ЛАБОРАТОРНАЯ РАБОТА №5

ИССЛЕДОВАНИЕ РАБОТЫ С ДИНАМИЧЕСКОЙ ПАМЯТЬЮ

Цель работы: изучение принципов работы с динамической памятью.

Содержание работы:

1. Изучение операторов для создания динамических структур.
2. Изучение способов работы с динамическими структурами.
3. Разработка приложений обработки структур.
4. Оформление отчета.

Требования к отчету: индивидуальный отчет студента по выполнению работы должен быть выполнен машинописным или рукописным способам на листах формата А4 с типовым титульным листом. Отчет должен содержать:

1. Титульный лист установленного образца.
2. Задание и вариант.
3. Тексты программ.
4. Алгоритм работы программы
5. Скриншоты работы программ.

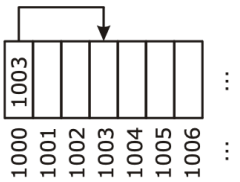
Длительность работы: 10 академических часов.

Защита работы: собеседование с преподавателем по контрольным вопросам.

Теоретические сведения

Указатели – одно из самых мощных и, в то же время, самых опасных средств любого языка программирования, а особенно С. Например, неинициализированные указатели (или указатели, содержащие неверные адреса) могут уничтожить операционную систему компьютера. И, что еще хуже, неправильное использование указателей порождает ошибки, которые крайне трудно обнаружить.

Указатель – это переменная, в которой хранится адрес дру- гого объекта (как правило, другой переменной). Например, если одна переменная содержит адрес другой переменной, говорят, что первая переменная ссылается на вторую.



Переменная, хранящая адрес ячейки памяти, должна быть объявлена как указатель. Объявление указателя состоит из имени базового типа, символа * и имени переменной. Общая форма этого объявления такова:

тип_указателя *имя_указателя;

Здесь тип_указателя означает базовый тип указателя. Им может быть любой допустимый тип.

Существуют два специальных оператора для работы с указателями:

- оператор разыменования указателя *
- оператор получения адреса &.

Оператор & является унарным и возвращает адрес своего операнда. Не забывайте: унарные операторы имеют только один операнд. Например, оператор присваивания

m = &count;

записывает в указатель m адрес переменной count. Этот адрес относится к ячейке памяти, которую занимает переменная count. Адрес и значение переменной никак не связаны друг с другом. Оператор & означает "адрес". Следовательно, предыдущий оператор присваивания можно прочитать так: присвоить указателю m адрес переменной count.

Если переменная занимает несколько ячеек памяти, ее адресом считается адрес первой ячейки.

Оператор разыменования указателя * является антиподом оператора &. Этот унарный оператор возвращает значение, хранящееся по указанному адресу. Например, если указатель m содержит адрес переменной count, то оператор присваивания

*q = *m;*

поместит значение count в переменную q. Символ * можно интерпретировать как "значение, хранящееся по адресу". В данном случае предыдущий оператор означает: "присвоить переменной q значение, хранящееся по адресу m".

В основе системы динамического распределения памяти в языке С лежат функции malloc() и free(). Во многих компиляторах предусмотрены дополнительные функции, позволяющие динамически выделять память, однако эти две – самые важные. Эти функции создают и поддерживают список свободной памяти. Функция malloc() выделяет память для объектов, а функция free() освобождает ее. Иными словами, при каждом вызове функция malloc() выделяет дополнительный участок памяти, а функция free() возвращает его операционной системе. Любая программа, использующая эти функции, должна включать в себя заголовочный файл stdlib.h.

Прототип функции malloc() выглядит следующим образом:

*void *malloc(size_t количество_байтов)*

под *size*_t подразумевается один из допустимых типов переменных.

Для выделения памяти под конкретный тип можно использовать оператор *sizeof(mun).*

Например: *malloc(5*sizeof(int))*

Выделит пятикратный размер ячейки типа int, иными словами массив из 5 элементов типа int.

Функция *free()* используется для освобождения выделенной динамической памяти и принимает в качестве параметра указатель на динамически сформированный участок памяти.

Например: *free(p)* освободит ранее созданный динамически участок памяти, на который указывает указатель p.

```
char *p;
p = (char *)malloc(1000*sizeof(char));
.
.
.
free(p);
```

Создаётся указатель символьного типа и для него выделяется 1000 элементов типа char. Функция malloc() возвращает указатель типа void на выделенную память, поэтому обязательно нужно явно преобразовать результат выполнения функции к типу указателя.

