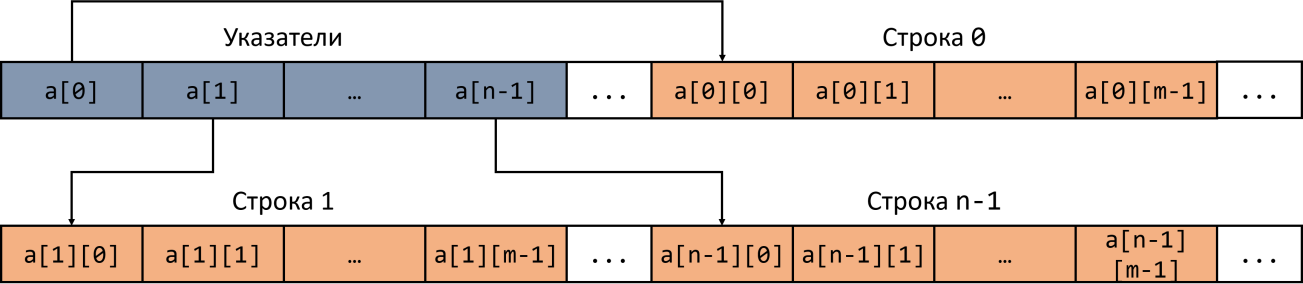
}

Результат:



Возможен также другой способ динамического выделения памяти под двумерный массив – с использованием массива указателей. Для этого необходимо:

* выделить блок оперативной памяти под массив указателей;
* выделить блоки оперативной памяти под одномерные массивы, представляющие собой строки искомой матрицы;
* записать адреса строк в массив указателей.

Графически такой способ выделения памяти можно представить так, как показано на рисунках:

При таком способе выделения памяти компилятору явно указано количество строк и количество столбцов в массиве.

int main() {

int \*\*a; // указатель на указатель на строку int i, j, n, m;

cout << "Введите количество строк: "; cin >> n;

cout << "Введите количество столбцов: "; cin >> m;

// Выделение памяти под указатели на строки a = (int \*\*) malloc(n \* sizeof(int \*));

// Ввод элементов массива

for (i = 0; i < n; i++) { // цикл по строкам

// Выделение памяти под хранение строк a[i] = (int \*) malloc(m \* sizeof(int));

for (j = 0; j < m; j++) { // цикл по столбцам cout << "a[" << i << "][" << j << "] = ";

cin >> a[i][j];

}

}

// Вывод элементов массива

for (i = 0; i < n; i++) { // цикл по строкам for (j = 0; j < m; j++) // цикл по столбцам

cout << a[i][j] << " "; cout << endl;

free(a[i]); // освобождение памяти под строку

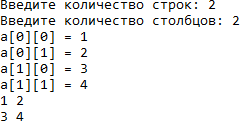
}

free(a);

return 0;

}

Результат выполнения программы аналогичен предыдущему случаю:



С помощью динамического выделения памяти под указатели строк можно размещать свободные массивы. **Свободным** называется двухмерный массив (матрица), размер строк которого может быть различным. Преимущество использования свободного массива заключается в том, что не требуется отводить память компьютера с запасом для размещения строки максимально возможной длины. Фактически свободный массив представляет собой одномерный массив указателей на одномерные массивы данных.

Для размещения в оперативной памяти матрицы со строками разной длины необходимо ввести дополнительный массив m, в котором будут храниться размеры строк.

Пример:

int main() {

int \*\*a;

int i, j, n, \*m;

cout << "Введите количество строк: "; cin >> n;

a = (int \*\*) malloc(n \* sizeof(int \*)); m = (int \*) malloc(n \* sizeof(int));

// Ввод элементов массива for (i = 0; i < n; i++) {

cout << "Введите количество столбцов строки " << i << ": "; cin >> m[i];

a[i] = (int \*) malloc(m[i] \* sizeof(int)); for (j = 0; j <m [i]; j++) {

cout << "a[" << i << "][" << j << "] = ";

cin >> a[i][j];

}

}

// Вывод элементов массива for (i = 0; i < n; i++) {

for (j = 0; j < m[i]; j++)

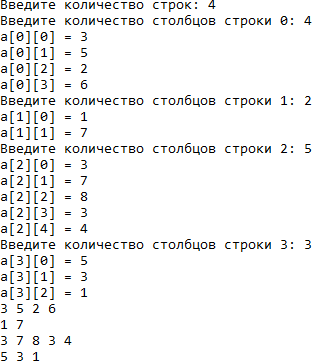
cout << a[i][j]; cout << endl;

}

return 0;

}

Результат:



Перераспределение памяти

Если размер выделяемой памяти нельзя задать заранее, например, при вводе последовательности значений до определенной команды, то для увеличения размера массива при вводе следующего значения необходимо выполнить следующие действия:

* Выделить блок памяти размерности n+1 (на 1 больше текущего размера массива).
* Скопировать все значения, хранящиеся в массиве во вновь выделенную область памяти.
* Освободить память, выделенную ранее для хранения массива.
* Переместить указатель начала массива на начало вновь выделенной области памяти.
* Дополнить массив последним введенным значением.

Все перечисленные выше действия (кроме последнего) выполняет следующая функция:

void \* realloc(void\* ptr, size\_t size);

Размер блока памяти, на который ссылается параметр ptr изменяется на size байтов. Блок памяти может уменьшаться или увеличиваться в размере. Содержимое блока памяти сохраняется даже если новый блок имеет меньший размер, чем старый. Но отбрасываются те данные, которые выходят за рамки нового блока. Если новый блок памяти больше старого, то содержимое вновь выделенной памяти будет неопределенным.

**Пример.** Выделить память для ввода массива целых чисел. После ввода каждого значения задавать вопрос о вводе следующего значения.

int main() {

int \*a = NULL, i = 0, elem;

do {

cout << "a[" << i << "] = "; cin >> elem; a = (int \*) realloc(a, (i + 1) \* sizeof(int)); a[i++] = elem;

} while (elem);

cout << "n: ";

for (int j = 0; j < i; j++) cout << a[j] << " ";

// уменьшение размера на 2 if (i > 2) i -= 2;

cout << endl << "n-2: ";

a = (int \*) realloc(a, i \* sizeof(int));

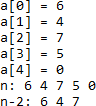
for (int j = 0; j < i; j++) cout << a[j] << " ";

cout << endl;

return 0;

}

Результат:



Практика

**Задание 1.** Создать указатель на переменную типа (а), выделить для нее динамическую память, считать ее значение и вывести на экран. Освободить память.

**Задание 2.** Создать указатель на строку длиной N символов. Ввести строку с клавиатуры. Уменьшить длину строки на 5 символов в конце добавить символ '\0'. Вывести результат на экран. Увеличить длину строки на M символов. Заполнить последние M элементов строки символом '\*'. Вывести результат на экран. Очистить память.

**Задание 3.** Создать массив типа (б), выделить память на N элементов. Считать элементы с клавиатуры и вывести полученный массив на экран. Очистить память.

* **Задание 4.** Создать свободный массив типа (а). Считать количество элементов в строке и значения элементов с клавиатуры. Вывести результат на экран. Очистить память.

**\*\* Задание 5.** В динамический свободный массив записывать строки, заранее считанные с клавиатуры. То есть, сначала считывается строка, затем определяется ее длина, после чего она заносится в строку динамического массива. Вывести содержимое массива в виде текста. Очистить память.

|  |  |  |  |
| --- | --- | --- | --- |
| **Вариант** | **Задание 1** | **Задание 2** | **Задание 3** |
| 1 | 1. int 2. char | 1. int 2. double | 1. int 2. float |
| 2 | 1. float 2. char | 1. float 2. int | 1. float 2. int |
| 3 | 1. char 2. int | 1. char 2. float | 1. char 2. float |
| 4 | 1. double 2. char | 1. double 2. int | 1. double 2. int |
| 5 | 1. int 2. double | 1. int 2. char | 1. int 2. double |
| 6 | 1. float 2. int | 1. float 2. char | 1. float 2. int |
| 7 | 1. char 2. double | 1. char 2. int | 1. char 2. double |
| 8 | 1. double 2. int | 1. double 2. char | 1. double 2. char |
| 9 | 1. int 2. float | 1. int 2. float | 1. int 2. char |
| 10 | 1. float 2. int | 1. float 2. int | 1. float 2. char |
| 11 | 1. char 2. float | 1. double 2. char | 1. char 2. int |
| 12 | 1. char 2. double | 1. char 2. float | 1. float 2. int |
| 13 | 1. int 2. char | 1. double 2. int | 1. int 2. double |

Лабораторная работа №4

**Структуры и перечисления**

**Цель работы:** научиться работать с векторными данными языка С++: структурами и перечислениями. Изучить правила объявления структур и перечислений, обращение к их полям, построение вложенных векторных типов данных.

Теория

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

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

struct ADDR {

char name[30]; char street[40]; char city[20]; char state[3]; int zip;

};

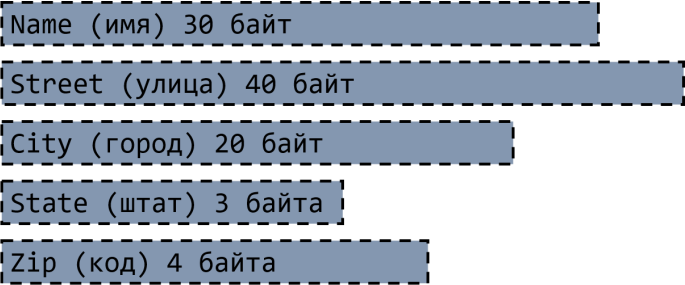
Обратите внимание, что объявление завершается точкой с запятой, потому что объявление структуры является оператором. Кроме того, тег структуры ADDR идентифицирует эту конкретную структуру данных и является спецификатором ее типа.

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

ADDR addr\_info;

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

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



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

struct ADDR {

char name[30]; char street[40]; char city[20]; char state[3]; int zip;

} addr\_info, binfo, cinfo;

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

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

struct {

char name[30]; char street[40]; char city[20]; char state[3];

unsigned long int zip;

} addr\_info;

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

Общий вид объявления структуры такой:

struct тег {

тип имя\_члена; тип имя\_члена;

...

} переменные\_структуры;

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

Доступ к членам структуры

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

addr\_info.zip = 12345;

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

имя\_объекта.имя\_члена

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

cout << addr\_info.zip;

Будет выведен ZIP-код, который находится в члене zip переменной-структуры addr\_infо. Точно так же в вызове gets() можно использовать массив символов addr\_infо.name:

gets(addr\_info.name);

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

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

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

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

Присваивание структур

Информация, которая находится в одной структуре, может быть присвоена другой структуре того же типа при помощи единственного оператора присваивания. Нет необходимости присваивать значения каждого члена в отдельности. Как выполняется присваивание структур, показывает следующая программа:

struct {

int a; int b;

} x, y;

int main(void) {

x.a = 10;

y = x; // присваивание одной структуры другой cout << y.a;

return 0;

}

После присвоения в **y.a** будет храниться значение **10**.

Массивы структур

Структуры часто образуют массивы. Чтобы объявить массив структур, вначале необходимо определить структуру (то есть определить агрегатный тип данных), а затем объявить переменную массива этого же типа. Например, чтобы объявить 100-элементный массив структур типа ADDR, который был определен ранее, напишите следующее:

ADDR addr\_list[100];

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

Чтобы получить доступ к определенной структуре, указывайте имя массива с индексом.

Например, чтобы вывести ZIP-код из третьей структуры, напишите следующее:

cout << addr\_list[2].zip;

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

**ДЛЯ СПРАВКИ:** чтобы указать определенную структуру, находящуюся в массиве структур, необходимо указать имя этого массива с определенным индексом. А если нужно указать индекс определенного элемента в структуре, то необходимо указать индекс этого элемента. Таким образом, в результате выполнения следующего выражения первому символу члена name, находящегося в третьей структуре из addr\_list, присваивается значение 'X'.

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

Передача членов структур функциям

При передаче функции члена структуры передается его значение, притом не играет роли то, что значение берется из члена структуры. Проанализируйте следующую структуру:

struct FRED {

char x; int y; float z;

char s[10];

} mike;

Обратите внимание, каким образом каждый член этой структуры передается функции:

func1(mike.x); // передается символьное значение x func2(mike.y); // передается целое значение y func3(mike.z); // передается значение с плавающей точкой z func4(mike.s); // передается адрес строки s func5(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:

func1(&mike.x); // передается адрес символа x func2(&mike.y); // передается адрес целого y

func3(&mike.z); // передается адрес члена z с плавающей точкой func4(mike.s); // передается адрес строки s func5(&mike.s[2]); // передается адрес символа s[2]

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

Передача целых структур функциям

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.

/\* Определение типа структуры \*/ struct struct\_type {

int a, b; char ch;

};

void f1(struct\_type parm);

int main(void) {

struct\_type arg; arg.a = 1000; f1(arg);

return 0;

}

void f1(struct\_type parm) { cout << 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().

Как уже говорилось, при передаче структуры тип аргумента должен совпадать с типом параметра. Для аргумента и параметра недостаточно просто быть физически похожими; должны совпадать даже имена их типов. Например, следующая версия предыдущей программы неправильная и компилироваться не будет. Дело в том, что имя типа для аргумента, используемого при вызове функции f1(), отличается от имени типа ее параметра.

Приведённая ниже программа неправильная и при её компиляции обнаружатся ошибки.

/\* Определение типа структур \*/ struct struct\_type {

int a, b; char ch;

};

/\* Определение структуры, похожей на \*

\* struct\_type, но с другими именам \*/ struct struct\_type2 {

int a, b; char ch;

};

void f1(struct\_type2 parm); int main(void) {

struct\_type arg; arg.a = 1000;

f1(arg); // несовпадение типов

return 0;

}

void f1(struct\_type2 parm) { printf("%d", parm.a);

}

Указатели на структуры

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

Объявление указателя на структуру

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

ADDR \*addr\_pointer;

Использование указателей на структуры

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

У такого способа, как передача любых (кроме самых простых) структур функциям, имеется один большой недостаток: при выполнении вызова функции, чтобы поместить структуру в стек, необходимы существенные ресурсы. (Вспомните, что аргументы передаются функциям через стек.) Впрочем, для простых структур с несколькими членами эти ресурсы являются не такими уж большими. Но если в структуре имеется большое количество членов или некоторые члены сами являются массивами, то при передаче структур функциям производительность может упасть до недопустимо низкого уровня. Как же решить эту проблему? Надо передавать не саму структуру, а указатель на нее.

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

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

struct BAL {

float balance; char name[80];

} person;

BAL \*p; // объявление указателя на структуру

p = &person; // адрес структуры person можно присвоить указателю p

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

p->balance

Оператор **->**, который обычно называют оператором стрелки, состоит из знака "минус", за которым следует знак "больше". Стрелка применяется вместо оператора точки тогда, когда для доступа к члену структуры используется указатель на структуру.

Чтобы увидеть, как можно использовать указатель на структуру, проанализируйте следующую простую программу, которая имитирует таймер, выводящий значения часов, минут и секунд:

/\* Программа-имитатор таймера \*/ #define DELAY 128000

struct MY\_TIME {

int hours; int minutes; int seconds;

};

void display(MY\_TIME \*t) { printf("%02d:", t->hours); printf("%02d:", t->minutes); printf("%02d\n", t->seconds);

}

void update(MY\_TIME \*t) { t->seconds++;

if (t->seconds == 60) { t->seconds = 0;

t->minutes++;

}

if (t->minutes == 60) { t->minutes = 0;

t->hours++;

}

if (t->hours == 24) t->hours = 0;

delay();

}

void delay(void) {

for (long int t = 1; t < DELAY; ++t);

}

int main(void) {

MY\_TIME systime;

systime.hours = 0;

systime.minutes = 0;

systime.seconds = 0;

for (;;) {

update(&systime); display(&systime);

}

return 0;

}

Эту программу можно настраивать, меняя определение DELAY.

В этой программе объявлена глобальная структура my\_time, но при этом не объявлены никакие другие переменные программы. Внутри же main() объявлена структура systime и она инициализируется значением 00:00:00. Это значит, что systime непосредственно видна только в функции main().

Функциям update() (которая изменяет значения времени) и display() (которая выводит эти значения) передается адрес структуры systime. Аргументы в обеих функциях объявляются как указатель на структуру my\_time.

Внутри update() и display() доступ к каждому члену systime осуществляется при помощи указателя. Так как функция update() принимает указатель на структуру systime, то она в состоянии обновлять значение этой структуры. Например, необходимо "в полночь", когда значение переменной, в которой хранится количество часов, станет равным 24, сбросить отсчет и снова сделать значение этой переменной равным 0. Для этого в update() имеется строка:

if (t->hours == 24) t->hours = 0;

Таким образом, компилятору дается указание взять адрес t (этот адрес указывает на переменную systime из main()) и сбросить значение hours в нуль.

Оператор точка используется для доступа к элементам структуры при работе с самой структурой. А когда используется указатель на структуру, то применяется оператор стрелка.

Массивы и структуры внутри структур

Членом структуры может быть или простая переменная, например, типа int или double, или составной (не скалярный) тип. В языке С составными типами являются массивы и структуры. Один составной тип вы уже видели – это символьные массивы, которые использовались в ADDR.

Члены структуры, которые являются массивами, можно считать такими же членами структуры, как и те, что нам известны из предыдущих примеров. Например, проанализируйте следующую структуру:

struct X {

int a[10][10]; // массив 10 x 10 из целых значений float b;

} y;

Целый элемент с индексами 3, 7 из массива a, находящегося в структуре y, обозначается таким образом:

y.a[3][7]

Когда структура является членом другой структуры, то она называется вложенной.

Например, в следующем примере структура address вложена в EMP:

struct EMP {

struct ADDR address; // вложенная структура float wage;

} worker;

Здесь структура была определена как имеющая два члена. Первым является структура типа ADDR, в которой находится адрес работника. Второй член – это wage, где находятся данные по его зарплате. В следующем фрагменте кода элементу zip из address присваивается значение 93456.

worker.address.zip = 93456;

Как вы видите, в каждой структуре любой член обозначают с помощью тех структур, в которые он вложен – начиная от самых общих и заканчивая той, непосредственно в которой он находится. В соответствии со стандартом С89 структуры могут быть вложенными вплоть до 15- го уровня. А стандарт С99 допускает уровень вложенности до 63-го включительно.

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

penny (пенни, монета в один цент),

nickel (никель, монета в пять центов), dime (монета в 10 центов),

quarter (25 центов, четверть доллара), half-dollar (полдоллара),

dollar (доллар).

Перечисления определяются во многом так же, как и структуры; началом объявления перечислимого типа (иногда используется термин перечисляемый тип) служит ключевое слово enum. Перечисление в общем виде выглядит так:

enum тег {

список\_перечисления

} список\_переменных;

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

enum coin {

penny, nickel, dime, quarter, half\_dollar, dollar

};

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

enum coin money;

С учетом этих объявлений совершенно верными являются следующие операторы:

money = dime;

if (money == quarter)

std::cout << "Денег всего четверть доллара." << std::endl;

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

cout << penny << " " << dime << endl;

на экран будет выведено: 0 2.

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

enum coin {

penny, nickel, dime, quarter = 100, half\_dollar, dollar

};

Вот какие значения появились у этих элементов:

|  |  |
| --- | --- |
| penny | 0 |
| nickel | 1 |
| dime | 2 |
| quarter | 100 |
| half\_dollar | 101 |
| dollar | 102 |

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

/\* Этот код работать не будет \*/ money = dollar;

puts(money);

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

/\* Этот код не правильный \*/ strcpy(money, "dime");

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

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

switch (money) {

case penny: cout << "пенни" << endl; break; case nickel: cout << "никель" << endl; break;

case dime: cout << "монета в 10 центов" << endl; break; case quarter: cout << "четверть доллара" << endl; break; case half\_dollar: cout << "полдоллара" << endl; break; case dollar: cout << "доллар" << endl;

}

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

char name[][20] = { "пенни",

"никель",

"монета в 10 центов", "четверть доллара",

"полдоллара", "доллар"

};

puts(name[money]);

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

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

Практика

1. Создать структуру для описания предметной области вашей курсовой работы.
2. В функции main() создать указатель на структуру из задания 1. Выделить динамическую память под эту структуру.
3. Написать функцию, которая будет считывать данные с клавиатуры и помещать их в структуру. Выделить дополнительную динамическую память в указателе на структуру под новую запись.
4. Написать функцию, которая будет печатать данные из структуры в формате "поле: значение\n".
5. Написать функцию, которая будет печатать данные из структуры в табличном формате.

* Организовать меню для вызова функций из заданий 3-5.

Лабораторная работа №5

**Потоки и файлы**

**Цель работы:** изучить библиотечные функции для работы с файлами. Освоить основные методы обработки файлов.

Теория

Перед тем как начать изучение файловой системы языка С++, необходимо уяснить, в чем разница между потоками и файлами. В системе ввода/вывода С++ для программ поддерживается единый интерфейс, не зависящий от того, к какому конкретному устройству осуществляется доступ. То есть в этой системе между программой и устройством находится нечто более общее, чем само устройство. Такое обобщенное устройство ввода или вывода (устройство более высокого уровня абстракции) называется потоком, в то время как конкретное устройство называется файлом. (Впрочем, файл – тоже понятие абстрактное.) Очень важно понимать, каким образом происходит взаимодействие потоков и файлов.

Потоки

Файловая система языка С++ предназначена для работы с самыми разными устройствами, в том числе терминалами, дисководами и накопителями на магнитной ленте. Даже если какое-то устройство сильно отличается от других, буферизованная файловая система все равно представит его в виде логического устройства, которое называется потоком. Все потоки ведут себя похожим образом. И так как они в основном не зависят от физических устройств, то та же функция, которая выполняет запись в дисковый файл, может ту же операцию выполнять и на другом устройстве, например, на консоли. Потоки бывают текстовые и двоичные. **Текстовый поток** – это последовательность символов. В стандарте С++ считается, что текстовый поток организован в виде строк, каждая из которых заканчивается символом новой строки. Однако в конце последней строки этот символ не является обязательным. В текстовом потоке по требованию базовой среды могут происходить определенные преобразования символов. Например, символ новой строки может быть заменен парой символов — возврата каретки и перевода строки. Поэтому может и не быть однозначного соответствия между символами, которые пишутся (читаются), и теми, которые хранятся во внешнем устройстве.

Кроме того, количество тех символов, которые пишутся (читаются), и тех, которые хранятся во внешнем устройстве, может также не совпадать из-за возможных преобразований.

**Двоичный поток** – это последовательность байтов, которая взаимно однозначно соответствует байтам на внешнем устройстве, причем никакого преобразования символов не происходит. Кроме того, количество тех байтов, которые пишутся (читаются), и тех, которые хранятся на внешнем устройстве, одинаково. Однако в конце двоичного потока может добавляться определяемое приложением количество нулевых байтов. Такие нулевые байты, например, могут использоваться для заполнения свободного места в блоке памяти незначащей информацией, чтобы она в точности заполнила сектор на диске.

Файлы

В языке С++ файлом может быть все что угодно, начиная с дискового файла и заканчивая терминалом или принтером. Поток связывают с определенным файлом, выполняя операцию открытия. Как только файл открыт, можно проводить обмен информацией между ним и программой.

Но не у всех файлов одинаковые возможности. Например, к дисковому файлу прямой доступ возможен, в то время как к некоторым принтерам – нет. Таким образом, мы пришли к одному важному принципу, относящемуся к системе ввода/вывода языка С++: все потоки одинаковы, а файлы – нет.

Если файл может поддерживать запросы на местоположение (указатель текущей позиции), то при открытии такого файла указатель текущей позиции в файле устанавливается в начало. При чтении из файла (или записи в него) каждого символа указатель текущей позиции увеличивается, обеспечивая тем самым продвижение по файлу.

Файл отсоединяется от определенного потока (т.е. разрывается связь между файлом и потоком) с помощью операции закрытия. При закрытии файла, открытого с целью вывода, содержимое (если оно есть) связанного с ним потока записывается на внешнее устройство. Этот процесс, который обычно называют дозаписью потока (или принудительным освобождением содержимого буфера), гарантирует, что никакая информация случайно не останется в буфере диска. Если программа завершает работу нормально, т.е. либо main() возвращает управление операционной системе, либо вызывается exit(), то все файлы закрываются автоматически. В случае аварийного завершения работы программы, например, в случае краха или завершения путем вызова abort(), файлы не закрываются.

Если вы новичок в программировании, то разграничение потоков и файлов может показаться излишним или даже "заумным". Однако надо помнить, что основная цель такого разграничения – это обеспечить единый интерфейс. Для выполнения всех операций ввода/вывода следует использовать только понятия потоков и применять всего лишь одну файловую систему. Ввод или вывод от каждого устройства автоматически преобразуется системой ввода/вывода в легко управляемый поток.

Основы работы с файлами

Для работы с файлами необходимо подключить заголовочный файл <fstream>. 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. Если ввод/вывод на стандартные устройства выполняется с помощью объектов cin и cout, то для организации файлового ввода/вывода достаточно создать собственные объекты, которые можно использовать аналогично операторам cin и cout.

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

* создать объект класса ofstream;
* связать объект класса с файлом, в который будет производиться запись;
* записать строку в файл;
* закрыть файл.

Почему необходимо создавать объект класса ofstream, а не класса ifstream? Потому, что нужно сделать запись в файл, а если бы нужно было считать данные из файла, то создавался бы объект класса ifstream.

// создаём объект для записи в файл

ofstream имя\_объекта; // объект класса ofstream

Назовём объект – fout, вот что получится:

ofstream fout;

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

Уже объект создан, но не связан с файлом, в который нужно записать строку.

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

Через операцию точка получаем доступ к методу класса open(), в круглых скобочках которого указываем имя файла. Указанный файл будет создан в текущей директории с

программой. Если файл с таким именем существует, то существующий файл будет заменен новым. Итак, файл открыт, осталось записать в него нужную строку. Делается это так:

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

Используя операцию передачи в поток совместно с объектом fout строка "Работа с файлами в С++" записывается в файл. Так как больше нет необходимости изменять содержимое файла, его нужно закрыть, то есть отделить объект от файла.

fout.close(); // закрываем файл

Итог – создан файл со строкой «Работа с файлами в С++».

Шаги 1 и 2 можно объединить, то есть в одной строке создать объект и связать его с файлом. Делается это так:

/\* создаём объект класса ofstream \*

\* и связываем его с файлом example.txt \*/ ofstream fout("example.txt ");

Объединим весь код и получим следующую программу.

#include <fstream>

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

/\* создаём объект класса ofstream для записи \*

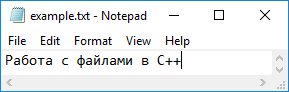
\* и связываем его с файлом example.txt \*/ std::ofstream fout("example.txt ");

fout << "Работа с файлами в С++"; // запись строки в файл fout.close(); // закрываем файл

return 0;

}

Осталось проверить правильность работы программы, а для этого открываем файл

example.txt и смотрим его содержимое.

Для того чтобы прочитать файл понадобится выполнить те же шаги, что и при записи в файл с небольшими изменениями:

* создать объект класса ifstream и связать его с файлом, из которого будет производиться считывание;
* прочитать файл;
* закрыть файл.

#include <fstream> #include <iostream>

using namespace std;

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

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

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

fin >> buff; // считали первое слово из файла cout << buff << endl; // напечатали это слово

fin.getline(buff, 50); // считали строку из файла cout << buff << endl; // напечатали эту строку

fin.close(); // закрываем файл

return 0;

}

В программе показаны два способа чтения из файла, первый – используя операцию передачи в поток, второй – используя функцию getline(). В первом случае считывается только первое слово, а во втором случае считывается строка, длинной 50 символов. Но так как в файле осталось меньше 50 символов, то считываются символы включительно до последнего. Обратите внимание на то, что считывание во второй раз (строка 14) продолжилось, после первого слова, а не с начала, так как первое слово было прочитано в строке 12. Результат работы программы:



Программа сработала правильно, но не всегда так бывает, даже в том случае, если с кодом всё в порядке. Например, в программу передано имя несуществующего файла или в имени допущена ошибка. Что тогда? В этом случае ничего не произойдёт вообще. Файл не будет найден, а значит и прочитать его невозможно. Поэтому компилятор проигнорирует строки, где выполняется работа с файлом. В результате корректно завершится работа программы, но ничего, на экране показано не будет.

Казалось бы, это вполне нормальная реакция на такую ситуацию. Но простому пользователю не будет понятно, в чём дело и почему на экране не появилась строка из файла. Так вот, чтобы всё было предельно понятно в С++ предусмотрена такая функция – is\_open(), которая возвращает целые значения: 1 – если файл был успешно открыт, 0 – если файл открыт не был. Доработаем программу с открытием файла, таким образом, чтобы если файл не открыт, то выводилось соответствующее сообщение.

#include <fstream> #include <iostream>

using namespace std;

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

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

// ввели некорректный тип файла ifstream fin("example.doc");

if (!fin.is\_open()) { // если файл не открыт, сообщить об этом cout << "Файл не может быть открыт!\n";

} else {

fin >> buff; // считали первое слово из файла cout << buff << endl; // напечатали это слово

fin.getline(buff, 50); // считали строку из файла cout << buff << endl; // напечатали эту строку

fin.close(); // закрываем файл

}

return 0;

}

Результат работы программы:



Как видно из примера программа сообщила о невозможности открыть файл. Поэтому, если программа работает с файлами, рекомендуется использовать функцию is\_open() даже если уверены, что файл существует.

Режимы открытия файлов

Режимы открытия файлов устанавливают характер использования файлов. Для установки режима в классе ios\_base предусмотрены константы, которые определяют режим открытия файлов. Их список представлен в следующей таблице.

|  |  |
| --- | --- |
| **Константа** | **Описание** |
| in | Открыть файл для чтения. |
| out | Открыть файл для записи. |
| ate | При открытии переместить указатель в конец файла. |
| app | Открыть файл для записи в конец файла. |
| trunc | Удалить содержимое файла, если он существует. |
| binary | Открытие файла в двоичном (бинарном) режиме. |

Режимы открытия файлов можно устанавливать непосредственно при создании объекта или при вызове функции open().

// открываем файл для добавления информации к концу файла ofstream fout("example.txt", ios\_base::app);

// открываем файл для добавления информации к концу файла fout.open("example.txt", ios\_base::app);

Режимы открытия файлов можно комбинировать с помощью поразрядной логической операции или "|", например: ios\_base::out|ios\_base::trunc – открытие файла для записи, предварительно очистив его.

Объекты класса ofstream, при связке с файлами по умолчанию содержат режимы открытия файлов ios\_base::out|ios\_base::trunc. То есть файл будет создан, если не существует. Если же файл существует, то его содержимое будет удалено, а сам файл будет готов к записи.

Объекты класса ifstream связываясь с файлом, имеют по умолчанию режим открытия файла ios\_base::in – файл открыт только для чтения. Режим открытия файла ещё называют – флаг, для удобочитаемости в дальнейшем будем использовать именно этот термин. В приведённой таблице перечислена лишь часть флагов.

Обратите внимание на то, что флаги ate и app по описанию очень похожи, они оба перемещают указатель в конец файла, но флаг app позволяет производить запись, только в конец файла, а флаг ate просто переставляет флаг в конец файла и не ограничивает места записи.

Разработаем программу, которая, используя операцию sizeof(), будет вычислять характеристики основных типов данных в С++ и записывать их в файл. Характеристики:

* число байт, отводимое под тип данных;
* максимальное значение, которое может хранить определённый тип данных. Запись в файл должна выполняться в таком формате:

|  |  |  |
| --- | --- | --- |
| data type | byte | max value |
| bool | =1 | 255 |
| char | =1 | 255 |
| short int | =2 | 32767 |
| unsigned short int | =2 | 65535 |
| int | =4 | 2147483647 |
| unsigned int | =4 | 4294967295 |
| long int | =4 | 2147483647 |
| unsigned long int | =4 | 4294967295 |
| float | =4 | 2147483647 |
| long float | =8 | 9223372036854775808 |
| double | =8 | 9223372036854775808 |

Для решения поставленной задачи необходимо открыть файл в режиме записи, с предварительным усечением текущей информации файла. Как только файл создан и успешно открыт, используем объект fout. В итоге вся информация о типах данных запишется в файл.

#include <fstream> #include <iostream>

#include <iomanip> // манипуляторы ввода/вывода #include <cmath>

using namespace std;

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

/\* связываем объект с файлом, при этом файл открываем в \*

\* режиме записи, предварительно удаляя все данные из него \*/ ofstream fout("data\_types.txt", ios\_base::out|ios\_base::trunc);

if (!fout.is\_open()) {

cout << "Файл не может быть открыт или создан\n";

return 1;

}

// заголовки столбцов

fout << "data type\t\tbyte\tmax value " << endl;

fout << "bool\t\t\t=" << sizeof(bool)

<< "\t" << fixed << setprecision(0)

<< (pow(2, sizeof(bool) \* 8.0) - 1) << endl;

fout << "char\t\t\t=" << sizeof(char)

<< "\t" << fixed << setprecision(0)

<< (pow(2, sizeof(char) \* 8.0) - 1) << endl;

fout << "short int\t\t=" << sizeof(short int)

<< "\t" << fixed << setprecision(0)

<< (pow(2, sizeof(short int) \* 8.0 - 1) - 1) << endl;

fout << "unsigned short int\t=" << sizeof(unsigned short int)

<< "\t" << fixed << setprecision(0)

<< (pow(2, sizeof(unsigned short int) \* 8.0) - 1) << endl;

fout << "int\t\t\t=" << sizeof(int)

<< "\t" << fixed << setprecision(0)

<< (pow(2, sizeof(int) \* 8.0 - 1) - 1) << endl;

fout << "unsigned int\t\t=" << sizeof(unsigned int)

<< "\t" << fixed << setprecision(0)

<< (pow(2, sizeof(unsigned int) \* 8.0) - 1) << endl;

fout << "long int\t\t=" << sizeof(long int)

<< "\t" << fixed << setprecision(0)

<< (pow(2, sizeof(long int) \* 8.0 - 1) - 1) << endl;

fout << "unsigned long int\t=" << sizeof(unsigned long int)

<< "\t" << fixed << setprecision(0)

<< (pow(2, sizeof(unsigned long int) \* 8.0) - 1) << endl;

fout << "float\t\t\t=" << sizeof(float)

<< "\t" << fixed << setprecision(0)

<< (pow(2, sizeof(float) \* 8.0 - 1) - 1) << endl;

fout << "long float\t\t=" << sizeof(long float)

<< "\t" << fixed << setprecision(0)

<< (pow(2, sizeof(long float) \* 8.0 - 1) - 1) << endl;

fout << "double\t\t\t=" << sizeof(double)

<< "\t" << fixed << setprecision(0)

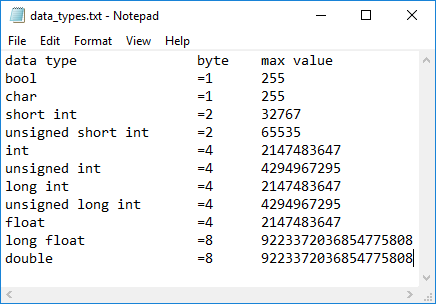
<< (pow(2, sizeof(double) \* 8.0 - 1) - 1) << endl; fout.close(); // закрываем файл

cout << "Данные успешно записаны в файл data\_types.txt\n";

return 0;

}

В конце программы, в красной строке явно закрываем файл, хотя это и не обязательно, но считается хорошим тоном программирования. Стоит отметить, что все функции и манипуляторы, используемые для форматирования стандартного ввода/вывода, актуальны и для файлового ввода/вывода. Поэтому не возникает никаких ошибок, когда оператор cout заменяется объектом fout.

Результат работы, представленной выше программы:



Практика

Дополнить проект, созданный в предыдущей лабораторной работе, следующими функциями и добавить новые пункты в меню программы:

1. Функция new\_DB(), в которой будет создаваться новый файл базы данных и открываться для записи. Функция должна возвращать объект потока ввода для нового файла.
2. Функция remove\_DB(), удаляющая файл по названию. В функции должна быть реализована обработка ошибок.
3. Функция clear(), которая будет очищать динамическую память, выделенную для структур базы данных.
4. Функция save\_DB(), которая будет сохранять данные из динамического массива структур в указанный файл.
5. Функция load\_DB(), которая будет выгружать данные из указанного файла в динамический массив структур.

Лабораторная работа №6

**Модульное программирование**

**Цель работы:** изучить принципы построения модулей на языке С++, а также реализацию программ, включающих несколько модулей.

Теория

Изначальной и, вероятно, все еще самой общей парадигмой программирования является следующая: «Решите, какие процедуры вы желаете; используйте лучшие из алгоритмов, которые можете найти». При этом внимание фокусируется на определении процедуры: выбор алгоритма, необходимого для выполнения желаемых вычислений.

Процедурное программирование использует функции для создания порядка в лабиринте алгоритмов. До сих пор Вы фактически изучали процедурное программирование.

Одной из важнейших составляющих процедурного программирования является модульное программирование. Главную идею модульного программирования можно сформулировать следующим образом.