Указатели и массивы

Указатели и массивы тесно связаны между собой. Рассмотрим следующий фрагмент программы:

```
char str[80], *p1;
p1 = str;
```

Здесь указателю p1 присвоен адрес первого элемента массива str. Чтобы получить доступ к пятому элементу этого массива, следует выполнить один из двух операторов:

```
str[4]
```

или

```
*(p1+4)
```

Оба оператора вернут значение пятого элемента массива str. Поскольку индексация массивов начинается с нуля, то индекс пятого элемента массива str равен 4. Кроме того, пятый элемент массива можно получить, прибавив 4 к указателю p1, который вначале ссылается на первый элемент. Имя массива без индекса представляет собой указатель на начало массива, то есть на его первый элемент.

Этот пример можно обобщить. В языке С существуют два способа обращения к элементам массива: индексация и адресная арифметика. Хотя индексация массива нагляднее, адресная арифметика иногда оказывается эффективнее. Поскольку быстродействие программы относится к одним из ее важнейших свойств, программисты на языке С широко используют указатели для обращения к элементам массива.

Оператор typedef.

С помощью ключевого слова typedef можно определить новое имя типа данных. Новый тип при этом не создается, просто уже существующий тип получит новое имя. Это позволяет повысить машинонезависимость программ. Если в программе используется машинозависимый тип, достаточно его переименовать, и на новом компьютере для модификации программы придется изменить лишь одну строку с оператором typedef. Кроме того, с помощью оператора typedef можно давать типам осмысленные имена, что повышает наглядность программы. Общий вид оператора typedef таков:

typedef тип новое_имя

Здесь элемент тип обозначает любой допустимый тип, а элемент новое_имя – псевдоним этого типа. Новое имя является дополнительным. Оно не заменяет существующее имя типа.

Например, тип float можно переименовать следующим образом:

```
typedef float real;
```

Этот оператор сообщает компилятору, что real – это новое имя типа float. Теперь в программе вместо переменных типа float можно использовать переменные типа real:

```
real over_due;
```

Здесь переменная over_due является числом с плавающей точкой, но имеет тип real, а не float. В свою очередь, используя оператор typedef, тип real можно переименовать еще раз. Например, оператор

```
typedef real length;
```

сообщает компилятору, что length – это синоним слова real, которое является другим именем типа float.

Структуры

Структура – это набор переменных, объединенных общим именем. Она обеспечивает удобный способ организации взаимосвязанных данных. Объявление структуры создает ее шаблон, который можно использовать при создании объектов структуры (т.е. ее экземпляров). Переменные, входящие в структуру, называются ее членами (или элементами, или полями.)

Как правило, все члены структуры логически связаны друг с другом. Например, имя и адрес в списке рассылки естественно представлять в виде структуры. Следующий фрагмент программы демонстрирует, как объявляется структура, состоящая из полей, в которых хранятся имена и адреса. Ключевое слово struct сообщает компилятору, что объявляется именно структура.

Синтаксис объявления структуры

```
struct <имя> {
   <тип1> <поле1>;
   <тип2> <поле2>;
   ...
   <типN> <полеN>;
};
```

Например

```
struct point_t {
   int x;
   int y;
};
```

Полями структуры могут быть любые объявленные типы, кроме самой структуры этого же типа, но можно хранить указатель на структуру этого типа:

```
struct node {
   void* value;
   struct node *next;
};
```

В том случае, если несколько полей имеют один тип, то их можно перечислить через запятую:

```
struct Point3D {
   int x, y, z;
};
```

После того, как мы объявили структуру, можно создавать переменную такого типа с использованием служебного слова struct. Доступ до полей структуры осуществляется с помощью операции точка:

```
struct point_t {
   int x;
   int y;
};

void main() {
   struct point_t A;
   float distance;

   A.x = 10;
   A.y = 20;

   distance = sqrt((float) (A.x*A.x + A.y*A.y));

   printf("x = %.3f", distance);
   getch();
}
```

Когда мы определяем новую структуру с помощью служебного слова struct, в пространстве имён структур создаётся новый идентификатор. Для доступа к нему необходимо использовать служебное слово struct. Можно определить новый тип с помощью служебного слова typedef. Тогда будет создан псевдоним для нашей структуры, видимый в глобальном контексте.

```
//Определяем новую структуру
struct point_t {
   int x;
   int y;
};

//Определяем новый тип
typedef struct point_t Point;

void main() {
   //Обращение через имя структуры
   struct point_t p = {10, 20};
   //Обращение через новый тип
   Point px = {10, 20};

   getch();
}
```

Теперь при работе с типом Point нет необходимости каждый раз писать слово struct. Два объявления можно объединить в одно

```
typedef struct point_t {
   int x;
   int y;
} Point;
```

Доступ к отдельным членам структуры обеспечивается оператором "." (обычно его называют оператором "точка" или оператором доступа к члену структуры). Например, в следующем фрагменте программы полю x структуры point_t, объявленной ранее, присваивается координата 15:

```
p.x =15;
```

Имя экземпляра структуры указывается перед точкой, а имя члена структуры – после нее. Оператор, предоставляющий доступ к члену структуры, имеет следующий вид:

имя_экземпляра.имя_члена

В языке С на структуры можно ссылаться точно так же, как и на любой другой тип данных. Однако указатели на структуры имеют несколько особенностей.

Указатели на структуры объявляются с помощью символа *, стоящего перед именем экземпляра структуры. Например, указатель dot_pointer на структуру point_t объявляется так:

```
struct point_t *dot_pointer;
```

Указатели на структуры используются в двух ситуациях: для передачи структуры в функцию по ссылке и для создания структур данных, основанных на динамическом распределении памяти (например, связанных списков).

Передача структур в качестве аргументов функции имеет один существенный недостаток: в стек функции приходится копировать целую структуру. Если структура невелика, дополнительные затраты памяти будут относительно небольшими. Однако, если структура состоит из большого количества членов или ее членами являются большие массивы, затраты ресурсов могут оказаться чрезмерными. Этого можно избежать, если передавать функции не сами структуры, а лишь указатели на них.

Если функции передается указатель на структуру, в стек заталкивается только ее адрес. В результате вызовы функции выполняются намного быстрее. Второе преимущество, которое проявляется в некоторых случаях, заключается в том, что, получив адрес, функция может модифицировать структуру, являющуюся ее фактическим параметром.

Для того чтобы определить адрес структуры, достаточно поместить перед ее именем оператор &. Рассмотрим следующий фрагмент программы:

```
struct bal {
    float balance;
    char name[80];
} person;
struct bal *p;
```

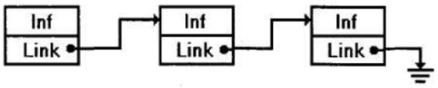
Теперь оператор p = &person; присваивает указателю p адрес структуры person.

Для того чтобы обратиться к элементу структуры через указатель на нее, нужно применить оператор "–>". Например, вот как выглядит ссылка на поле balance:

```
p->balance
```

Связанные динамические данные

Линейные списки – это данные динамической структуры, которые представляют собой совокупность линейно связанных однородных элементов, для которых разрешается добавлять элементы между любыми двумя другими, и удалять любой элемент.



Кольцевые списки – это такие же данные, как и линейные списки, но имеющие дополнительную связь между последним и первым элементами списка.

Очередь – частный случай линейного односвязного списка, для которого разрешены только два действия: добавление элемента в конец (хвост) очереди и удаление элемента из начала (головы) очереди.

Стек – частный случай линейного односвязного списка, для которого разрешено добавлять или удалять элементы только с одного конца списка, который называется вершиной (головой) стека.

Деревья – это динамические данные иерархической структуры произвольной конфигурации. Элементы дерева называются вершинами (узлами).

Пирамидой (упорядоченным деревом) называется дерево, в котором значения вершин (узлов) всегда возрастают или убывают при переходе на следующий уровень.

Связанные динамические данные характеризуются высокой гибкостью создания структур данных различной конфигурации. Это достигается благодаря возможности выделять и освобождать память под элементы в любой момент времени работы программы и возможности установить связь между любыми двумя элементами с помощью указателей.

Для организации связей между элементами динамической структуры данных требуется, чтобы каждый элемент содержал кроме информационных значений как минимум один указатель. Отсюда следует, что в качестве элементов таких структур необходимо использовать записи, которые могут объединять в единое целое разнородные элементы.