При решении задач выполняются одни и те же действия. Поэтому разумно реализовать такие операции один раз в виде некоторых модулей и в дальнейшем иметь возможность присоединять эти модули к своей программе.

Как правило, модуль представляет собой совокупность функций и данных, используемых для решения задач определенного класса или реализующих набор операций над некоторым понятием. Например, в одном модуле могут быть определены функции, обеспечивающие ввод- вывод. В другом модуле могут содержаться функции, позволяющие выполнять операции над строками. Третий модуль поддерживает выполнение операций над геометрическими объектами. А еще один, например, обеспечивает доступ к информации в некоторой базе данных. Если при разработке программы мы обнаружим, что нам необходимо выполнять операции над строками и работать с геометрическими объектами, то нет необходимости реализовывать эти подпрограммы заново. Достаточно подключить соответствующие модули к нашей программе. Ну и, разумеется, необходимо знать, а каким образом надо обращаться к этим

модулям, чтобы решить наши задачи.

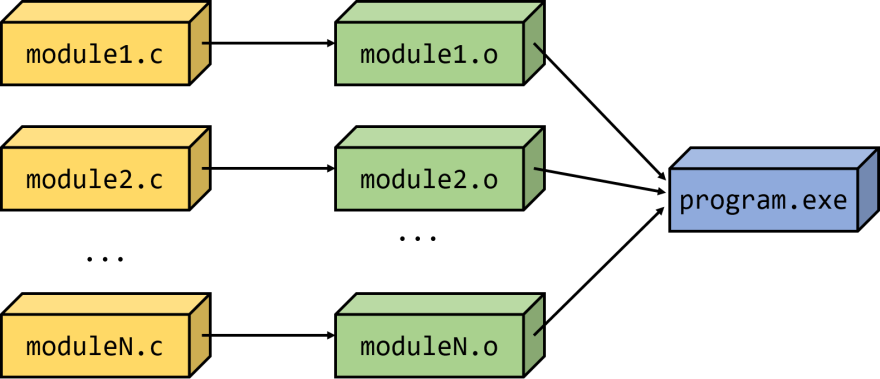
Каждый модуль, входящий в состав программы, решает задачи из своей области. Поэтому модули могут и должны разрабатываться отдельно друг от друга. Это позволяет вести коллективную разработку программной системы. Отдельные программисты или небольшие коллективы независимо от других разрабатывают свою часть программной системы. Кроме того, такой подход обеспечивает возможность модульной отладки, при которой отлаживается не вся программная система в целом, а отдельные ее компоненты. Такая технология обеспечивает быструю локализацию и устранение ошибок, что приводит к ускорению разработки программного продукта.

Основным стержнем, на котором держится модульное программирование, является аппарат подпрограмм (функций). Внутри модулей действия реализуются посредством определения соответствующих подпрограмм. В вызывающей программе содержатся фактически только вызовы необходимых подпрограмм.

Для того чтобы воспользоваться средствами, предоставляемыми модулем, мы должны иметь некую дополнительную информацию по этому модулю. Во-первых, мы должны знать, какие подпрограммы какие действия выполняют. Во-вторых, мы должны знать, какую информацию мы должны передать этим подпрограммам для решения задачи, и каким образом она должна быть передана. Таким образом, важным моментом при использовании модульного программирования является обеспечение взаимодействия между модулями, или как говорят, определение интерфейса. Описание интерфейса, как правило, приводят в специальном

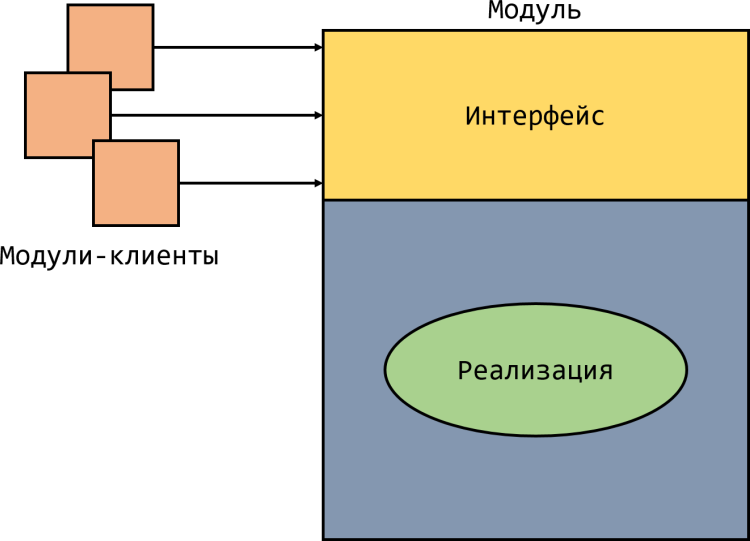
заголовочном файле. Передача информации между подпрограммами (функциями) модулей в большинстве случаев осуществляется через список параметров при вызове соответствующей подпрограммы. Иногда допускается использование глобальных переменных, но каждый такой случай должен быть обоснован.

Для рассматриваемой технологии характерна еще одна задача, а именно задача сборки программы из модулей. Существует несколько способов её решения, некоторые из которых рассматриваются ниже.

Наиболее распространённым способом сборки программы является сборка на уровне объектных файлов. Суть его состоит в том, что для каждого разработанного модуля, посредством компилятора с языка, получают отдельно объектный файл, а затем, используя компоновщик, объединяют их в единый исполняемый образ программы. Такие известные языки программирования как язык ассемблера, Си, C++, Фортран поддерживают данную технологию. Однако, например, не менее известный язык Паскаль в его классическом варианте такой возможности не предусматривает. На рисунке приведена иллюстрация рассмотренного метода.

Характеристики модулей

Важнейшим принципом разработки модуля является принцип информационной закрытости. Этот принцип утверждает, что содержание модулей должно быть скрыто друг от друга. Модуль должен определяться и проектироваться так, чтобы его содержимое (функции и данные) было недоступно тем модулям, которые не нуждаются в такой информации (клиентам). Этот принцип иллюстрируется на следующем рисунке.



Информационная закрытость означает, что все модули независимы и обмениваются только необходимой для работы информацией. Кроме этого доступ к операциям и структурам данных модуля извне его ограничен.

Следование принципу информационной закрытости даёт определённые преимущества. С одной стороны, обеспечивается возможность разработки модулей независимыми разработчиками, что уменьшает время разработки программной системы. Но он даёт преимущества и в малых проектах, когда количество разработчиков невелико, или он вообще один. Это преимущество заключается в лёгкой модификации системы. При использовании этого принципа вероятность распространения ошибок сильно уменьшается, так как многие данные и функции локализованы внутри модуля и скрыты от других частей системы.

В этом случае модуль играет роль «чёрного ящика». Его содержимое (реализация) недоступно другим модулям, а управление им (интерфейс) простое. Изменение реализации модуля никоим образом не влияет на его пользователей.

Для оценки уровня информационной закрытости модуля можно использовать его внутреннюю характеристику – связность. Под связностью модуля понимается мера зависимости его частей. Чем выше связность модуля, тем лучше результат проектирования. Ниже приведены различные типы связности в порядке её уменьшения. При этом первые три свидетельствуют о хорошем качестве модуля, а остальные о недостаточно высоком качестве разработки.

**Функциональная связность.** Части модуля вместе реализуют одну проблемную задачу (операцию). Например, вычисление синуса угла, проверка орфографии, вычисление зарплаты сотрудника, определение места пассажира. Модуль выполняет только то, для чего он предназначен. Поэтому модуль вычисления синуса не должен печатать его значение.

**Информационная связность.** Выходные данные одной части используются как входные данные в другой части модуля. Порядок выполнения действий строго определён и подобен конвейеру. **Коммуникативная связность.** Части модуля связаны по данным, то есть они работают с одним и тем же типом структуры данных. При этом порядок выполнения действий безразличен.

**Процедурная связность.** Этот тип связности соответствует случаю, когда части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения. При этом зависимость по данным между частями модуля отсутствует.

**Временная связность.** Различные части модуля не связаны между собой ни по данным, ни по порядку выполнения, но необходимы в один и тот же период работы системы.

**Логическая связность.** Части модуля объединены по принципу функционального подобия. Например, модуль включает в себя различные функции обработки ошибок. К серьёзным недостаткам таких модулей относится сложное сопряжение с другими модулями и большая вероятность внесения ошибок при необходимости изменений.

**Связность по совпадению.** В модуле отсутствуют явно выраженные внутренние связи вообще.

Для оценки меры взаимозависимости модулей по данным применяется другая характеристика – сцепление. Сцепление является внешней характеристикой модуля, которую желательно уменьшать. Выделяют несколько типов сцепления, которые перечислены ниже в порядке его увеличения.

**Сцепление по данным.** В этом случае функции модуля А вызывают функции модуля B, причём все входные и выходные параметры вызываемого модуля являются простыми элементами данных.

**Сцепление по образцу.** Здесь в качестве параметров используются структуры данных. **Сцепление по управлению.** Один модуль явно управляет функционированием другого модуля с помощью флагов или переключателей, посылая ему управляющие данные.

**Сцепление по внешним ссылкам.** В этом случае оба модуля ссылаются на один и тот же глобальный элемент данных.

**Сцепление по общей области.** Здесь модули разделяют одну и ту же глобальную структуру данных.

**Сцепление по содержанию.** Один модуль прямо ссылается на содержание другого модуля (в обход интерфейсной части вызываемого модуля).

Соглашения по разработке модулей

За время существования такой дисциплины как программирование, в ней были выработаны определенные правила оформления программных модулей. Подавляющее большинство разработчиков программного обеспечения стремятся их соблюдать. Следование этим правилам позволит получить хорошо структурированную, легко читаемую программу. Использование одних и тех же правил оформления модулей позволяет легче и быстрее разобраться в них, значительно уменьшает количество «необязательных» ошибок. Те же, кто эти правила совершенно игнорируют, всех этих преимуществ лишены. На этапе обучения следование приведённым ниже правилам обязательно. Вместе с тем нельзя рассматривать их как некие догмы. Это всего лишь рекомендации. В процессе трудовой деятельности вы вынуждены будете их несколько адаптировать в соответствии с требованиями трудовой организации, компании.

* Проектируйте модуль со связностью не ниже коммуникативной. Он не должен быть просто совокупностью никак не связанных между собой функций и данных.
* Модуль должен охватывать, как можно большую часть задач рассматриваемой области, и в то же время набор доступных подпрограмм следует сделать минимальным и простым в использовании.
* Добейтесь, чтобы сцепление модуля было по данным или по образцу. В противном случае изменение некоторого модуля может вызвать лавинообразные изменения в других модулях.
* При разработке модуля с именем Name, как правило, создается два файла: файл- заголовок Name.h и файл-реализация Name.cpp. В файле-заголовке определяется интерфейс к данному модулю. Он включает в себя определение констант, новых типов данных, описаний функций с обязательным указанием их параметров и описаний глобальных переменных, определенных в модуле и доступных для других частей программы. Если модуль «A» обращается к модулю «B», то в «A» должен быть подключен файл-заголовок модуля «B».

Поскольку файл-заголовок может подключаться несколько раз, то во избежание дублирования используют конструкцию следующего вида:

#ifndef name\_h #define name\_h

// описания функций, глобальных переменных, констант #endif

Файл-реализация Name.cpp содержит определения глобальных и статических переменных модуля, определение новых типов данных, используемых только внутри этого модуля и реализации функций. Очевидно, что в этом модуле должно производиться подключение файла-заголовка.

* Минимизируйте использование глобальных переменных. Наилучшим решением является отказ от их использования. Поскольку глобальные переменные будут доступными из любого модуля, то чрезвычайно трудно контролировать их использование. Настоятельно рекомендуется передавать в функцию все необходимые ей данные в качестве параметров.
* Избегайте слишком больших реализаций функций. Например, можно потребовать, чтобы вся реализация функции умещалась в определённое количество строк (количество строк на экране монитора, в пределах одного листа при печати и т.д.). Кроме этого важно избегать глубоких вложений внутри функции (не более 2 вложенных операторов). Игнорирование этой рекомендации приводит к более трудному восприятию программного текста, реализующего данную функцию, а что еще более важно – к повышению вероятности совершить ошибку при кодировании алгоритма.
* Каждая строка программы должна содержать не более одного оператора. Также старайтесь избегать объявления нескольких переменных на одной строке.
* Программный код пишется для чтения не компилятором, а человеком. Программный код должен сам себя документировать и средства языка C позволяют обеспечить выполнение этого требования. Как минимум, все используемые в программах имена должны быть содержательными. Понять смысл оператора digitCount = digitCount + 2 гораздо проще, чем c = c + 2.
* Хорошая практика программирования состоит в том, чтобы всем явным константам давать символические имена и их использовать в программе. Указанное правило не относится к константам, которые даже в принципе не могут поменяться. Например, индексация элементов в векторе начинается с 0, поэтому определение для этой константы символического имени бессмысленно и только запутывает программный код.
* Используйте отступы для выделения операторов и их блоков внутри условного оператора, оператора цикла и выбора. Следование этой рекомендации позволяет избежать ошибок, связанных, например, с расстановкой скобок { и }. Но еще более важно, что при таком подходе четко прослеживается структура программы.

Пример: математические операции

В качестве примера, иллюстрирующего «хороший стиль» программирования, рассмотрим следующую задачу. Пусть требуется реализовать набор функций для выполнения операций над двумя целыми числами. Первая функция – это сумма двух целых чисел, вторая – результат деления первого числа на второе (возвращает вещественное число), третья возвращает среднее арифметическое из двух чисел, умноженное на число π, а четвертая функция возводит первое число в степень, равную второму числу.

Конечная программа будет состоять из четырех модулей. Основной модуль – main – именно с него начинается работа программы и из этой функции происходит вызов функций из дополнительных модулей. Данный модуль будет представлен файлом main.cpp.

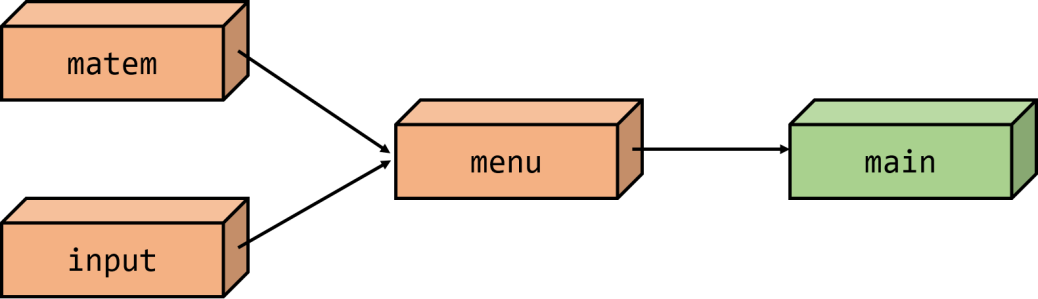
Дополнительный модуль первого уровня – menu. Именно этот модуль будет подключен к main.c и в нем будет реализовано пользовательское меню. Этот модуль будет представлен заголовочным файлом menu.h и файлом-реализацией menu.cpp.

Оставшийся недостающий функционал можно разделить на 2 категории:

* математические операции;
* операции ввода/вывода данных.

Для реализации этих функций создадим два модуля второго уровня:

* модуль matem, состоящий из двух файлов – matem.h и matem.cpp. в котором реализованы математические функции;
* модуль input, состоящий из файлов input.h и input.cpp, в которых будут реализованы функции форматированного ввода исходных данных и вывода результатов.

На рисунке ниже представлена модульная схема разрабатываемой программы.

Далее представлены листинги всех вышеперечисленных файлов.

* 1. main.cpp

#include <clocale> #include <iostream> #include "menu.h"

using namespace std; int main(void) {

setlocale(LC\_ALL, "");

cout << "Вас приветствует модульная программа" << endl; menu();

cout << "Спасибо за внимание!" << endl;

}

* 1. input.h

#ifndef INPUT\_H #define INPUT\_H

#include <iostream> using namespace std;

void input(int\* a, int\* b);

void outputInt(int x, char\* str); void outputFloat(float x, char\* str);

#endif

* 1. input.cpp

#include "input.h"

void input(int \*a, int \*b) { cout << "Введите а и b: "; cin >> \*a >> \*b;

}

void outputInt(int x, char \*str) {

cout << "Результат выполнения функции " << str

<< ": " << x << endl << endl;

}

void outputFloat(float x, char \*str) {

cout << "Результат выполнения функции " << str

<< ": " << x << endl << endl;

}

* 1. matem.h

#ifndef MATEM\_H #define MATEM\_H

#include <iostream> using namespace std; #define PI 3.1415

int sum(int a, int b); float del(int a, int b); float func1(int a, int b); int func2(int a, int b);

#endif

* 1. matem.cpp

#include "matem.h"

int sum(int a, int b) { return (a + b);

}

float del(int a, int b) {

return (float) a / (float) b;

}

float func1(int a, int b) {

return (del(sum(a, b), 2) \* PI);

}

int func2(int a, int b) { int res = 1;

for (int i = 1; i <= b; i++) { res \*= a;

}

return res;

}

* 1. menu.h

#ifndef MENU\_H #define MENU\_H

using namespace std; void menu(void); #endif

* 1. menu.cpp

#include "matem.h" #include "input.h"

void menu(void) {

int q = 9; while (q > 0) {

int a = 0, b = 0;

cout << "Выберите функцию:" << endl

<< "\t1: (a+b)" << endl

<< "\t2: (a/b)" << endl

<< "\t3: ((a+b)/2)\*pi" << endl

<< "\t4: a^b" << endl

<< "\t0: выход" << endl; cin >> q;

switch (q) {

case 1: {

input(&a, &b); outputInt(sum(a, b), "sum"); break;

}

case 2: {

input(&a, &b); outputFloat(del(a, b), "del"); break;

}

case 3: {

input(&a, &b); outputFloat(func1(a, b), "func1"); break;

}

case 4: {

input(&a, &b); outputInt(func2(a, b), "func2"); break;

}

default: {

cout << "Неверный ввод!" << endl;

}

}

}

return;

}

Практика

* 1. Создайте новый модуль для меню.
  2. Создайте новый модуль для функций ввода/вывода данных.
  3. Создайте новый модуль для функций работы с файлами.

1. Перенесите пользовательские функции, реализованные в предыдущих лабораторных работах, в соответствующие модули.
2. Оформите заголовочные файлы для всех модулей.
3. Подключите модули друг к другу в соответствии с тем, каким образом они будут взаимодействовать.
4. Нарисуйте модульную схему вашего приложения.

Лабораторная работа №7

**Цель работы:** изучить методы организации списочных структур в динамической памяти. Реализовать алгоритмы добавления, удаления и сортировки списков.