В простейшем случае элемент динамической структуры данных должен состоять из двух полей: информационного и указательного.

Схематично такую структуру данных можно представить как на рисунке выше.

Соответствующие ей объявления будут иметь такой вид:

```
struct NODE {
    INFO info;
    struct NODE * next;
};
```

где тип INFO может быть произвольного типа.

Работа со стеком

Для работы со стеком необходимо иметь один основной указатель на вершину стека top и один дополнительный временный указатель prom, который используется для выделения и освобождения памяти элементов стека.

Пустой стек характеризуется тем, что основной указатель указывает на пустой адрес, то есть NULL. Таким образом в начале программы, работающей со стеком должна быть следующая строка:

```
struct NODE * top = NULL;
```

Для добавления элемента в стек, необходимо сначала выделить память под этот элемент, а затем внести значения в информационное поле нового элемента и установить связь между ним и "старой" вершиной стека Top:

И последнее что надо сделать при добавлении элемента в стек, переместить вершину стека top на новый элемент.

```
#include <stdio.h>
#include <conio.h>
```

```
#include <stdlib.h>
typedef int INFO;
struct NODE {
     INFO info;
     struct NODE * next;
};
typedef struct NODE sNODE;
struct NODE * top = NULL;
void push(INFO val)
{
         sNODE * prom = (sNODE *)malloc(sizeof (sNODE));
         prom->info = val;
         prom->next = top;
         top = prom;
}

INFO pop(void)
{
       sNODE * prom = top;
       INFO val = prom->info;
       top = prom->next;
       free(prom);
       return val;
}

void show(void)
{
    struct NODE * prom = top;
    printf("Содержимое стека\n----------------\n");
    if (!prom) {
                printf("Стек пуст!\n");
                return;
         }
    while (prom) {
         printf("%d\n",prom->info);
         prom = prom->next;
         }
}
```

```
int main()
{
       INFO val;
       show();
         do {
              printf("Введите элемент (0 - конец ввода): ");
              scanf("%d",&val);
              if (!val) break;
              printf("-> %d\n",val);
              push(val);
      } while (1);
      show();
      val = pop();
      printf("<- %d\n",val);
      show();
      return 0;
}
```

Работа с очередью

Для создания очереди и работы с ней необходимо иметь как минимум два указателя:
– на начало очереди (возьмем идентификатор head);
– на конец очереди (возьмем идентификатор tail).

Кроме того, также, как и при работе со стеком, для добавления элементов и освобождения памяти удаляемых элементов требуется дополнительный временный указатель (возьмем идентификатор prom). Дополнительный указатель также часто используется в других ситуациях для удобства работы с очередью.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef int INFO;
struct NODE {
      INFO info;
      struct NODE * next;
};
typedef struct NODE sNODE;
struct NODE * head = NULL, * tail = NULL;

void add(INFO val)
{
       sNODE * prom = (sNODE *)malloc(sizeof(sNODE));
       prom->info = val;
       prom->next = NULL;
       if (!tail) head = prom;
       else tail->next = prom;
       tail = prom;
}

INFO del(void)
{
       INFO val = head->info;
       sNODE * prom = head;
       head = prom->next, free(prom);
       return val;
}
```

```
void show(void)
{
    struct NODE * prom = head;
    printf("Содержимое очереди\n----------------\n");
    if (!prom) {
                printf("Очередь пуста!\n");
                return;
        }
    while (prom) {
        printf("%d ",prom->info);
        prom = prom->next;
        }
    printf("\n");
}

int main()
{
       INFO val;
       show();
         do {
              printf("Введите элемент (0 - конец ввода): ");
              scanf("%d",&val);
              if (!val) break;
              printf("-> %d\n",val);
              add(val);
     } while (1);
     show();
     printf("<- %d ",del());
     show();
}
```

Линейные списки

Линейные списки отличаются тем, что элементы в список можно добавлять и удалять в любое место списка: начало, конец, после заданного элемента.

Мы будем рассматривать только линейные односвязные списки, поэтому для работы с ними необходим указатель на начало списка first, который в начале работы будет пустым:

```
struct NODE *first = NULL;
```

Также как и при работе со стеком и очередью, для добавления элементов и освобождения памяти удаляемых элементов требуется дополнительный указатель (prom).

Механизм добавления и удаления элемента в любую часть списка реализован в приведенной ниже программе. Следует отметить, что добавление и удаление последнего элемента предлагается реализовать с помощью добавления и удаления после заданного элемента.

В некоторых функциях используется функция getNode, которая принимает номер элемента и должна возвращать указатель на этот элемент.

Пример функции добавления в начало списка:

```
/*
функция принимает целочисленный параметр со значением элемента списка
*/
void addhead(int val)
{
       // создаётся дополнительная переменная – указатель на
структуру, которая является элементом списка.
       // также выделяется память под этот элемент
      sNODE * prom = (sNODE *) malloc(sizeof (sNODE));
      // записывается значение в поле значения…
      prom->info = val;
      // и указатель на первый элемент в поле указателя
      prom->next = first;
      // меняем указатель на первый элемент
      first=prom;
      // увеличиваем длину списка на 1
      length++;
```

```
}
```

Пример функции удаления последнего элемента списка (в качестве параметра передаётся длина списка – 1, то есть номер предпоследнего элемента), функция также подходит для удаления любого элемента списка после заданного:

```
/* функция принимает номер элемента и возвращает значение этого элемента, а также
удаляет заданный элемент
*/
int delnum(int num)
{
	// создаётся указатель и вызывается функция getNode, которая должна
возвращать указатель на элемент, перед искомым
	sNODE * predlast = getNode(num - 1);

	// сохраняем указатель на следующий элемент, чтобы не потерять его
при удалении
	sNODE * prom = predlast->next;

	// сохраняем информационную часть в переменную
	int val = prom->info;

	// меняем указатель для элемента, предшествующего искомому на
указатель на элемент, следующий за искомым
	predlast->next = prom->next;

	// уменьшаем длину списка на 1
	length--;

	// «удаляем» элемент – освобождаем выделенную под него память
	free(prom);

	// возвращаем сохранённое значение
	return val;
}
```

Пример функции отображения списка:

```
void show(void)
{
	sNODE * prom = first;
	printf("Содержимое списка: ");
	if (!prom) {
		printf("Список пуст!\n");
		return;
	}
	while (prom) {
		printf("%d ",prom->info);
		prom = prom->next;
	}
	printf("\n");
}
```

```
show();
puts("\n1. Добавить элемент в начало списка");
puts("2. Добавить элемент в конец списка");
puts("3. Добавить после заданного элемента");
puts("4. Удалить элемент из начала списка");
puts("5. Удалить элемент из конца списка");
puts("6. Удалить после заданного элемента");
puts("0. Выйти");
```

Пример функции, возвращающей элемент по номеру (в качестве аргумента передаётся интересующий номер):