Теория

**Введение в абстрактные структуры**

Абстрактные структуры данных предназначены для удобного хранения и доступа к информации. Они предоставляют удобный интерфейс для типичных операций с хранимыми объектами, скрывая детали реализации от пользователя. Конечно, это весьма удобно и позволяет добиться большей модульности программы. Абстрактные структуры данных иногда делят на две части: интерфейс, набор операций над объектами, который называют АТД (абстрактный тип данных) и реализацию.

Каждый уже имел дело с простыми абстрактными структурами данных - например, когда оперировал с числами. Языки программирования высокого уровня (Паскаль, Си) предоставляют удобный интерфейс для чисел: операции +, \*, = и тому подобные, но при этом скрывают саму реализацию этих операций, машинные команды.

Статические структуры данных

Статические структуры относятся к разряду не примитивных структур, которые, фактически, представляют собой структурированное множество примитивных, базовых, структур. Например, вектор может быть представлен упорядоченным множеством чисел. Поскольку по определению статические структуры отличаются отсутствием изменчивости, память для них выделяется один раз и ее объем остается неизменным до уничтожения структуры. Слово 'статический' относится скорее к реализации структуры, нежели к АТД.

Простейшая статическая структура данных – массив, где обращение к элементу происходит через его номер.

Слово "массив" употребляется в различных контекстах:

* как АТД, то есть множество с операциями:
  + получить элемент с номером N;
  + записать элемент с номером N.
* и как физическая структура, реализованная в виде непрерывной области памяти.

В случае реализации массива через такую структуру, номер соответствует смещению от начала области.

Плюсов у массива (здесь реализация) всего два, но зато больших:

* доступ за константное время к любому элементу;
* память тратится только на данные\*.

\* здесь и далее будут опущены константные затраты памяти операционной системой, возникающие при реализации структуры. В частности, в большинстве ОС, при динамическом выделении памяти под массив, в начале соответствующей области памяти ставится специальная метка.

Минус – один, но тоже большой: статичность, неизменность структуры.

Одномерный массив иногда называют вектором.

Динамические структуры данных

Динамические структуры по определению характеризуются отсутствием физической смежности элементов структуры в памяти непостоянством и непредсказуемостью размера (числа элементов) структуры в процессе ее обработки.

Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Для установления связи между элементами динамической структуры используются указатели, через которые устанавливаются явные связи между элементами. Такое представление данных в памяти называется связным. Элемент динамической структуры состоит из двух полей:

* 1. информационного поля или поля данных, в котором содержатся те данные, ради которых и создается структура; в общем случае информационное поле само является интегрированной структурой – вектором, массивом, другой динамической структурой и тому подобным;
  2. поле связок, в котором содержатся один или несколько указателей, связывающий данный элемент с другими элементами структуры.

Когда связное представление данных используется для решения прикладной задачи, для конечного пользователя "видимым" делается только содержимое информационного поля, а поле связок используется только программистом- разработчиком.

Достоинства связного представления данных:

* возможность обеспечения значительной изменчивости структур;
* размер структуры ограничивается только доступным объемом машинной памяти;
* при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;
* большая гибкость структуры.

Вместе с тем связное представление не лишено и недостатков:

* на поля связок расходуется дополнительная память;
* доступ к элементам связной структуры может быть менее эффективным по времени.

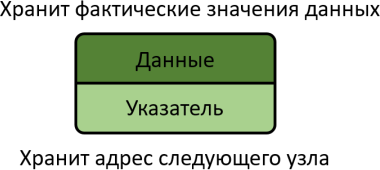
Последний недостаток является наиболее серьезным и именно им ограничивается применимость связного представления данных. Если в смежном представлении данных для вычисления адреса любого элемента нам во всех случаях достаточно было номера элемента и информации, содержащейся в дескрипторе структуры, то для связного представления адрес элемента не может быть вычислен из исходных данных. Дескриптор связной структуры содержит один или несколько указателей, позволяющих войти в структуру, далее поиск требуемого элемента выполняется следованием по цепочке указателей от элемента к элементу. Поэтому связное представление практически никогда не применяется в задачах, где логическая структура данных имеет вид вектора или массива

- с доступом по номеру элемента, но часто применяется в задачах, где логическая структура требует другой исходной информации доступа (таблицы, списки, деревья и т.д.).

Списки

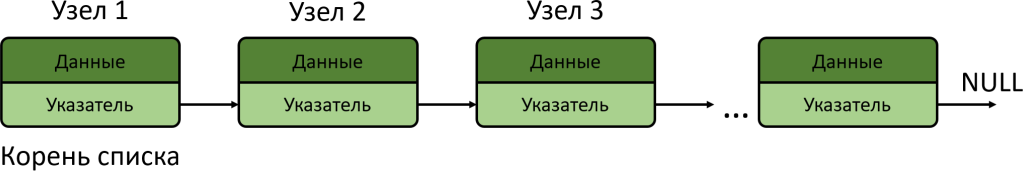
Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения, исключения.

Существует два типа связанных списков: односвязный список и двусвязный список. В односвязном списке каждый элемент содержит некоторые данные и ссылку на следующий элемент. С другой стороны, каждый узел в двусвязном списке содержит некоторые данные, ссылку на следующий узел и ссылку на предыдущий узел.

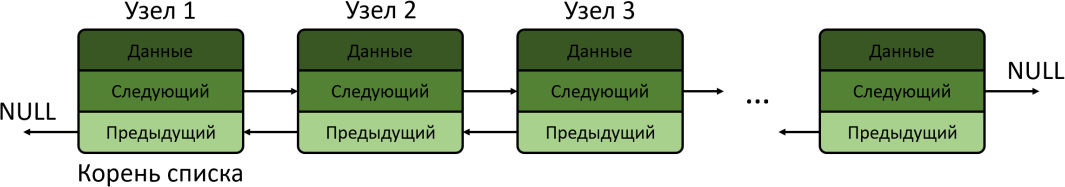
Элементы связанного списка называются узлами. У узла есть два поля, такие как данные и указатель на следующий элемент списка. Поле данных содержит данные, хранящиеся в этом конкретном узле. Это не может быть только одна переменная. Может быть много переменных, представляющих раздел данных узла. Следующее поле

содержит адрес следующего узла. Таким образом, это место, где установлена связь между узлами.

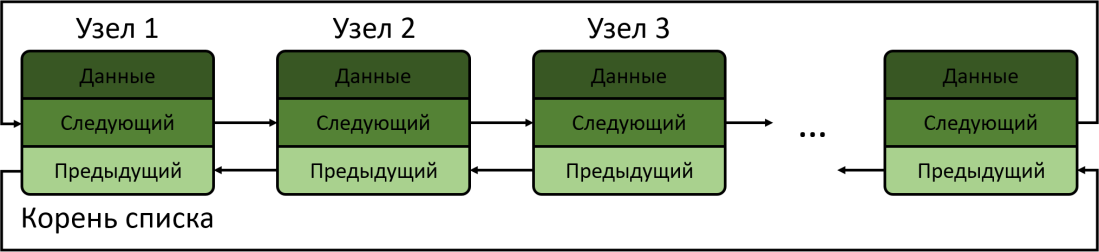
На рисунке ниже приведена структура односвязного списка. Каждый список должен иметь особый элемент, называемый указателем начала списка или головой (корнем) списка, который обычно по формату отличен от остальных элементов. В поле указателя последнего элемента списка находится специальный признак NULL, свидетельствующий о конце списка.



Двусвязный список характеризуется наличием пары указателей в каждом элементе (на предыдущий элемент и на следующий):



Очевидный плюс тут в том, что от данного элемента структуры мы можем пойти в обе стороны. Таким образом упрощаются многие операции. Однако на указатели тратится дополнительная память.

Разновидностью рассмотренных видов линейных списков является кольцевой список, который может быть организован на основе как односвязного, так и двухсвязного списков. При этом в односвязном списке указатель последнего элемента должен указывать на первый элемент; в двухсвязном списке в первом и последнем элементах соответствующие указатели переопределяются, как показано на следующем рисунке.

При работе с такими списками несколько упрощаются некоторые процедуры. Однако, при просмотре такого списка следует принять некоторые меры предосторожности, чтобы не попасть в бесконечный цикл.

Описываемые ниже АТД могут быть организованы на базе массива: выделяется место под N элементов разом, а затем описываются операции над данным типом данных в терминах операций над элементами массива.

списка: память выделяется и освобождается по мере необходимости.

Первый вариант быстрее, но **лишь второй истинно динамический**. Соответственно, в различных приложениях может быть предпочтителен первый (размер структуры известен и небольшой) или второй (размер заранее неизвестен). Мы будем рассматривать преимущественно динамические решения.

Линейный список

Линейный список – это структура данных, состоящая из элементов одного типа, связанных между собой последовательно посредством указателей.

Вообще говоря, чаще всего списки используются с фиктивным элементом, который является первым в последовательности. Использование фиктивного элемента позволяет упростить реализацию некоторых операций. Добавлю, что не стоит пугаться таких элементов, это всего-навсего элемент списка, поле данных которого не используется.

Над линейный списком определены следующие операции:

* инициализация;
* включение/исключение элемента;
* переход в начало/конец списка;
* переход к следующему элементу;
* уничтожение списка.

Реализация односвязного линейного списка (ОЛС)

При реализации ОЛС в статической памяти располагают дескриптор, состоящий из двух полей:

1. указатель на фиктивный элемент;
2. указатель на текущий элемент.

/\* Базовый тип списка \*/ typedef int listOneBaseType;

/\* Структура элемента списка \*/ struct element {

listOneBaseType data; // данные

element \*link; // указатель на следующий элемент списка

};

/\* Дескриптор списка \*/ struct ListOne{

element \*start; // указатель на фиктивный элемент element \*ptr; // указатель на рабочий элемент

};

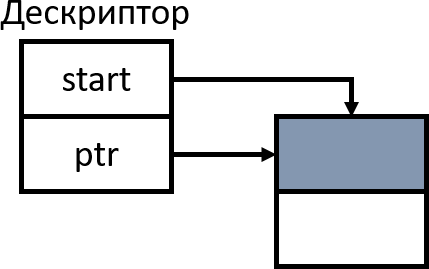
Хочу обратить ваше внимание на тот факт, что в любой момент времени нам известен лишь дескриптор и, соответственно, информация в нем, поэтому все операции над списком выполняются непосредственно через него.

**ВНИМАНИЕ!** При использовании фиктивного элемента, все операции нужно выполнять после того элемента, на который указывает ptr.

Инициализация списка

Инициализация происходит в три шага:

1. Выделение памяти для фиктивного элемента.
2. В дескрипторе указателю start присвоить блок с только что выделенной памятью.
3. Указателю ptr присвоить start.

При инициализации происходит создание списка, но не заполнение его элементами. Создание списка подразумевает создание фиктивного элемента. Так как его поле данных не используется, то список остается по-прежнему пуст. Когда память выделена, остается лишь присвоить ссылки указателям.

Код инициализации таков:

/\* Инициализация списка \*/ void initListOne(ListOne \*L) {

// выделение памяти под фиктивный элемент

L->start = (element \*)malloc(sizeof(element));

// если памяти не хватило if (!L->start) {

listOneError = listOneNoMem; return;

}

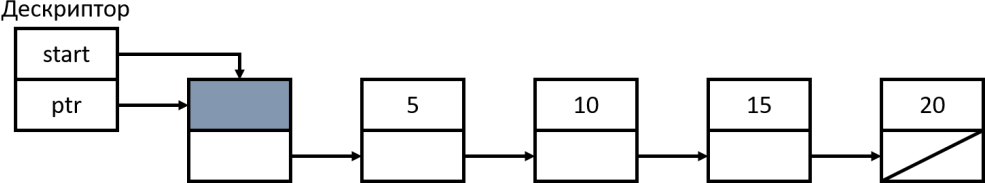
// ptr указывает на то, что и start - на фиктивный L->ptr = L->start;

// нет ссылки на следующий элемент L->ptr->link = NULL;

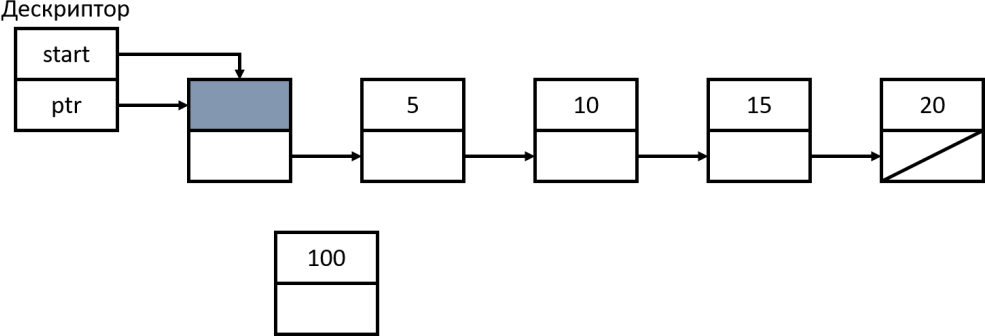
}

Включение элемента

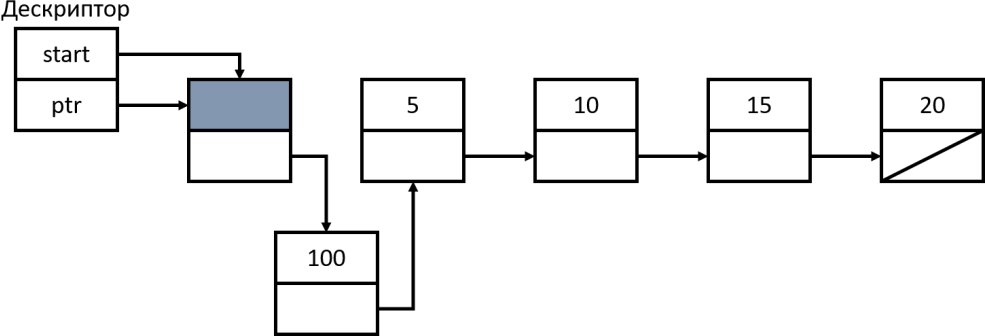
Включение элемента порой сложно понять, поэтому этот вопрос будет рассмотрен максимально подробно. Рассмотрим список из нескольких элементов:



Так как ptr указывает на фиктивный элемент, то новый элемент нужно вставить после него, но перед элементом со значением 5. Прежде чем вставить, нужно выделить память под новый элемент и занести информацию в него. Допустим, мы вставляем элемент с данными – 100:



Теперь нужно вставить новый элемент между двумя элементами списка, то есть изменить ссылки, находящиеся в полях ptr:



Теперь рассмотрим код операции включения:

/\* Включение элемента в список \*/

void putListOne(ListOne \*L, listOneBaseType E) {

// выделение памяти под новый элемент

element\* temp = (element \*)malloc(sizeof(element));

// если памяти не хватило if (!L->start) {

ListOneError = listOneNoMem; return;

}

// внесение информации в новый элемент temp->data = E;

// новый элемент указывает на следующий temp->link = L->ptr->link;

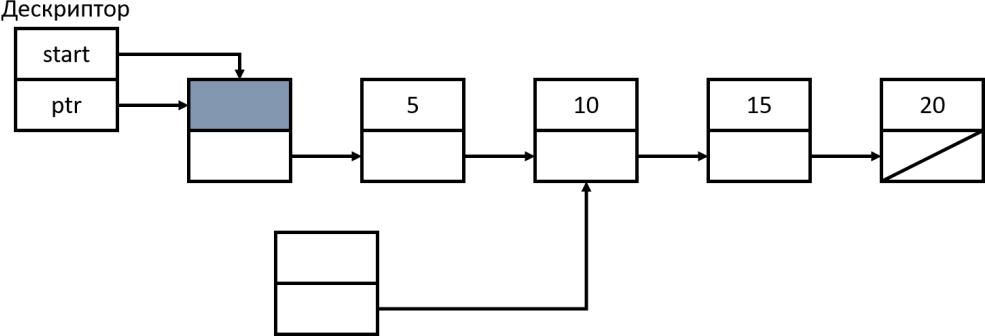
// текущий элемент указывает на новый L->ptr->link = temp;

}

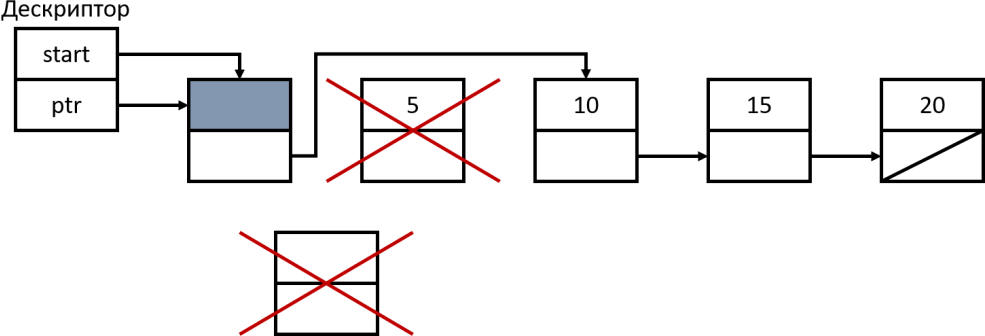
Исключение элемента

Пусть нам дан тот же список, что и в примере с включением элемента.

Так как ptr указывает на фиктивный – исключаем первый. Здесь перемычка ссылок немного сложнее, чем в предыдущем примере. Нужно создать временный элемент так, чтобы он был равен исключаемому, т.е. имел один и тот же с ним адрес. Это нужно для того, чтобы не потерять тот элемент, на который исключаемый ссылается. Данный процесс изображен ниже.



Теперь необходимо произвести перемычку указателей и удаление временного элемента:



Вот и вся операция исключения. Ниже представлен код:

/\* Исключение элемента из списка \*/

void getListOne(ListOne \*L, listOneBaseType \*E) {

// если список пуст

if (isEmptyListOne(L)) return;

// Если рабочий указатель указывает на последний элемент if (!L->ptr->link) {

listOneError = listOneEnd; return;

}

// временный элемент

element \*pntr = L->ptr->link;

// текущий элемент ссылается на тот, на который ссылается исключаемый L->ptr->link = pntr->link;

// считывание информации из элемента списка

\*E = pntr->data;

// временный элемент ни на что не ссылается pntr->link = NULL;

// удаление временного элемента free((void \*)pntr);

}

Остальные операции нет смысла так подробно расписывать, так как они просты в реализации.

Реализация остальных функций линейного списка

Для начала, подключим к проекту необходимые библиотеки и объявим глобальные переменные, которые помогут организовать корректную работу итогового приложения.