```
sNODE * getNode(int num)
    {
        sNODE * prom = first;
        int i;
        for (i = 0; i < num; i++)
            prom = prom->next;
        return prom;
    }
```

В данной функции создаётся дополнительный указатель (prom), который указывает на первый элемент списка (для того чтобы не потерять основной указатель на первый элемент). Далее в цикле с помощью цикла мы «отсчитываем» необходимое количество элементов перемещая указатель на следующий элемент. В результате после остановки цикла после заданного количества элементов указатель prom будет указывает на нужный нам элемент.

Задания

Разработать программу для работы с динамически связным списком из структур в соответствии с вариантом, поддерживающую следующие действия(функции):

1) Первоначальное заполнение списка без указания размера (окончание ввода определить самостоятельно, например, ввод 0 или -1, или другого символа)
2) Добавление элемента:
    a. в начало
    b. конец
    c. любое заданное место в списке
3) Удаление элемента
    a. из начала
    b. конца
    c. любого заданного места в списке.

   При удалении элемента должна освобождаться выделенная под него память.
4) Вывод списка на экран
5) Получение элемента списка по заданному номеру
6) Выполнение индивидуального задания в соответствии с вариантом

Программа должна выводить меню после каждого действия и завершаться только после ввода пользователя (можно сделать дополнительный пункт меню «выход», по выбору которого завершать программу).

Варианты:

Вариант	Задание
1	Описать структуру с именем Student, содержащую следующие поля: • Name – фамилия и инициалы; • Group – номер группы; • Sess - успеваемость (массив из пяти элементов). Индивидуальное задание: • вывод на дисплей фамилий и номеров групп для всех студентов, включенных с клавиатуры, если средний балл студента больше 4,0;
2	Описать структуру с именем Student, содержащую следующие поля:

	• Name - фамилия и инициалы; • Group- номер группы; • Sess- успеваемость (массив из пяти элементов). Индивидуальное задание: • вывод на дисплей фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2;
3	Описать структуру с именем Worker, содержащую следующие поля: • Name — фамилия и инициалы работника; • Pos — название занимаемой должности; • Year — год поступления на работу. Индивидуальное задание: • вывод на дисплей фамилий работников, чей стаж работы в организации превышает значение, введенное с клавиатуры;
4	Описать структуру с именем Train, содержащую следующие поля: • Nazn — название пункта назначения; • Numr — номер поезда; • Time — время отправления. Индивидуальное задание • вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени;
5	Описать структуру с именем Marsh, содержащую следующие, поля: • Begst — название начального пункта маршрута; • Term — название конечного пункта маршрута; • Number — номер маршрута. Индивидуальное задание: • вывод на экран информации о маршруте, номер которого введен с клавиатуры;
6	Описать структуру с именем Note, содержащую следующие поля: • Name — фамилия, имя; • Tele — номер телефона; • BDay — день рождения (массив из трех чисел). Индивидуальное задание: • вывод на экран информации о человеке, номер телефона которого введен с клавиатуры;
7	Описать структуру с именем Znak, содержащую следующие поля: • Name — фамилия, имя; • Zodiak — знак Зодиака; • BDay — день рождения (массив из трех чисел). Индивидуальное задание: • вывод на экран информации о людях, родившихся в месяц, значение которого введено с клавиатуры;
8	Описать структуру с именем Order, содержащую следующие поля: • Plat — расчетный счет плательщика; • Pol — расчетный счет получателя; • Summa — перечисляемая сумма в руб. Индивидуальное задание: • вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры;
9	Описать структуру с именем Order, содержащую следующие поля: • Plat — расчетный счет плательщика;

	• Pol — расчетный счет получателя; • Summa — перечисляемая сумма в руб. Индивидуальное задание: • вывод на экран информации о том откуда и кому переведены суммы, больше введенной с клавиатуры;
10	Описать структуру с именем Note, содержащую следующие поля: • Name — фамилия, имя; • Tele — номер телефона; • BDay — день рождения (массив из трех чисел). Индивидуальное задание: • вывод на экран информации о людях, чьи дни рождения приходятся на месяц, значение которого введено с клавиатуры;
11	Описать структуру с именем Order, содержащую следующие поля: • Plat — расчетный счет плательщика; • Pol — расчетный счет получателя; • Summa — перечисляемая сумма в руб. Индивидуальное задание: • вывод на экран информации о сумме, переведенный на расчетный счет получателя, введенного с клавиатуры;
12	Описать структуру с именем Aeroflot, содержащую следующие поля: • Nazn – название пункта назначения рейса; • Numr – номер рейса; • Tip – тип самолета. Индивидуальное задание: • вывод на экран номеров рейсов и типов самолетов, вылетающих в пункт назначения, название которого совпало с названием, введенным с клавиатуры;
13	Описать структуру с именем Price, содержащую следующие поля: • Tovar – название товара; • Mag – название магазина, в котором продается товар; • Stoim – стоимость товара в руб. Индивидуальное задание: • вывод на экран информации о товаре, название которого введено с клавиатуры;
14	Описать структуру с именем Znak, содержащую следующие поля: • Name — фамилия, имя; • Zodiak — знак Зодиака; • BDay — день рождения (массив из трех чисел). Индивидуальное задание: • вывод на экран информации о людях со знаком зодиака, совпадающим с введенным с клавиатуры;
15	Описать структуру с именем Price, содержащую следующие поля: • Tovar – название товара; • Mag – название магазина, в котором продается товар; • Stoim – стоимость товара в руб. Индивидуальное задание: • вывод на экран информации о товарах со стоимостью, введенной с клавиатуры;

Контрольные вопросы

1. Для чего нужны указатели?
2. Как создать массив динамического размера?
3. Опишите прототипы функций для динамической работы с памятью?
4. Как освободить динамически выделенную память?