#include <iostream> #include <cstdlib> #include <clocale>

using namespace std;

// переменная ошибок int listOneError;

// описание исключительных ситуаций const int listOneOK = 0;

const int listOneEmpty = 1; const int listOneNoMem = 2; const int listOneEnd = 3;

Нам необходимо реализовать следующие пользовательские функции для работы со списком:

// инициализация списка

void initListOne(ListOne \*L);

// включение элемента в список

void putListOne(ListOne \*L, listOneBaseType E);

// исключение элемента из списка

void getListOne(ListOne \*L, listOneBaseType \*E);

// печать всех элементов списка с начала до конца void printListOne(ListOne \*L);

// сдвиг рабочего указателя

void movePtrListOne(ListOne \*L);

// установка рабочего указателя в начало списка void beginPtrListOne(ListOne \*L);

// установка рабочего указателя в конец списка void endPtrListOne(ListOne \*L);

// очистка списка

void doneListOne(ListOne \*L);

// предикат: является ли список пустым int isEmptyListOne(ListOne \*L);

Функции initListOne(), putListOne() и getListOne() были описаны выше, поэтому их листинг приводить повторно не будем, а вот реализация всех остальных функций представлена ниже.

/\* Печать всех элементов списка с начала до конца \*/ void printListOne(ListOne \*L) {

// дополнительный элемент для независимого

// прохода от начала до конца списка element \*tmp;

// ссылка на первый элемент tmp = L->start->link;

// пока не конец списка while (tmp != NULL) {

cout << "\t" << tmp->data; // печать текущего элемента tmp = tmp->link; // независимый переход на следующий элемент

}

free(tmp); // удаление дополнительного элемента

}

/\* Сдвиг рабочего указателя \*/ void movePtrListOne(ListOne \*L) {

// если рабочий указатель указывает на последний элемент if (!L->ptr->link) {

listOneError = listOneEnd; return;

}

// инчае смещение указателя L->ptr = L->ptr->link;

}

/\* Установка рабочего указателя в начало списка \*/ void beginPtrListOne(ListOne \*L) {

L->ptr = L->start;

}

/\* Установка рабочего указателя в конец списка \*/ void endPtrListOne(ListOne \*L) {

while (L->ptr->link)

movePtrListOne(L);

}

/\* Очистка списка \*/

void doneListOne(ListOne \*L) {

beginPtrListOne(L); // установка рабочего указателя в начало listOneBaseType temp; // создание временной переменной

while (!isEmptyListOne(L)) // пока список не пуст getListOne(L, &temp); // извлечь элемент

free((void \*)L->ptr); // удалить фиктивный элемент

}

/\* Предикат: является ли список пустым \*/ int isEmptyListOne(ListOne \*L) {

if (!L->start->link) { listOneError = listOneEmpty; return 1;

}

return 0;

}

Тестирование функций линейного списка

Проверим работу описанных выше функций, написав следующее содержимое функции main():

int main(void) {

ListOne \*List = (ListOne \*)malloc(0); // создание списка initListOne(List); // инициализация

// добавление элементов putListOne(List, 1);

putListOne(List, 2);

putListOne(List, 3);

putListOne(List, 4);

putListOne(List, 5);

cout << "Список после добавления элементов:" << endl; printListOne(List); // печать списка

int i = 0;

movePtrListOne(List); // смещение на один элемент getListOne(List, &i); // удаление текущего элемента

cout << endl << "После смещения удалён элемент: " << i << endl; printListOne(List);

movePtrListOne(List); // смещение на movePtrListOne(List); // два элемента getListOne(List, &i); // удаление текущего элемента

cout << endl << "После ещё двух смещений удалён элемент: " << i << endl; printListOne(List);

beginPtrListOne(List); // смещение на стартовый фиктивный элемент cout << endl << "Первый элемент: " << List->ptr->link->data << endl; printListOne(List);

endPtrListOne(List); // смещение на последний элемент

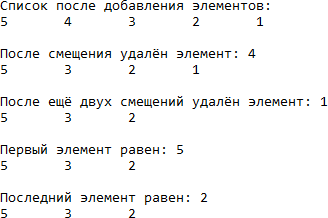
cout << endl << "Последний элемент: " << List->ptr->data << endl; printListOne(List);

doneListOne(List); // очистка списка

return 0;

}

Результат тестирования программы:



Осталось только добавить, что перечисленные выше операции – далеко не все, которые можно реализовать для списка. Вы сами можете создавать новые, исходя из поставленных перед вами задач.

Двусвязный список

Двусвязный список – это структура данных, которая состоит из узлов, которые хранят данные, указатели на предыдущий и следующий элементы.

Так как каждый элемент двусвязного линейного списка (ДЛС) содержит, помимо поля с данными, два указателя, то отсюда вытекает то, что для ДЛС требуется больше памяти.

Использование двух указателей дает несколько преимуществ. Самым главным, из которых является то, что перемещаться по списку можно в двух направлениях, благодаря чему упрощаются некоторые операции (по сравнению с ОЛС) и добавляются новые.

Операции над ДЛС:

* инициализация;
* включение до рабочего указателя;
* включение после рабочего указателя;
* исключение до рабочего указателя;
* исключение после рабочего указателя;
* сдвиг рабочего указателя назад (к предыдущему элементу);
* сдвиг рабочего указателя вперед (к следующему элементу);
* установка рабочего указателя в начало списка;
* установка рабочего указателя в конец списка;
* проверка пустоты списка;
* удаление списка.

Реализация двусвязного списка

Как и с односвязным списком, в статической памяти будет располагаться дескриптор. Этот дескриптор будет в точности таким же, как и у ОЛС. Единственное отличие

– два указателя.

/\* Базовый тип списка \*/ typedef int listTwoBaseType;

/\* Структура элемента списка \*/ struct element {

listTwoBaseType data; // данные

element \*predLink; // указатель на предыдущий элемент списка element \*nextLink; // указатель на следующий элемент списка

};

/\* Дескриптор списка \*/ struct ListTwo {

element \*start; // указатель на левый фиктивный элемент element \*end; // указатель на правый фиктивный элемент element \*ptr; // указатель на рабочий элемент

};

Инициализация двусвязного списка

Суть инициализации состоит в том, чтобы создать фиктивные элементы, то есть выделить для них память. Интерес вызывает то, что эти указатели должны указывать друг на друга.

/\* Инициализация списка \*/ void initListTwo(ListTwo \*L) {

// выделение памяти под левый фиктивный элемент L->start = (element \*)malloc(sizeof(element));

// если памяти не хватило

if (!L->start) { listTwoError = listTwoNoMem; return; }

// выделение памяти для правого фиктивного элемента L->end = (element \*)malloc(sizeof(element));

// если памяти не достаточно if (!L->end) {

listTwoError = listTwoNoMem;

free((void \*) L->start); // удаление левого фиктивного элемента L->start = NULL;

return;

}

// элемент, следующий за левым фиктивным - правый фиктивный элемент L->start->nextLink = L->end;

// элемента, предшествующего левому фиктивному элементу нет L->start->predLink = NULL;

// элемента, следующего за правым фиктивным элементом нет L->end->nextLink = NULL;

// элемент, предшествующий правому фиктивному - левый фиктивный элемент L->end->predLink = L->start;

// рабочий указатель и левый фиктивный элемент указывают на одно и то же L->ptr = L->start;

listTwoError = listTwoOK; // всё в порядке

}

Как видите, суть такая же, как это было с односвязным списком. Остальные операции реализуются по такому же принципу, как и соответствующие операции ОЛС. Единственное, о чем нужно заботиться – это о втором указателе.

Функции для работы с двусвязным списком

Для начала подключим библиотеки, объявим служебные (дополнительные) переменные и прототипы функций.

#include <iostream> #include <cstdlib> #include <clocale>

using namespace std;

/\* Переменная ошибок \*/ int listTwoError;

/\* Описание исключительных ситуаций \*/ const int listTwoOK = 0; // всё впорядке

const int listTwoEmpty = 1; // список пуст

const int listTwoNoMem = 2; // недостаточно памяти const int listTwoEnd = 3; // указатель в конце

const int listTwoBegin = 4; // указатель в начале

/\* Функции работы со списком \*/

// инициализация списка

void initListTwo(ListTwo \*L);

// включение элемента до рабочего указателя void predPut(ListTwo \*L, listTwoBaseType E);

// включение элемента после рабочего указателя void postPut(ListTwo \*L, listTwoBaseType E);

// исключение элемента до рабочего указателя void predGet(ListTwo \*L, listTwoBaseType \*E);

// исключение элемента после рабочего указателя void postGet(ListTwo \*L, listTwoBaseType \*E);

// печать всех элементов списка с начала до конца void printListTwo(ListTwo \*L);

// сдвиг рабочего указателя назад void movePtrListTwoLeft(ListTwo \*L);

// сдвиг рабочего указателя вперёд void movePtrListTwoRight(ListTwo \*L);

// установка рабочего указателя в начало списка void beginPtrListTwo(ListTwo \*L);

// установка рабочего указателя в конец списка void endPtrListTwo(ListTwo \*L);

// очистка списка

void doneListTwo(ListTwo \*L);

// предикат: является ли список пустым int isEmptyListTwo(ListTwo \*L);

Функции для вставки элементов:

/\* Включение элемента в список \*/

void predPut(ListTwo \*L, listTwoBaseType E) {

// если рабочий указатель установлен на начало if (L->ptr = L->start) {

listTwoError = listTwoBegin; return;

}

// выделение памяти под новый элемент

element \*pntr = (element \*)malloc(sizeof(element));

// если памяти не достаточно if (!pntr) {

listTwoError = listTwoNoMem; return;

}

// запись информации в новый элемент pntr->data = E;

// новый элемент указывает слева на предыдущий pntr->predLink = L->ptr->predLink;

// новый элемент указывает справа на текущий pntr->nextLink = L->ptr;

// предыдущий элемент указывает справа на новый L->ptr->predLink->nextLink = pntr;

// текущий элемент указывает слева на новый L->ptr->predLink = pntr;

}

/\* Включение после рабочего указателя \*/ void postPut(ListTwo \*L, listTwoBaseType E) {

// если рабочий указатель установлен в конец if (L->ptr == L->end) {

listTwoError = listTwoEnd; return;

}

// выделение памяти под новый элемент

element \*pntr = (element \*)malloc(sizeof(element));

// если памяти недостаточно if (!pntr) {

listTwoError = listTwoNoMem; return;

}

// запись информации в новый элемент pntr->data = E;

// новый элемент указывает справа на следующий pntr->nextLink = L->ptr->nextLink;

// новый элемент указывает слева на текущий pntr->predLink = L->ptr;

// следующий элемент указывает слева на новый L->ptr->nextLink->predLink = pntr;

// текущий элемент указывает справа на новый L->ptr->nextLink = pntr;

}

Функции для удаления элементов:

/\* Исключение элемента до рабочего указателя \*/ void predGet(ListTwo \*L, listTwoBaseType \*E) {

// если список пуст

if (isEmptyListTwo(L)) return;

// если рабочий указатель указывает на начало (левый фиктивный элемент) if (L->ptr == L->start) {

listTwoError = listTwoBegin; return;

}

// запись информации в переменную

\*E = L->ptr->predLink->data;

// временный элемент, указывающий на удаляемый element \*pntr = L->ptr->predLink;

// текущий элемент указывает на предшествующий удалемому L->ptr->predLink = pntr->predLink;

// предшествующий удаляемому элемент указывает на текущий pntr->predLink->nextLink = L->ptr;

// удаление временного элемента free((void \*)pntr);

}

/\* Исключение элемента после рабочего указателя \*/ void postGet(ListTwo \*L, listTwoBaseType \*E) {

// если список пуст

if (isEmptyListTwo(L)) return;

// если рабочий указатель указывает на конец (правый фиктивный элемент) if (L->ptr == L->end) {

listTwoError = listTwoEnd; return;

}

// запись информации в переменную

\*E = L->ptr->nextLink->data;

// временный элемент, указывающий на удаляемый element \*pntr = L->ptr->nextLink;

// текущий элемент указывает на предшествующий удалемому L->ptr->nextLink = pntr->nextLink;

// предшествующий удаляемому элемент указывает на текущий pntr->nextLink->predLink = L->ptr;

// удаление временного элемента free((void \*)pntr);

}

Функции сдвига рабочего указателя (ptr) вправо и влево относительно текущей позиции, а также в начало и в конец списка.

/\* Установка рабочего указателя в начало списка \*/ void beginPtrListTwo(ListTwo \*L) {

while (L->ptr != L->start) movePtrListTwoLeft(L);

}

/\* Установка рабочего указателя в конец списка \*/ void endPtrListTwo(ListTwo \*L) {

while (L->ptr != L->end) movePtrListTwoRight(L);

}

/\* Сдвиг рабочего указателя назад \*/ void movePtrListTwoLeft(ListTwo \*L) {

// если рабочий указатель указывает на начало

if (L->ptr == L->start) { listTwoError = listTwoBegin; return; }

L->ptr = L->ptr->predLink; // иначе смещение указателя

}

/\* Сдвиг рабочего указателя вперёд \*/ void movePtrListTwoRight(ListTwo \*L) {

// если рабочий указатель указывает на конец списка

if (L->ptr == L->end) { listTwoError = listTwoEnd; return; }

L->ptr = L->ptr->nextLink; // иначе смещение указателя

}

Дополнительные функции:

/\* Очистка списка \*/

void doneListTwo(ListTwo \*L) {

beginPtrListTwo(L); // установка рабочего указателя в начало listTwoBaseType temp; // создание временной переменной

while (!isEmptyListTwo(L)) // пока список не пуст postGet(L, &temp); // извлечь элемент

free((void \*)L->end); // удалить фиктивный элемент

// L->ptr ссылается на L->start, поэтому присваиваем ему значение NULL L->ptr = NULL;

free((void \*)L->start);

}

/\* Предикат: является ли список пустым \*/ int isEmptyListTwo(ListTwo \*L) {

if (L->start->nextLink == L->end) { listTwoError = listTwoEmpty; return 1;

}

return 0;

}

/\* Печать всех элементов списка с начала до конца \*/ void printListTwo(ListTwo \*L) {

// дополнительный элемент для прохода от начала до конца списка element \*tmp;

tmp = L->start->nextLink; // ссылка на первый элемент

// пока не конец списка while (tmp != L->end) {

cout << tmp->data << " "; // печать текущего элемента

tmp = tmp->nextLink; // переход на следующий элемент

}

cout << endl;

}

Тестирование функций для работы с двусвязным списком

Протестируем правильность работы описанных выше функций, написав функцию

main() следующего содержания:

int main(void) {

setlocale(LC\_ALL, "Russian");

ListTwo \*L = (ListTwo \*)malloc(0); // создание списка initListTwo(L); // инициализцаия списка

postPut(L, 1); // добавление элементов postPut(L, 2); // после текущего

cout << "Список после добавления элементов 1 и 2: "; printListTwo(L); // печать списка от начала до конца

endPtrListTwo(L); // перемещение текущего указателя в конец predPut(L, 3); // добавление элементов

predPut(L, 4); // перед текущим

cout << "Список после добавления элементов 3 и 4: "; printListTwo(L); // печать списка от начала до конца

beginPtrListTwo(L); // перемещение текущего указателя в начало postPut(L, 5); // добавление элементов

postPut(L, 6); // после текущего

cout << "Список после добавления элементов 5 и 6: "; printListTwo(L); // печать списка от начала до конца

movePtrListTwoRight(L); // перемещение указателя movePtrListTwoRight(L); // на три элемента movePtrListTwoRight(L); // вправо

int i = 0;

postGet(L, &i); // удаление элемента за текущим cout << endl << "Удалён 4-ый элемент: " << i << endl; cout << "Получившийся список: ";

printListTwo(L);

movePtrListTwoLeft(L); // перемещение на один элемент влево predGet(L, &i); // удаление элемента перед текущим cout << endl << "Удалён 1-ый элемент: " << i << endl;

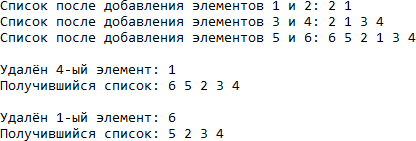
cout << "Получившийся список: "; printListTwo(L);

doneListTwo(L);

return 0;

}

Результат тестирования программы:



Практика

Создайте динамический список (**НЕ** на основе массива) указанного типа для хранения данных, указанных в варианте задания. Напишите функции для работы со списком:

**Задание 1.** Функции для создания нового списка и его удаления.

**Задание 2.** Функции добавления элемента в начало и после указанного элемента.

**Задание 3.** Функция удаления элемента.

**Задание 4.** Функция печати всех элементов списка.

**Дополнительное задание \*** Функция перестановки двух элементов списка (например, если на вход функции подается k-тый и g-тый элементы, то после срабатывания функции в списке на позиции k-того элемента окажется g-тый, а на позиции g-того – k-тый).

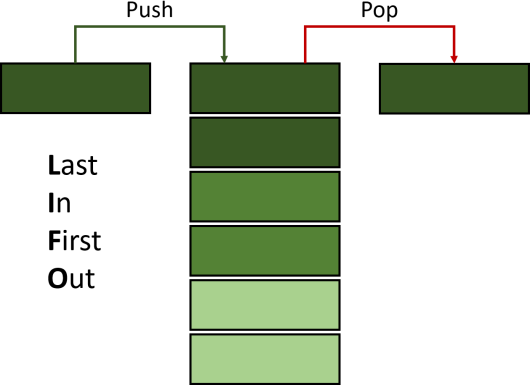
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Вариант** | **Задание** | **Вариант** | **Задание** | **Вариант** | **Задание** |
| 1.1 | линейный  char[20] | 2.1 | линейный  int | 3.1 | двусвязный  double |
| 1.2 | кольцевой  char[20] | 2.2 | кольцевой  int | 3.2 | кольцевой  double |
| 1.3 | двусвязный  int | 2.3 | двусвязный  int | 3.3 | линейный  int |
| 1.4 | линейный float | 2.4 | линейный char[20] | 3.4 | двусвязный int |
| 1.5 | кольцевой  double | 2.5 | кольцевой  char[20] | 3.5 | кольцевой  int |
| 1.6 | двусвязный  int | 2.6 | двусвязный  char[20] | 3.6 | линейный  char[15] |
| 1.7 | линейный double | 2.7 | линейный float | 3.7 | кольцевой char[15] |
| 1.8 | кольцевой  int | 2.8 | кольцевой  float | 3.8 | двусвязный  char[15] |
| 1.9 | двусвязный  float | 2.9 | двусвязный  float | 3.9 | линейный  float |
| 1.10 | линейный  int | 2.10 | линейный  double | 3.10 | двусвязный  float |
| 1.11 | кольцево  float | 2.11 | двусвязный  int | 3.11 | кольцевой  float |
| 1.12 | двусвязный char[15] | 2.12 | кольцевой float | 3.12 | линейный float |
| 1.13 | кольцевой  int | 2.13 | двусвязный  char[15] | 3.13 | двусвязный  int |

Лабораторная работа №8

**Динамические структуры данных: очереди, стеки, деки**

**Цель работы:** изучить методы организации списочных структур в динамической памяти. Реализовать алгоритмы помещения и изъятия элементов из стека, дека или очереди.

Теория

**Стек** характерен тем, что получить доступ к его элементам можно лишь с одного конца, называемого вершиной стека; стек – структура данных типа «список», функционирующая по принципу **LIFO** (last in – first out, «последним пришёл – первым вышел»).

Графически его удобно изобразить в виде вертикального списка, например, стопки книг, где чтобы воспользоваться одной из них, и не нарушить установленный порядок, нужно поднять все те книги, что лежат выше нее, а положить книгу можно лишь поверх всех остальных.

Впервые стек был предложен в 1946 году Аланом Тьюрингом, как средство возвращения из подпрограмм. В 1955 году немцы Клаус Самельсон и Фридрих Бауэр из Технического университета Мюнхена использовали стек для перевода языков программирования и запатентовали идею в 1957 году. Но международное признание пришло к ним лишь в 1988 году. На рисунке выше показан стек, операции над элементами которого, происходят строго с одного конца: для включения нужного элемента в n-ую ячейку, необходимо сдвинуть n-1

элементов, и исключить тот элемент, который занимает n-ую позицию.

Стек, чаще всего, реализуется на основе обычных массивов, односвязных и двусвязных списков. В зависимости от конкретных условий, выбирается одна из этих структур данных.

Основными операциями над стеками являются:

* добавление элемента;
* удаление элемента;
* чтение верхнего элемента.

Динамическая реализация стека на основе списка

Для начала определим структуру узел:

#include <iostream> #include <cstdlib>

using namespace std;

|  |  |  |
| --- | --- | --- |
| #define | STACK\_OVERFLOW | -100 |
| #define | STACK\_UNDERFLOW | -101 |
| #define | OUT\_OF\_MEMORY | -102 |
| typedef | int T; |  |

struct Node {

T value;

Node \*next;

};

Функция вставки первого элемента проста: создаём новый узел. Указатель next кидаем на старый узел. Далее указатель на вершину стека перекидываем на вновь созданный узел. Теперь вершина стека указывает на новый узел.

void push(Node \*\*head, T value) {

Node \*tmp = (Node \*)malloc(sizeof(Node)); if (tmp == NULL) {

exit(STACK\_OVERFLOW);

}

tmp->next = \*head; tmp->value = value;

\*head = tmp;

}

Функция pop берёт первый элемент (тот, на который указывает вершина), перекидывает указатель на следующий элемент и возвращает первый. Здесь есть два варианта – можно вернуть узел или значение. Если вернём значение, то придётся удалять узел внутри функции.

Node \* pop\_node(Node \*\*head) { Node \*out;

if ((\*head) == NULL) { exit(STACK\_UNDERFLOW);

}

out = \*head;

\*head = (\*head)->next;

return out;

}

T pop\_t(Node \*\*head) {

Node \*out; T value;

if ((\*head) == NULL) { exit(STACK\_UNDERFLOW);

}

out = \*head;

\*head = (\*head)->next; value = out->value; free(out);

return value;

}

Теперь вместо проверки на длину массива везде используется проверка на равенство

NULL вершины стека.

Простая функция peek возвращает значение, хранящееся в голове стека.

T peek(const Node \*head) { if (head == NULL) {

exit(STACK\_UNDERFLOW);

}

return head->value;

}

Печать стека достаточно интересна: для этого необходимо извлекать элемент из стека, печатать его значение на экран, а затем добавлять его в дополнительный временный (служебный) стек. После того, как основной стек будет опустошен, все элементы по очереди возвращаются из дополнительного стека в основной, затем дополнительный стек уничтожается.

void printStack(Node \*head) {

Node \*new\_head = NULL; Node \*tmp = NULL;

while (head) {

tmp = pop\_node(&head); cout << tmp->value << " ";

push(&new\_head, tmp->value);

}

while (new\_head) {

tmp = pop\_node(&new\_head); push(&head, tmp->value);

}

free(tmp);

}

И ещё одна проблема – теперь нельзя просто посмотреть размер стека. Функция, возвращающая количество элементов стека работает аналогично функции печати:

int getSize(Node \*head) { int size = 0;

Node \*new\_head = NULL; Node \*tmp = NULL;

while (head) {

tmp = pop\_node(&head); size++;

push(&new\_head, tmp->value);

}

while (new\_head) {

tmp = pop\_node(&new\_head); push(&head, tmp->value);

}

free(tmp);

return size;

}

Конечно, можно хранить размер отдельно, можно обернуть стек со всеми данными ещё в одну структуру и т.д.

Два варианта реализации функции main и результат выполнения программы:

int main(void) {

Node \*head = NULL;

for (int i = 0; i < 10; i++) push(&head, i);

cout << "Stack: "; printStack(head);

cout << endl << "Stack size: " << getSize(head) << endl << endl; cout << "Peek and Pop:" << endl;

while (head) {

cout << "peek: " << peek(head) << "\t"; cout << "pop\_t: " << pop\_t(&head) << endl;

}

return 0;

}

или

int main(void) {

Node \*head = NULL; Node \*tmp;

for (int i = 0; i < 10; i++) push(&head, i);

cout << "Stack: "; printStack(head);

cout << endl << "Stack size: " << getSize(head) << endl << endl; cout << "Peek and Pop:" << endl;

while (head) {

cout << "peek: " << peek(head) << "\t"; tmp = pop\_node(&head);

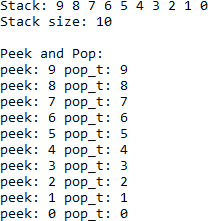
cout << "pop\_t: " << tmp->value << endl; free(tmp);

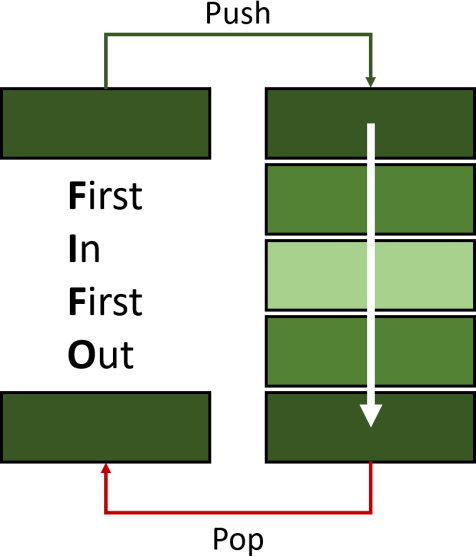
}

return 0;

}

Результат тестирования работы функций main():



**Очередь** – структура данных типа «список», позволяющая добавлять элементы лишь в конец списка, и извлекать их из его начала. Она функционирует по принципу FIFO (First In – First Out – «первым пришёл – первым вышел»), для которого характерно, что все элементы 𝑎1, 𝑎2,

…, 𝑎𝑛−1, 𝑎𝑛, добавленные раньше элемента 𝑎𝑛+1, должны быть удалены прежде, чем будет удален элемент 𝑎𝑛+1.

Также очередь может быть определена как частный случай односвязного списка, который обслуживает элементы в порядке их поступления. Как и в «живой» очереди, здесь первым будет обслужен тот, кто пришел первым.

Стандартный набор операций (часто у разных авторов он не идентичен), выполняемых над очередями, совпадает с тем, что используется при обработке стеков:

* добавление элемента;
* удаление элемента;
* чтение первого элемента.

Только, если в отношении стека в момент добавления или удаления элемента допустимо задействование лишь его вершины, то касательно очереди эти две операции должны быть применены так, как это регламентировано в определении этой структуры данных, то есть добавление – в конец, удаление – из начала. Далее, при реализации интерфейса очереди, список стандартных операций будет расширен.

Выделяют два способа программной реализации очереди. Первый из них основан на базе массива, а второй на базе указателей (связного списка). Первый способ – статический, так как очередь представляется в виде простого статического массива, второй – динамический.

Динамическая реализация односвязной очереди на основе списка

Данный способ предполагает работу с динамической памятью. Для представления очереди используется односвязный список, в конец которого помещаются новые элементы, а старые извлекаются, соответственно, из начала списка. Здесь каждый узел списка имеет два поля: информационное и связующее:

#include <iostream> #include <cstdlib> #include <clocale>

using namespace std; struct Node {

int value;

Node \*next;

};

Также понадобиться определить указатели на начало и конец очереди:

struct Queue {

Node \*head;

Node \*tail;

};

Следующее консольное приложение обслуживает очередь, каждый элемент которой – целое число. Весь процесс обуславливают две основные операции:

* PushTail;
* PopHead.

Функция PushTail() добавляет в конец очереди новый элемент:

int PushTail(Queue \*q, int v) { Node \*n;

if ((n = (Node \*)malloc(sizeof(Node))) == NULL) return -1;

n->value = v; n->next = NULL;

if (q->head == NULL) { q->head = n;

} else {

q->tail->next = n;

}

q->tail = n;

return 0;

}

Функция PopHead() удаляет один элемент из головы очереди.

int popHead(Queue \*q, int \*vptr) {

if (q->head == NULL) { return -1;

} else {

Node \*n = q->head->next;

\*vptr = q->head->value; free(q->head);

q->head = n;

}

return 0;

}

Для упрощения читаемости исходного кода вынесем операции работы с меню в отдельную функцию menu().

int menu(void) {

setlocale(LC\_ALL, "Russian");

cout << "1 - добавить в конец" << endl; cout << "2 - удалить из головы" << endl; cout << "0 - выход" << endl << "> ";

int ret; cin >> ret;

if (ret < 0 || ret > 2) ret = -1;

return ret;

}

Протестируем функции, написав следующее содержание функции main(). Результаты тестирования представлены на рисунке 5.

int main() {

Queue q = { NULL, NULL };

int ret;

while ((ret = menu()) > 0) { switch (ret) {

case 1: {

int val;

cout << endl << "Value: "; cin >> val;

if (pushTail(&q, val))

cout << "Memory error!" << endl;

break;

}

case 2: {

int val;

if (popHead(&q, &val))

cout << endl << "Empty queue." << endl;

else

cout << endl << "Returned: " << val << endl;

break;

}

case 0: {

exit(1);

}

default: {

cout << "Wrong choice!" << endl;

}

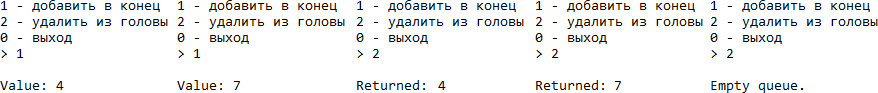
}

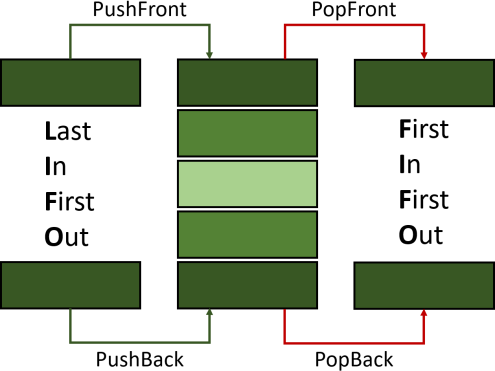
}

system("pause"); return 0;

}

Результат тестирования функций работы с очередью:



**Дек** (deque – double ended queue, «двусторонняя очередь») – структура данных типа «список», функционирующая одновременно по двум принцам организации данных: **FIFO** и **LIFO**. Определить дек можно как очередь с двумя сторонами, так и стек, имеющий два конца. То есть данный подвид списка характерен двухсторонним доступом: выполнение поэлементной операции, определенной над деком, предполагает возможность выбора одной из его сторон в качестве активной.

Число основных операций, выполняемых над стеком и очередью, как помнит читатель, равнялось трем: добавление элемента, удаление элемента, чтение элемента. При этом не указывалось место структуры данных, активное в момент их выполнения, поскольку ранее оно однозначно определялось свойствами (определением) самой структуры. Теперь, ввиду дека как обобщенного случая, для приведенных операций следует указать эту область. Разделив каждую из операций на две: одну применительно к «голове» дека, другую – его «хвосту», получим набор из шести операций:

* добавление элемента в начало;
* добавление элемента в конец;
* удаление первого элемента;
* удаление последнего элемента;
* чтение первого элемента;
* чтение последнего элемента.

На практике этот список может быть дополнен проверкой дека на пустоту, получением его размера и некоторыми другими операциями.

В плане реализации двусторонняя очередь очень близка к стеку и обычной очереди: в качестве ее базиса приемлемо использовать как массив, так и список.

Реализация дека (двунаправленной очереди)

Обычно для реализации дека используют двусвязный линейный список. То есть, на основе готовых функций списка создают новые для дека. Иными словами, для реализации двунаправленной очереди необходимо переписать несколько функций, которые будут обеспечивать корректную работу с динамически организованным двусвязным списком, как с деком.

В первую очередь опишем узел и саму структуру данных:

/\* Базовый тип дека \*/ typedef int deckBaseType;

/\* Структура элемента дека \*/ struct element {

deckBaseType data; // данные

element \*prev; // указатель на предыдущий элемент element \*next; // указатель на следующий элемент

};

/\* Описание типа дек \*/ struct Deque {

element \*start; // указатель на левый фиктивный элемент element \*end; // указатель на правый фиктивный элемент int size;

};

Кроме того, добавим в начале файла переменные, отвечающие за служебную информацию, и подключим нужные библиотеки.

#include <iostream> #include <cstdlib> #include <clocale>

using namespace std;

/\* Описание исключительных ситуаций \*/ const int dequeOK = 0;

const int dequeNoMem = 2; const int dequeEmpty = 1;

/\* Переменная ошибок \*/ extern int dequeError = 0;

Опишем функции для непосредственной работы с деком.

void initDeque(Deque \*D); // инициализация дека void putDequeLeft(Deque \*D, dequeBaseType E); // включение в начало дека

void getDequeLeft(Deque \*D, dequeBaseType \*E); // исключение из начала дека void putDequeRight(Deque \*D, dequeBaseType E); // включение в конец дека void getDequeRight(Deque \*D, dequeBaseType \*E); // исключение из конца дека void printDeque(Deque \*D); // печать дека

void doneDeque(Deque \*D); // удаление дека

int isDequeEmpty(Deque \*D); // предикат: пуст ли дек

Инициализация нового дека:

void initDeque(Deque \*D) {

// Выделение памяти для левого фиктивного элемента D->start = (element \*)malloc(sizeof(element));

// если памяти недостаточно if (!D->start) {

dequeError = dequeNoMem; return;

}

// выделение памяти для правого фиктивного элемента D->end = (element \*)malloc(sizeof(element));

// если памяти недостаточно if (!D->end) {

dequeError = dequeNoMem;

free((void \*)D->start); // удаление левого фиктивного элемента D->start = NULL;

return;

}

// элемент, следующий за стартовым - конечный D->start->next = D->end;

// элемента, предшествующего стартовому элементу нет D->start->prev = NULL;

// элемента, следующего за конечным элементом нет D->end->next = NULL;

// элемент, предшествующий конечному - стартовый элемент D->end->prev = D->start;

D->size = 0;

dequeError = dequeOK; // всё в порядке

}

Очистка дека:

void doneDeque(Deque \*D) {

dequeBaseType E; // создание временного элемнета

while (!isDequeEmpty(D)) // пока дек не пуст getDequeLeft(D, &E); // извлечь элемент

// удаление фиктивных (крайних) элементов free((void \*)D->start);

free((void \*)D->end);

}

Функция, проверяющая, пустой дек или нет:

int isDequeEmpty(Deque \*D) {

if (D->start->next == D->end) { dequeError = dequeEmpty; return 1;

}

return 0;

}

Далее описаны функции, которые обеспечивают добавление и изъятие элементов из обоих концов двунаправленной очереди.

/\* Добавление элемента в начало \*/

void putDequeLeft(Deque \*D, dequeBaseType E) {

// выделение памяти под новый элемент

element \*pntr = (element \*)malloc(sizeof(element));

// если памяти недостаточно if (!pntr) {

dequeError = dequeNoMem; return;

}

pntr->data = E; // запись информации в новый элемент

pntr->next = D->start->next; // новый элемент указывает справа на следующий pntr->prev = D->start; // новый элемнет указывает слева на текущий

D->start->next->prev = pntr; // следующий элемент указывает слева на новый D->start->next = pntr; // текущий элемент указывает справа на новый D->size++;

}

/\* Извлечение элемента из начала \*/

void getDequeLeft(Deque \*D, dequeBaseType \*E) {

// если дек пуст

if (isDequeEmpty(D)) { return;

}

\*E = D->start->next->data; // запись информации в переменную

element \*pntr = D->start->next; // временный элемент указывает на удаляемый D->start->next = pntr->next; // стартовый элемент указывает на второй pntr->next->prev = D->start; // второй элемент указывает на стартовый free((void \*)pntr); // удаление временного элемента

D->size--;

}

/\* Добавление элемента в конец \*/

void putDequeRight(Deque \*D, dequeBaseType E) {

// выделение памяти под новый элемент

element \*pntr = (element \*)malloc(sizeof(element));

// если памяти недостаточно if (!pntr) {

dequeError = dequeNoMem; return;

}

pntr->data = E; // запись информации в новый элемент

pntr->prev = D->end->prev; // новый элемент указывает слева на последний pntr->next = D->end; // новый элемнет указывает справа на конечный D->end->prev->next = pntr; // последний элемент указывает справа на новый D->end->prev = pntr; // конечный элемент указывает слева на новый

D->size++;

}

/\* Извлечение элемента из конца \*/

void getDequeRight(Deque \*D, dequeBaseType \*E) {

// если дек пуст

if (isDequeEmpty(D)) { return;

}

\*E = D->end->prev->data; // запись информации в переменную

element \*pntr = D->end->prev; // временный элемент указывает на удаляемый

D->end->prev = pntr->prev; // конечный элемент указывает на предпоследний pntr->prev->next = D->end; // предпоследний элемент указывает на конечный free((void \*)pntr); // удаление временного элемента

D->size--;

}

Протестируем работу описанных выше функций, посредствам функции main()

следующего содержания:

int main(void) {

Deque \*d = (Deque \*)calloc(1, sizeof(Deque)); // создадим указатель на дек initDeque(d); // и проинициализируем его

putDequeLeft(d, 12); // добавим число 12 в начало cout << "add: 12" << endl;

putDequeLeft(d, 7); // добавим число 7 в начало cout << "add: 7" << endl;

putDequeRight(d, 248); // добавим число 248 в конец cout << "add: 248" << endl;

cout << "Now into deque: "; printDeque(d);

cout << endl; int k;

cout << endl << "start -> end" << endl;

getDequeLeft(d, &k); // извлечём из начала

cout << k << " "; // выведем на экран [1]

putDequeRight(d, k); // и добавим в конец

getDequeLeft(d, &k); //

cout << k << " "; // [2]

putDequeRight(d, k); //

getDequeLeft(d, &k); //

cout << k << " "; // [3]

putDequeRight(d, k); // cout << endl << "end -> start" << endl;

getDequeRight(d, &k); // извлечём из конца

cout << k << " "; // выведем на экран [1]

getDequeRight(d, &k); //

cout << k << " "; // [2]

getDequeRight(d, &k); //

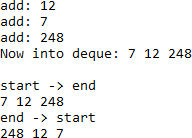
cout << k << " "; // [3] cout << endl;

doneDeque(d); // очистим дек

system("pause"); return 0;

}

Результат выполнения программы:



Практика

Создайте указанную динамическую структуру данных (ДСД) на основе списка для хранения данных, указанных в варианте задания. Напишите следующие функции:

1. Функции для создания новой ДСД и ее удаления.
2. Функции добавления элементов (все, что необходимы для полноценной работы).
3. Функции удаления элементов (все, что необходимы для полноценной работы).
4. Функция печати всех элементов ДСД.
5. Функция очистки ДСД (то есть удаления всех вложенных в нее элементов).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Вариант** | **Задание** | **Вариант** | **Задание** | **Вариант** | **Задание** |
| 1.1 | стек  char[15] | 2.1 | стек  double | 3.1 | очередь FIFO  char[20] |
| 1.2 | очередь FIFO  float | 2.2 | дек  char | 3.2 | дек  double |
| 1.3 | стек  char[20] | 2.3 | очередь FIFO  double | 3.3 | стек  char |
| 1.4 | дек  double | 2.4 | стек  char[15] | 3.4 | очередь FIFO  char[15] |
| 1.5 | стек  char | 2.5 | очередь FIFO  float | 3.5 | стек  float |
| 1.6 | очередь FIFO  char[15] | 2.6 | дек  char[20] | 3.6 | дек  char[20] |
| 1.7 | дек  float | 2.7 | стек  double | 3.7 | очередь FIFO  double |
| 1.8 | очередь FIFO  char[20] | 2.8 | очередь FIFO  char | 3.8 | дек  char |
| 1.9 | дек  double | 2.9 | стек  char[15] | 3.9 | стек  char[15] |
| 1.10 | стек  char | 2.10 | дек  float | 3.10 | очередь FIFO  float |
| 1.11 | дек  char[15] | 2.11 | очередь FIFO  float | 3.11 | дек  char[20] |
| 1.12 | очередь FIFO  float | 2.12 | стек  char[20] | 3.12 | стек  char[20] |
| 1.13 | стек  char[15] | 2.13 | дек  double | 3.13 | очередь FIFO  char[20] |

Лабораторная работа №9

**Сортировка и поиск**

**Цель работы:** изучить быстрые и медленные алгоритмы сортировки. Изучить алгоритмы линейного и бинарного поиска.

Теория

В мире компьютеров сортировка и поиск принадлежат к числу наиболее распространенных и хорошо изученных задач. Процедуры сортировки и поиска используются почти во всех программах управления базами данных, (а также в компиляторах, интерпретаторах и операционных системах. В настоящей главе представлены основные алгоритмы сортировки и поиска. Как вы сможете убедиться, они также иллюстрируют некоторые важные приемы программирования на языке С. Вообще говоря, поскольку целью сортировки, является облегчение и ускорение поиска данных, алгоритмы сортировки рассматриваются в первую очередь.

Сортировка

**Сортировка** – это упорядочивание набора однотипных данных по возрастанию или убыванию. Сортировка является одной из наиболее приятных для умственного анализа категорией алгоритмов, поскольку процесс сортировки очень хорошо определен. Алгоритмы сортировки были подвергнуты обширному анализу, и способ их работы хорошо понятен. К сожалению, вследствие этой изученности сортировка часто воспринимается как нечто само собой разумеющееся. При необходимости отсортировать данные многие программисты просто вызывают стандартную функцию qsort(), входящую в стандартную библиотеку <cstdlib>. Однако различные подходы к сортировке обладают разными характеристиками. Несмотря на то, что некоторые способы сортировки могут быть в среднем лучше, чем другие, ни один алгоритм не является идеальным для всех случаев. Поэтому широкий набор алгоритмов сортировки – полезное добавление в инструментарий любого программиста.

Будет полезно кратко остановиться на том, почему вызов qsort() не является универсальным решением всех задач сортировки. Во-первых, функцию общего назначения вроде qsort() невозможно применить во всех ситуациях. Например, qsort() сортирует только массивы в памяти. Она не может сортировать данные, хранящиеся в связанных списках. Во- вторых, qsort() – параметризованная функция, благодаря чему она может обрабатывать широкий набор типов данных, но вместе с тем вследствие этого она работает медленнее, чем эквивалентная функция, рассчитанная на какой-то один тип данных. Наконец, как вы увидите, хотя алгоритм быстрой сортировки, примененный в функции qsort(), очень эффективен в общем случае, он может оказаться не самым лучшим алгоритмом в некоторых конкретных ситуациях.

Существует две общие категории алгоритмов сортировки: алгоритмы, сортирующие объекты с произвольным доступом (например, массивы или дисковые файлы произвольного доступа), и алгоритмы, сортирующие последовательные объекты (например, файлы на дисках и лентах или связанные списки). В данной лабораторной работе рассматриваются только алгоритмы первой категории, поскольку они наиболее полезны для среднестатистического программиста.

Чаще всего при сортировке данных лишь часть их используется в качестве ключа сортировки. Ключ – это часть информации, определяющая порядок элементов. Таким образом, ключ участвует в сравнениях, но при обмене элементов происходит перемещение всей структуры данных. Например, в списке почтовой рассылки в качестве ключа может использоваться почтовый индекс, но сортируется весь адрес. Для простоты в нижеследующих примерах будет производиться сортировка массивов символов, в которых ключ и данные совпадают. Далее вы увидите, как адаптировать эти методы для сортировки структур данных любого типа.

Классы алгоритмов сортировки

Существует три общих метода сортировки массивов:

* Обмен
* Выбор (выборка)
* Вставка

Чтобы понять, как работают эти методы, представьте себе колоду игральных карт. Чтобы отсортировать карты методом обмена, разложите их на столе лицом вверх и меняйте местами карты, расположенные не по порядку, пока вся колода не будет упорядочена. В методе выбора разложите карты на столе, выберите карту наименьшей значимости и положите ее в руку. Затем из оставшихся карт снова выберите карту наименьшей значимости и положите ее на ту, которая уже находится у вас в руке. Процесс повторяется до тех пор, пока в руке не окажутся все карты; по окончании процесса колода будет отсортирована. Чтобы отсортировать колоду методом вставки, возьмите все карты в руку. Выкладывайте их по одной на стол, вставляя каждую следующую карту в соответствующую позицию. Когда все карты окажутся на столе, колода будет отсортирована.

Медленные алгоритмы сортировки

Среди медленных алгоритмов сортировки наибольшее распространение получили следующие алгоритмы:

* алгоритм сортировки вставками;
* алгоритм сортировки пузырьком;
* минимаксные алгоритмы;
* алгоритм сортировки перечислениями.

Алгоритм сортировки вставками заключается в прохождении по всем элементам массива, начиная со второго, при котором каждый элемент массива вставляется на свое место. Для этого, начальный элемент каждой итерации цикла помещается во временную переменную и его значение сравнивается со всеми элементами, стоящими до этого элемента. Если их значение больше текущего, то они сдвигаются на один элемент вправо. После достижения элемента меньшего текущего, или начала массива, текущий элемент вставляется на новое место. Так продолжается до последнего элемента массива.

void sortInsert(TYPE array[], int num) {

// цикл по всему массиву начиная со второго элемента for (int j = 1; j < num; j++) {

// сохраняем индекс предыдущего элемента в массиве int i = j - 1; // перед текущим элементом

// сохраняем текущий элемент в локальной переменной TYPE value = array[j];

// цикл по всем элементам от текущего до первого элемента,

// меньшего его по значению, или до начала массива while ((value < array[i]) && (i >= 0)) {

// сдвиг всех ранее идущих элементов на одну позицию "вправо" array[i + 1] = array[i];

// движение в сторону начала массива i--;

}

// вставка текущего элемента на новую позицию в массиве array[i + 1] = value;

}

}

**ПРИМЕЧАНИЕ!** Здесь и далее во всех функциях сортировки:

* 1. первый параметр array – массив, который необходимо упорядочить;
  2. второй параметр num – количество элементов в массиве.

Алгоритм пузырьковой сортировки является наиболее распространенным среди медленных алгоритмов сортировки. Он заключается в сравнении двух элементов попарно и их перестановкой согласно направлению сортировки. Существует несколько реализаций.

void sortFullBubble(TYPE array[], int num) {

// внешний и внутренний циклы от первого до последнего элемента в массиве for (int i = 0; i < num - 1; i++) {

for (int j = 0; j < num - 1; j++) {

// если предыдущий элемент "больше" последующего элемента,

// то обмен значений элементов через дополнительную переменную if (array[j] > array[j + 1]) {

TYPE tmp = array[j]; array[j] = array[j + 1]; array[j + 1] = tmp;

}

}

}

}

Приведенный алгоритм сортировки полным пузырьком не учитывает «эффекта пузырька»: самый легкий (тяжелый) элемент за одну итерацию по массиву поднимается (опускается) до самого конца массива, поэтому его следует исключить из обработки на последующих итерациях по массиву.

void sortSimpleBubble(TYPE array[], int num) {

// в переменной last на каждой итерации по массиву содержится

// позиция последнего элемента, который необходимо обрабатывать int last = num;

// пока есть элементы для обрботки while (last > 0) {

// в переменной pos содержится индекс последнего элемента массива,

// значение которого было обменено с предыдущим элементом int pos = 0;

// цикл от первого элемента массива, до того элемента, который

// исключён из обработки на предыдущей итерации цикла while for (int i = 0; i < last - 1; i++) {

// если предыдущий элемент "больше" последующего элемента,

// то обмен значений элементов через дополнительную переменную if (array[i] > array[i + 1]) {

TYPE tmp = array[i]; array[i] = array[i + 1]; array[i + 1] = tmp;

// запоминаем позицию последнего переставленного элемента pos = i + 1;

}

}

// исключаем упорядоченные элементы из последующей обработки last = pos;

}

}

Суть минимаксных алгоритмов сортировки заключается в поиске максимального и/или минимального элементов в массиве и обмен значений найденных элементов с последним и/или первым элементами в массиве. После каждой итерации границы массива сужаются.

void sortMaximum(TYPE array[], int num) {

// цикл по количеству элементов в массиве while (num > 0) {

// записываем в переменную maxpos позицию начального максимума

// (первый элемент массива) int maxpos = 0;

// цикл поиска максимума

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

if (array[i] > array[maxpos]) // если новый максимум найден, maxpos = i; // то запоминаем его позицию

}

// если найденный максимум находится не на последней позиции,

// то обмен значений элементов через дополнительную переменную if (maxpos != num - 1) {

TYPE tmp = array[maxpos]; array[maxpos] = array[num - 1]; array[num - 1] = tmp;

}

// исключение нейденного и переставленного

// максимума из дальнейшей обработки num--;

}

}

Далее приведен алгоритм, реализованный в виде функции, осуществляющий сортировку минимаксным способом: ищутся и максимальный и минимальный элементы и помещаются в конец и начало рассматриваемого массива соответственно.

void sortMinMax(TYPE array[], int num) {

// в переменных lo и hi содержатся индексы первого

// и последнего рассматриваемых элемнетов массива int lo = 0, hi = num - 1;

// пока не сойдёмся на середине массива while (hi > lo) {

// позиции максимума и минимума в текущей итерации int imax = lo, imin = lo;

// цикл поиска максимума и минимума массива for (int i = lo; i <= hi; i++) {

// если найден новый максимум, if (array[i] > array[imax])

imax = i; // то запоминаем его позицию

// если найден новый минимум, if (array[i] < array[imin])

imin = i; // то запоминаем его позицию

}

// если позиция максмума и минимума совпадают,

// то массив состоит из одинаковых элементов, if (imax == imin)

return; // поэтому выход из сортировки

// если максимум расположен не в последнем рассматриваемом элементе

// массива, то обмен значениями через дополнительную переменную if (imax != hi) {

TYPE tmp = array[imax]; array[imax] = array[hi]; array[hi] = tmp;

}

// если минимум расположен не в первом рассматриваемом элементе if (imin != lo) {

// если минимум находился на позиции последнего элемента,

// который был переставлен с максимальным элемнетом местами,

// то устанавливаем новую позицию максимума if (imin == hi)

imin = imax;

// обмен значениями через дополнительную переменную TYPE tmp = array[imin];

array[imin] = array[lo]; array[lo] = tmp;

}

// исключение найденных максимума и минимума из дальнейшей обработки lo++;

hi--;

}

}

Идея сортировки перечислением состоит в том, чтобы сравнить попарно все элементы и подсчитать, сколько из них меньше каждого отдельного элемента. Для подсчета используется вспомогательный массив, в котором, после завершения подсчета, находится окончательное положение элементов исходного массива в упорядоченной последовательности.

void sortEnumeration(TYPE array[], int num) {

// создание целочисленного массива

int \*intarray = (int \*)calloc(num, sizeof(int));

// если массив не создан, if (!intarray)

return; // то завершение сортировки

// создание дополнительного массива для сортировки TYPE \*newarray = (TYPE \*)calloc(num, sizeof(TYPE));

// если массив не создан, if (!newarray) {

free(intarray); // то удаление целочисленного массива return; // и завершение сортировки

}

// заполнение целочисленного массива нулями memset(intarray, 0, num \* sizeof(int));

// два вложенных цикла перебора всех комбинаций из

// двух элементов исходного массива для их сравнения for (int i = 0; i < num - 1; i++) {

for (int j = i + 1; j < num; j++) {

// если i-ый элемент "больше" j-го, if (array[i] > array[j])

intarray[i]++; // то инкрементируем i-ую ячейку else intarray[j]++; // в противном случае j-ую ячейку

}

}

// заполнение дополнительного массива элементами

// из исходного массива в упорядоченном виде for (int i = 0; i < num; i++)

newarray[intarray[i]] = array[i];

// копирование дополнительного массива в исходный массив memcpy(array, newarray, num \* sizeof(int));

free(intarray); // удаление целочисленного массива free(newarray); // удаление дополнительного массива

}

Быстрые алгоритмы сортировки

Среди быстрых алгоритмов сортировки наибольшее распространение получили следующие алгоритмы:

* алгоритм Шелла;
* алгоритм Хоара;
* алгоритм Флойда.

Идея алгоритма Шелла состоит в том, что в исходном наборе элементов (массиве) сначала упорядочиваются элементы, расположенные на расстоянии друг от друга, т.е. итерации по массиву осуществляются через несколько элементов. Затем шаг уменьшается и производится сортировка с новым размером шага. Так продолжается до тех пор, пока шаг не станет равным единице. Таким образом, алгоритм Шелла представляет собой обычный медленный алгоритм сортировки, в котором ускорение процесса осуществляется за счет того, что на первых этапах (с большим шагом) обрабатывается меньше элементов, и они упорядочиваются быстрее, перемещаясь по массиву с большими шагами. На практике скорость данного алгоритма редко достигает указанной ранее зависимости (𝑛 × log(2 ∙ 𝑛)), поэтому данный алгоритм относят к классу быстрых алгоритмов скорее условно, чем в действительности. Сама сортировка в алгоритме Шелла может осуществляться любым из рассмотренных ранее медленных алгоритмов.

void ShellSort(TYPE array[], int num) {

// массив, содержащий размеры шагов обработки массива

// данных (должен представлять собой убывающую

// последовательность с последним элементом равным 1) int steps[] = { 9, 5, 3, 1 };

// цикл сортировки Шелла (по всему массиву шагов) for (int n = 0; n < 4; n++) {

int first = 0; // установка первой позиции в массиве

// пока не дойдём до конца массива while (first < num) {

// устанавливаем в переменную pos текущую позицию,

// в переменную min - позицию текущего минимума int pos = first, min = first;

// цикл минимального элемента от позиции first до конца массива while (pos < num) {

// если найден новый минимум, if (array[pos] < array[min])

min = pos; // то запоминаем его позицию

// переходим к следующему элементу

// (с заданным на текущей итерации шагом) pos += steps[n];

}

// если позиция найденного минимума не совпадает с первым,

// то обмен значений элементов через дополнительную переменную if (min > first) {

TYPE tmp = array[min]; array[min] = array[first]; array[first] = tmp;

}

// переход к новой итерации по массиву first += steps[n];

}

}

}

Алгоритм быстрой сортировки Хоара заключается в следующем: берется массив, выбирается один элемент, а все остальные элементы делятся на два подмножества - элементы, большие выбранного или равные ему, и элементы, меньшие выбранного. Затем процедура применяется рекурсивно к каждому из двух подмножеств. Если в подмножестве меньше двух элементов, сортировка ему не требуется, и рекурсия на этом прекращается.

void HoareSort(TYPE array[], int num) {

// вызов сортировки для всего массива quickSort(array, 0, num - 1);

}

void quickSort(TYPE \*array, int iLo, int iHi) {

// заносим границы подмножества в лакальные переменные int Lo = iLo, Hi = iHi;

// выбираем один элемент, с которым будем сравнивать все остальные

// элементы модмножества массива (центральный элемент) TYPE mid = array[(Lo + Hi) / 2];

do {

// поиск элемента "больше" выбранного элемента слева от него while (array[Lo] < mid) Lo++;

// поиск элемента "меньше" выбранного элемента справа от него while (array[Hi] > mid) Hi--;

// если элементы найдены, if (Lo <= Hi) {

// то обмен значений через дополнительную переменную TYPE tmp = array[Lo];

array[Lo] = array[Hi]; array[Hi] = tmp;

// модификация индексов для продолжения поиска Lo++;

Hi--;

}

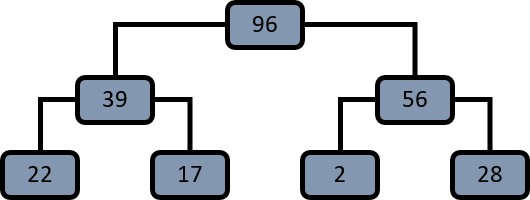
// выход из цикла, когда всё подможество будет рассмотрено

} while (Lo <= Hi);

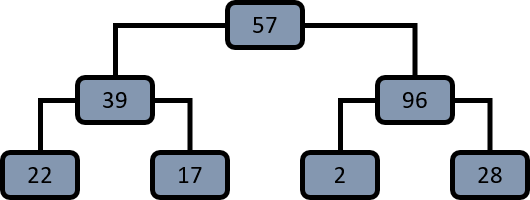
// рекурсивный вызов для левой части подмножества if (Hi > iLo) quickSort(array, iLo, Hi);

// рекурсивный вызов для правой части подмножества if (Lo < iHi) quickSort(array, Lo, iHi);

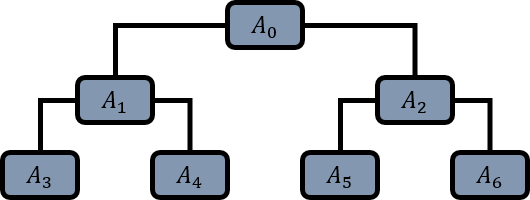
}

Алгоритм Флойда является самым оптимальным из алгоритмов сортировки. В нём активно используется упорядоченное двоичное дерево:

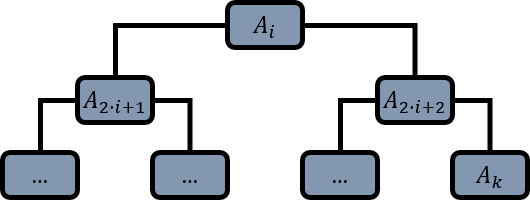
Значение в каждой из его вершин не меньше, чем значения в его дочерних вершинах. Двоичное дерево называется частично упорядоченным, если свойство упорядоченности выполняется для каждой из его вершин, однако для корня это свойство нарушается. Пример частично упорядоченного дерева:



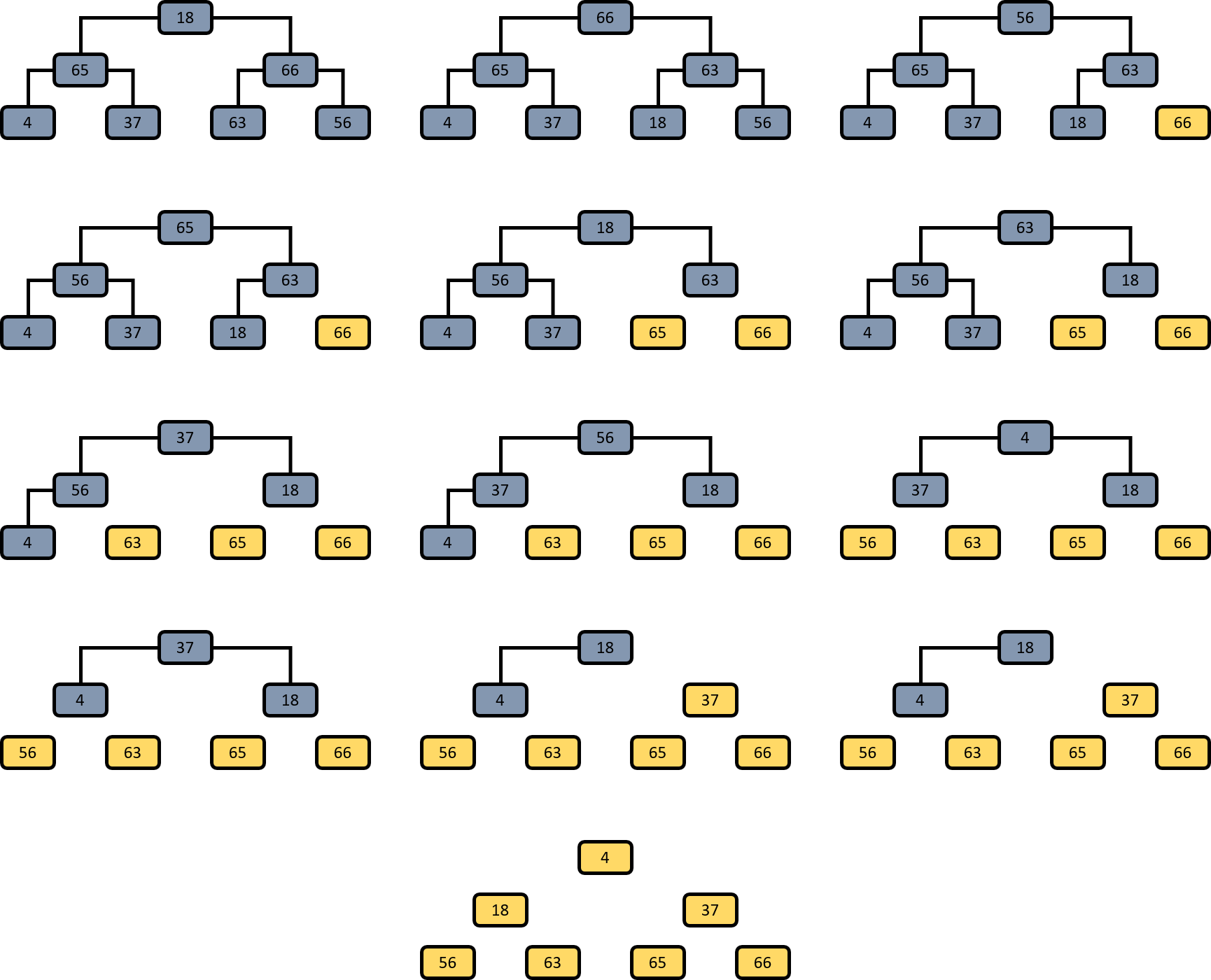
Структура дерева позволяет сохранить состояние процесса сортировки последовательности на каждом его шаге, с целью использования этого состояния в дальнейших расчетах и уменьшении числа операций сравнения при поиске наибольшего (наименьшего) из оставшихся элементов.

В алгоритме сортировки Флойда исходная последовательность данных представляется в виде дерева на смежной памяти. В таком дереве ребра присутствуют неявно и вычисляются с помощью арифметических операций над индексами элементов массива. Корень дерева a[0], за каждой вершиной a[k] следуют вершины a[2\*k+1] и a[2\*k+2]. На рисунке ниже приведен пример нумерации вершин двоичного дерева на смежной памяти.

Основу алгоритма составляет функция surface (a[i..k]) всплытия Флойда, которая за

𝑄(𝑛)~(log(2 ∙ 𝑛)) сравнений преобразует почти упорядоченное поддерево в упорядоченное. Поддерево представляется на одномерно массиве a[i..k], где a[i] – корень дерева, a[k] – максимальный элемент массива, который еще может принадлежать поддереву:

Функция заключается в том, что значение корня (здесь может нарушаться условие упорядоченности) всплывает по направлению к листьям (последний уровень вершин в дереве) до тех пор, пока дерево не преобразуется в упорядоченное. Функция surface всплытия Флойда позволяет в почти упорядоченном дереве найти наибольший (наименьший) элемент за число сравнений 𝑄(log(2 ∙ 𝑛)), преобразуя дерево к упорядоченному виду. В результате найденный элемент будет располагаться в вершине дерева. Для сортировки множества элементов, из них сначала организуется почти упорядоченное двоичное дерево при помощи повторного применения функции surface сначала к самым мелким его поддеревьям от листьев и затем ко все более крупным. Листья тривиально упорядочены, поэтому можно начинать с минимальных поддеревьев, содержащих несколько вершин и укрупнять их, каждый раз полностью, применяя алгоритм всплытия до тех пор, пока не будет достигнут корень дерева. Заметим, что каждое из поддеревьев, к которому применяется алгоритм всплытия, удовлетворяет условию почти упорядоченности, поскольку упорядочивание происходит от листьев к корню.

После того, как дерево упорядочено, наибольший (наименьший) элемент оказывается в его корне. Найденный элемент меняют местами с самым последним листом в дереве (последний элемент рассматриваемого массива), дерево уменьшается на одну вершину и все готово для определения нового наибольшего (наименьшего) элемента множества. На рисунке ниже показана полная последовательность перестановок и всплытий, которые происходят после формирования из исходного множества почти упорядоченного дерева и вплоть до того, как в этом дереве останется всего одна вершина, а исходное множество окажется отсортированным.

Листинг функции сортировки Флойда:

void FloidSort(TYPE array[], int num) {

// цикл для построения почти упорядоченных деревьев for (int i = num / 2; i > 0; i--)

surface(array, i - 1, num);

// цикл сортировки

for (int k = num - 1; k > 0; k--) {

// построение упорядоченного дерева surface(array, 0, k);

// обмен значениями первого элемента в массиве (корня дерева)

// и текущего последнего элемента в массиве TYPE tmp = array[k];

array[k] = array[0]; array[0] = tmp;

}

}

void surface(TYPE array[], int i, int k) {

// сохраняем корень в локальной переменной TYPE copy = array[i];

// вычисляем индекс элемента дерева "слева" от корня int m = 2 \* i + 1, j = 0;

while (m < k) {

// если это последний элемент в дереве, if (m == (k - 1))

j = m; // то запоминаем его позицию

// если элемент слева "больше" элемента справа, else if (array[m] > array[m + 1])

j = m; // то запоминаем позицию элемента справа

// иначе else

j = m + 1; // запоминаем позицию элемента справа

// если найденный элемент больше расположенного в корне дерева, if (array[j] > copy) {

array[i] = array[j]; // то записываем его в корень

i = j; // корень устанавливаем на его позицию m = 2 \* i + 1; // вычисляем индекс следующего элемента

}

// иначе выходим из цикла else break;

}

// итоговой вершине дерева присваиваем

// сохранённое значение исходной вершины array[i] = copy;

}

При выборе того или иного алгоритма сортировки следует учитывать то, что быстрые алгоритмы выгодны для сортировки достаточно больших массивов (размер которых исчисляется сотнями и даже тысячами элементов). Также необходимо обращать внимание на то, что сортируемые данные могут иметь большой объем. Поэтому алгоритмы сортировки следует реализовывать так, чтобы минимизировать число перестановок элементов в процессе сортировки.

Поиск

Базы данных существуют для того, чтобы время от времени пользователи могли найти нужную запись, введя ее ключ. Существует один метод поиска информации в неупорядоченном массиве, и другой для поиска в упорядоченном массиве. В набор стандартной библиотеки

<cstdlib> входит стандартная функция bsearch(). Тем не менее, как и в случае сортировки, процедуры общего назначения иногда совсем не эффективны при использовании в критических ситуациях из-за накладных расходов, связанных с их обобщением. Кроме того, функцию bsearch() невозможно применить к неупорядоченным данным.

Методы поиска

Для нахождения информации в неупорядоченном массиве требуется последовательный поиск, начинающийся с первого элемента и заканчивающийся при обнаружении подходящих данных либо при достижении конца массива. Этот метод применим для неупорядоченной информации, но также можно использовать и на отсортированных данных. Однако если данные уже отсортированы, можно применить двоичный поиск, который находит данные быстрее.

Последовательный поиск

Сущность этого алгоритма заключается в последовательном просмотре всех элементов множества и сравнении каждого из них с искомым значением. Как только элемент множества будет найден, то поиск завершается и возвращается номер найденного элемента во множестве. Если все множество было просмотрено, а искомый элемент не найден, то возвращается код отрицательного результата поиска.

int linearSearch(TYPE value, TYPE array[], int num) {

// цикл по всем элементам массива for (int i = 0; i < num; i++) {

// если текущий элемент массива равен искомому, if (array[i] == value)

return i; // то возвращаем его индекс

}

// если элемент не был найден,

// то возвращаем код отрицательного результата return -1;

}

Первый параметр value – искомое значение, второй параметр array – множество элементов в котором осуществляется поиск (массив), третий параметр num – количество элементов во множестве. Функция возвращает: положительное значение или ноль – индекс найденного элемента множества, -1 – если элемент не найден.

Приведенная функция позволяет найти только первый элемент во множестве. Для организации продолжения поиска элементов необходимо модифицировать данную функцию, добавив еще один параметр: номер элемента, с которого необходимо начинать поиск.

int linearSearchModified(TYPE value, TYPE array[], int num, int ind) {

// цикл по всем элементам массива, начиная с номером (индексом) ind for (int i = ind; i < num; i++) {

// если текущий элемент массива равен искомому, if (array[i] == value)

return i; // то возвращаем его индекс

}

// если элемент не был найден,

// то возвращаем код отрицательного результата return -1;

}

Двоичный поиск

Если данные, в которых производится поиск, отсортированы, для нахождения элемента можно применять метод, намного превосходящий предыдущий – двоичный поиск. В нем применяется метод половинного деления. Сначала проверим средний элемент. Если он больше, чем искомый ключ, проверим средний элемент первой половины, в противном случае – средний элемент второй половины. Будем повторять эту процедуру до тех пор, пока искомый элемент не будет найден либо пока не останется очередного элемента.

Например, чтобы найти число 4 в массиве {1 2 3 4 5 6 7 8 9} при двоичном поиске сначала проверяется средний элемент – число 5. Поскольку оно больше, чем 4, поиск продолжается в первой половине {1 2 3 4 5}. Средний элемент теперь равен 3. Это меньше, чем 4, поэтому первая половина отбрасывается. Поиск продолжается в части {4 5}. На этот раз искомый элемент найден.

В двоичном поиске количество сравнений в худшем случае равно log(2 ∙ 𝑛). В среднем случае количество немного ниже, а в лучшем – количество сравнений равно 1.

Для реализации множественного поиска необходимо в функцию множественного поиска передать массив, куда будут занесены индексы найденных элементов, а также размер этого массива (чтобы не допустить выхода за его границы).

int multiLinearSearch(TYPE value, TYPE array[], int num, int index[], int size) {

int count = 0; // количество найденых значений

int ind = -1; // начальный номер элемента для поиска

// пока искомое значение найдено во множестве

while ((ind = linearSearchModified(value, array, num, ind + 1)) != -1) {

// записываем индекс найденного элемента в массив index[count++] = ind;

if (count == size) // если нашли заданное количество элементов, break; // то прерываем цикл

}

return count; // возвращаем число найденных элементов

}

Параметры функции:

* value – искомое значение;
* array – множество, в котором осуществляется поиск;
* num – количество элементов во множестве;
* index – массив, в котором будут содержаться индексы найденных элементов;
* size – размер массива, ограничение количества искомых элементов. Функция возвращает количество найденных элементов.

Практика

1. Организуйте выгрузку данных из Вашей структуры данных в динамический массив.
2. Организуйте выгрузку данных из динамического массива в Вашу структуру данных.
3. Организуйте сортировку элементов массива по любому полю любым алгоритмом.
4. Организуйте медленный поиск записей в Вашей структуре данных (или быстрый поиск в предварительно выгруженном и отсортированном динамическом массиве) по любому полю.
5. Добавьте написанные Вами функции в курсовую работу.

Дополнение

Для проверки корректности работы описанных выше функций необходимо в файл исходного кода добавить следующее содержимое:

#include <iostream> #include <cstdlib> #include <ctime>

using namespace std; typedef int TYPE;

void sortInsert(TYPE array[], int num); void sortFullBubble(TYPE array[], int num);

void sortSimpleBubble(TYPE array[], int num); void sortMaximum(TYPE array[], int num); void sortMinMax(TYPE array[], int num);

void sortEnumeration(TYPE array[], int num); void ShellSort(TYPE array[], int num);

void HoareSort(TYPE array[], int num);

void quickSort(TYPE \*array, int iLo, int iHi);

void FloidSort(TYPE array[], int num); void surface(TYPE array[], int i, int k);

int linearSearch(TYPE value, TYPE array[], int num);

int linearSearchModified(TYPE value, TYPE array[], int num, int ind);

int multiLinearSearch(TYPE value, TYPE array[], int num, int index[], int size);

void printArray(char \*str, TYPE array[], int num); void randomInt(TYPE array[], int num);

int main(void) {

int n = 10;

TYPE \*arr = new TYPE[n]; randomInt(arr, n);

printArray((char \*) "Array", arr, n); sortInsert(arr, n);

printArray((char \*) "SortInsert", arr, n); cout << endl;

randomInt(arr, n);

printArray((char \*) "Array", arr, n); sortFullBubble(arr, n);

printArray((char \*) "SortFullBubble", arr, n); cout << endl;

randomInt(arr, n);

printArray((char \*) "Array", arr, n); sortSimpleBubble(arr, n);

printArray((char \*) "SortSimpleBubble", arr, n); cout << endl;

randomInt(arr, n);

printArray((char \*) "Array", arr, n); sortMaximum(arr, n);

printArray((char \*) "SortMaximum", arr, n); cout << endl;

randomInt(arr, n);

printArray((char \*) "Array", arr, n); sortMinMax(arr, n);

printArray((char \*) "SortMinMax", arr, n); cout << endl;

randomInt(arr, n);

printArray((char \*) "Array", arr, n); sortEnumeration(arr, n);

printArray((char \*) "SortEnumeration", arr, n); cout << endl;

randomInt(arr, n);

printArray((char \*) "Array", arr, n); ShellSort(arr, n);

printArray((char \*) "ShellSort", arr, n); cout << endl;

randomInt(arr, n);

printArray((char \*) "Array", arr, n); HoareSort(arr, n);

printArray((char \*) "HoareSort", arr, n); cout << endl;

randomInt(arr, n);

printArray((char \*) "Array", arr, n); FloidSort(arr, n);

printArray((char \*) "FloidSort", arr, n); cout << endl;

TYPE key; int x = 0;

randomInt(arr, n);

cout << "Enter value to search (linear): "; cin >> key;

x = linearSearch(key, arr, n); if (x != -1)

cout << "arr[" << x << "] = " << arr[x] << endl << endl;

cout << "Enter value to search (linear modified): "; cin >> key;

x = -1;

while ((x = linearSearchModified(key, arr, n, x + 1)) != -1) cout << "arr[" << x << "] = " << arr[x] << endl;

cout << endl;

cout << "Enter value to search (multi linear after sort): "; cin >> key;

int \*ind = new int[n]; sortFullBubble(arr, n);

int col = multiLinearSearch(key, arr, n, ind, n); for (int i = 0; i < col; i++)

cout << "arr[" << ind[i] << "] = " << arr[ind[i]] << endl;

return 0;

}

void printArray(char \*str, TYPE array[], int num) { cout << str << ": ";

for (int i = 0; i < num; i++) cout << array[i] << " ";

cout << endl;

}

void randomInt(TYPE array[], int num) { srand((unsigned int) time(NULL));

for (int i = 0; i < num; i++) array[i] = rand() % 50;

}

Результат работы программы:

