**Лабораторная работа №1**

Перечисление. Объединение. STL-контейнеры. Массив. Дэк. Одно- и двухсвязный список. Вектор

**Цель:** изучить и применить на практике односвязные списки. Изучить принципы работы со структурированными типами данных – объединением, перечислением.

**Объединения**

*Объединение* ***Union*** – частный случай структуры – это поименованная совокупность данных разных типов, размещаемых в ***одной и той же области памяти*** (по одному и тому же адресу). Размер *Union* равен размеру *наибольшего элемента*. Как и в случае структуры, члены объединения могут иметь любой тип за исключением типов void.

Достоинствами union является экономия памяти. Недостаток – в каждый момент времени в переменной типа *union* может храниться значение *только одного поля* из этих типов данных и возможно использовать лишь значение этого элемента (компонента). Определяется это тем, что все поля начинаются с одного адреса и их значения перекрываются.

Синтаксическое определение объединения:

**union имя\_типа\_структуры {**

*<тип\_данных> <имя1>;*

*<тип\_данных> <имя2>;*

.

*<тип\_данных> <имяN>;*

**}**;

Например,

*union num {*

*int n;*

*double f;*

*};*

Определение объекта объединения:

***union******имя\_типа\_структуры*** *Имя\_объекта\_объединения;*

Например,

*union num primer;*

или

*num primer;*

Доступ к элементам объединения осуществляется так же, как и к элементам структур. Для доступа к элементу объединения используется оператор точка «•»:

*Имя\_объекта\_объединения* **. имя\_элемента** (имя1)**;**

Например,

*primer.n = 10;*

При объявлении переменной типа объединение её можно инициализировать значением, которое должно иметь тип первого члена объединения. Например,

*union num d = { 1 }; // правильно, d = 1*

*union num d = { 1.0 }; // ошибка*

Объединения одного типа можно присваивать друг другу. В этом случае оператор присваивания выполняет по членное копирование объединений. Например,

*union num e, g;*

*e = g;*

Также, как и структуры объединения нельзя сравнивать.

***Выводы:*** **Структуры** – когда удобно хранить одновременно несколько вещей вместе в одной коробке, **объединения** – когда в коробке в каждый момент хранится только одна из этих вещей.

Пример:

*#include <conio.h>*

*#include <stdio.h>*

***union Register32 {***

*struct {* //занимает 4 байта

*unsigned char byte1;* //занимает 1 байт

*unsigned char byte2;* //занимает 1 байт

*unsigned char byte3;* //занимает 1 байт

*unsigned char byte4;* //занимает 1 байт

*} bytes;*

*struct {* //занимает 4 байта

*unsigned short low;* //занимает 2 байта

*unsigned short high;* //занимает 2 байта

*} words;*

*unsigned dword;* //без знаковое int от 0 до 4 294 967 295

*};*

*typedef union Register32 EAX;* //*синоним*

*void main() {*

*EAX reg;*

*reg.dword = 0x0000C0FF;*

*printf(" dword \t%08x\n", reg.dword);*

*printf(" low word \t%04x\n", reg.words.low);*

*printf(" high word \t%04x\n", reg.words.high);*

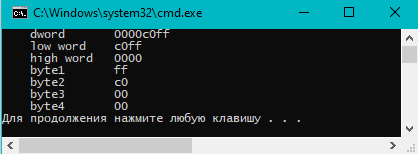
*printf(" byte1 \t%02x\n", reg.bytes.byte1);*

*printf(" byte2 \t%02x\n", reg.bytes.byte2);*

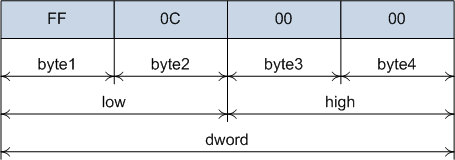
*printf(" byte3 \t%02x\n", reg.bytes.byte3);*

*printf(" byte4 \t%02x\n", reg.bytes.byte4);*

*}*



Здесь было создано объединение, которое содержит три поля – одно поле (*unsigned dword*) – целого типа (4 байта), (*words*) – два поля типа short int (2 байта каждое) и (*bytes*) – 4 поля по одному байту. После того, как значение было присвоено полю dword, оно также стало доступно и остальным полям.



На x86 байты располагаются справа налево (от младшего к старшим байтам). Все поля объединения обладают одинаковыми данными, но каждое поле имеет доступ только ко своей части.

Например:

*#include <iostream>*

*using namespace std;*

*union*

*{*

*short int name1;*

*int name2;*

*long int name3;*

*} myUnion;*

*int main()*

*{*

*myUnion.name1 = 22;*

*cout << myUnion.name1 << endl;*

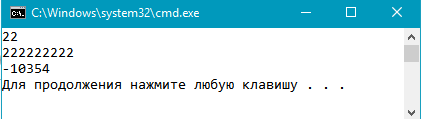
*myUnion.name3 = 222222222;*

*cout << myUnion.name3 << endl;*

*cout << myUnion.name1 << endl; // снова обращаемся к name1*

*}*

Запускаем программу:



После того, как записали значение в элемент **name3** типа **long int**, уже **нельзя** снова обратиться к элементу **name1.** Все потому, что в их ***общую память*** уже записано значение **long int**, а переменная типа **short int** неспособна работать с данными такого объема. Схематически можно это представить так:

[](http://purecodecpp.com/wp-content/uploads/2015/08/obiedineniia-c-union.jpg)

Поэтому, чтобы опять работать с данными типа **short int** необходимо снова присвоить элементу **name1** новое значение. В результате, получаем – память одна и та же, а переменные в ней размещаются разные. К какой обращаемся – такая и запрашивает из этой памяти значение.

Элементы объединения располагаются в памяти начиная с одного места (как бы накладываются друг на друга), в отличии от элементов структуры (они располагаются в памяти последовательно один за другим).

***Вывод:*** Применение объединений вызвано необходимостью экономии памяти, когда нужно хранить и использовать данные разных типов, но обращаться к ним можно не одновременно.

Пример: рассмотрим представление числа с плавающей точкой:

|  |  |
| --- | --- |
| *#include <conio.h>*  *#include <stdio.h>*  *union floatint{*  *float f;*  *int i;*  *};*  *void main() {*  *union floatint u = { 10.0 };*  *printf("%f\n", u.f);*  *printf("%x\n", u.i);*  *}* |  |
| *#include <conio.h>*  *#include <stdio.h>*  *union floatint{*  *float f;*  *int i;*  *};*  *void main() {*  *union floatint u = { 10.5 };*  *printf(" u.f = %f\n", u.f);*  *printf(" u.i = %d\n", u.i);*  *u.i = 7;*  *printf("\n u.f = %f\n", u.f);*  *printf(" u.i = %d\n", u.i);*  *getch();*  *}* |  |

**Перечисления ENUM**

***Перечислением*** ***enum*** называется тип данных, который включает множество именованных целочисленных констант. Именованные константы, принадлежащие перечислению, называются перечислимыми константами.

Таким образом Перечисления (***enum***) используются в С/C++ для создания констант. Допустим надо объявить константы для музыкальных нот и каждую инициализировать соответствующим порядковым номером. Можно воспользоваться обычным способом:

*const int DO = 1;*

*const int RE = 2;*

*const int MI = 3;*

*const int FA = 4;*

*const int SOL = 5;*

*const int LA = 6;*

*const int SI = 7;*

Такое объявление занимает много строк кода и не совсем удобно. Используя перечисление, можно определить эти константы иным способом. Синтаксис ***enum*** похож на синтаксис структур:

***ключевое слово – дескриптор – элементы в фигурных скобках через запятую***:

*enum notes {DO, RE, MI, FA, SOL, LA, SI}; // объявление типа*

Это полная форма – с дескриптором (именем перечисления). Где ***enum*** – ключевое слово, *notes* – имя типа перечисления, *DO, RE, MI,FA, SOL, LA, SI* – сами перечислимые константы.

Как и в случае со структурами, имя перечисления можно не использовать:

*enum {DO=1, RE, MI, FA, SOL, LA, SI};*

В первом случае – можно создавать переменные типа перечисления *notes*. Во втором случае – нет.

Если инициализация отсутствует, то перечислимым константам по умолчанию присваиваются последовательные значения: 0, 1, 2, … – *DO=0, RE=1, MI=2, FA=3, SOL=4, LA=5, SI=6.*

Обратите внимание, что в примере значение 1 было присвоено только первому элементу перечисления. Если явно задать значение, как в примере, то увеличение на единицу начнется с этого числа.

При объявлении типа перечисления его значения могут инициализироваться произвольными целочисленными константами или константным выражением.

*enum notes { DO = 2, RE =5, MI = 255, SOL = 0};*

Переменная типа перечисление объявляется следующим образом:

enum *notes* c; // объявление переменной c, которая имеет тип notes (язык С)

*notes* d; // объявление переменной d, которая имеет тип notes (язык С++)

Выведем на экран значения некоторых элементов перечисления.

*#include <iostream>*

*using namespace std;*

*enum notes{ DO = 1, RE, MI, FA, SOL, LA, SI };*

*void main()*

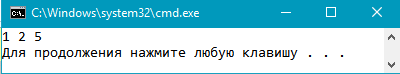
*{*

*setlocale(LC\_ALL, "rus");*

*cout << DO << ' ' << RE << ' ' << SOL << endl;*

*}*

В консоли отобразится:



Хотя не присваивали явно значения **RE** и **SOL** – они хранят правильные порядковые номера нот.

Имена перечислителей (элементов перечисления) должны быть уникальными. Значения же могут совпадать:

*enum champions{ Домрачева =* ***1****, Скардино =* ***1****, Цупер, Иванов };*

Значение 1 будут хранить *Домрачева* и *Скардино*. 2 запишется в *Цупер*, 3 – в *Иванов*. В элементы перечисления могут быть записаны только целые числа. Изменить ниже в программе значения, которые присвоены элементам в фигурных скобках нельзя. Если создается переменная типа перечисления – она может принять значение только одного из своих элементов:

*champions Dasha = 2222;* *//Ошибка*

*champions Dasha = Домрачева;*

**Записать число в переменную типа *champions* нельзя.** Можно записать только **именованную константу**, которая объявлена во время определения *enum*.

Приведем пример объявления типа цвет:

*enum color { r = 2, g = 2, b = 6 }; // объявление типа*

Если инициализация отсутствует, то перечислимым константам присваиваются последовательные значения: 0, 1, 2, … Например,

*enum color { r, g, b }; // r = 0, g = 1, b = 2*

Объявление типа перечисления и переменной, которая имеет этот тип, может быть объединено в одну инструкцию. Например,

|  |  |
| --- | --- |
| *enum turn { off, on };*  *turn a;* | *enum turn { off, on } a;*  *// а – переменная типа turn* |

Пример: присвоим переменным перечислимого типа значения. Например,

*color c = r; // правильно*

*color c = 0; // ошибка*

Любым целочисленным переменным можно присваивать значения перечислимых констант. Например,

*int primer = r; // правильно*

Пример. Создадим перечисление **levelsName**. Оно будет содержать шесть элементов – «названия» этажей. Пользователю предложим перемещаться по этажам на лифте.

*#include <iostream>*

*using namespace std;*

*enum levelsName { parking, bank, insurance, exchange, fitnesroom, Restaurant };*

*int main()*

*{*

*setlocale(LC\_ALL, "rus");*

*int floor = parking; // номер этажа– выбор пользователя*

*int exitOrNot = 9; // выйти или нет*

*while (exitOrNot == 9)*

*{*

*cout << "Номер этажа (0,1,2,3,4,5): ";*

*cin >> floor;*

*switch (floor)*

*{*

*case(parking) :*

*cout << "\a\a\aВы в паркинге!\n\n";*

*break;*

*case(bank) :*

*cout << "\a\a\aПервый этаж!";*

*cout << "\n Здесь находится отделение Беларусбанк.\n\n";*

*break;*

*case(insurance) :*

*cout << "\a\a\aВторой этаж!";*

*cout << "\n Страховая компания Белгосстрах.\n\n";*

*break;*

*case(exchange) :*

*cout << "\a\a\aТретий этаж!";*

*cout << "\n Белорусская валютно-фондовая биржа.\n\n";*

*break;*

*case(fitnesroom) :*

*cout << "\a\a\aЧетвёртый этаж!";*

*cout << "\n Фитнес центр.\n\n";*

*break;*

*case(Restaurant) :*

*cout << "\a\a\aПятый этаж!";*

*cout << "\nРесторан «7-е небо».\n\n";*

*break;*

*default: cout << "\a\a\aОшибка!\n\n";*

*}*

*cout << "Выйти из лифта - нажмите 0.\n";*

*cout << "Выбрать другой этаж - нажмите 9: ";*

*do*

*{*

*cin >> exitOrNot;*

*} while (exitOrNot != 0 && exitOrNot != 9);*

*cout << endl;*

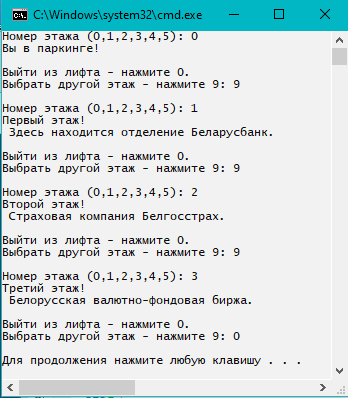
*}*

*return 0;*

*}*

Здесь объявляем целочисленную переменную **floor** и инициализируем ее именованной константой **parking**. Так **floor** примет значение 0. В цикле While – пока пользователь не введет в переменную **exitOrNot** значение 0, программа будет ему предлагать выбрать номер этажа. После каждого выбора этажа на экран выводится номер этажа и информация о том, что на нём размещено. Это реализовано через оператор выбора switch. Его блоки **case** перебирают элементы перечисления. То есть пользователь вводит целое число, а **case** ищет, какому из элементов перечисления оно соответствует. Например, если введено 5 – это соответствует значению элемента **Restaurant** – на экран выводится соответствующее сообщение.

Результат:

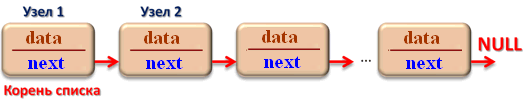


**Список** представляет собой линейную последовательность переменных, каждая из которых связана указателями со своими соседями. Списки бывают *односвязные* (каждый элемент имеет указатель на следующий); *двусвязные* (каждый элемент имеет указатель на следующий и на предыдущий); *двусвязные циклические* (первый и последний элементы ссылаются друг на друга).

**Связный список** – это динамическая структура данных, состоящая из узлов, которые содержат в себе в классическом варианте два значения: первое – это какое-либо данное (этим данным может быть: обычная переменная, объект класса, структура и так далее), а второе – это указатель на следующий узел в списке. Таким образом все узлы списка связаны между собой с помощью указателей.

Динамически во время выполнения программы можно расширять структуру списка путем добавления новых узлов в список. В отличие от массива будь то статического, либо динамического, динамический список можно увеличивать *во время работы программы*.

Рассмотрим схему списка на рисунке 1. Каждый узел однонаправленного (односвязного) линейного списка (ОЛС) содержит одно поле указателя на следующий узел. Поле указателя последнего узла содержит нулевое значение (указывает на NULL).



**Рисунок 1 – Схема списка**

На схеме представлены узлы, содержащие в себе два значения – данное ***data*** и указатель ***next***. Указатель (содержит в себе адрес памяти) всегда указывает на следующий узел связанного списка.

**Внимание:** Указатель последнего узла должен всегда выставляться в ***нуль*** (***NULL***, nullptr или просто 0). Это говорит о том, что данный узел является последним узлом связанного списка и что больше элементов нет. При добавлении нового узла в динамический список значение NULL заменяется на адрес нахождения в памяти нового узла, а сам указатель нового узла опять выставляется в NULL.

По ходу работы программы можно сколь угодно создавать таких узлов связанного списка, единственное ограничение накладывает размер свободной оперативной памяти компьютера.

На схеме представлена логическая структура списка, в которой все узлы расположены по порядку. Реально, в памяти компьютера, узлы могут располагаться в любом месте оперативной памяти. Найти каждый узел списка несложно, т.к. в каждом узле поле-указатель содержит адрес следующей ячейки памяти.

**Работа со связанным списком**

Опишем структуру, представляющую собой **узел** динамического списка. Эта структура состоит из двух полей – данное и указатель на структуру того же типа. Такие структуры еще называются самоссылающимися (или структуры с самоадресацией):

***struct Node***

***{***

***int data;*** // поле данных

***Node \*next;*** // указатель на следующий элемент

***};***

Далее создам сам **объект «связанный список»**, который будет хранить эти самоссылающиеся узлы (Node – с англ. узел):

class **List** //класс, описывающих объект "связанный список"

{

private:

**Node** \*head; //"голова" связанного списка

public:

List() //конструктор класса без параметров

{

head = NULL; //первого элемента пока нет

}

//метод, добавляющий новый узел в список

void addNode(int d);

//метод, выводящий связанный список на экран

void printList();

};

Рассмотрим более подробно структуру ***List***, представляющую «связанный список». Так как динамический список состоит из узлов, значит класс ***List*** должен манипулировать этими узлами: создавать их, удалять, выводить на печать и так далее. В классе ***List*** имеются два метода: ***addNode()*** – создает новый узел в динамическом списке, ***printList()*** – выводит содержание списка (всех узлов по очереди) на печать. Есть закрытое поле класса ***head*** типа Node – это так называемая «голова» связанного списка, которая должна всегда указывать на начало списка в памяти компьютера, т.е. на его ***первый узел – указатель, указывающий просто на начало динамического списка и не содержащий никаких данных.*** Зная, где начинается связанный список, на него нам указывает «голова» head, и где заканчивается, об этом нам сообщает указатель последнего узла, выставленный в NULL, можно перемещаться по всему динамическому списку и выполнять с узлами необходимые операции.

В конструкторе класса ***List*** переменной ***head*** выставляется значение в NULL, т.к. при создании объекта класса ***List*** связанный список еще пуст и узлов в нем нет, соответственно и указывать не на что.

Представим код программы, реализующей динамический список.

*#include <iostream>*

*using namespace std;*

***struct Node***

***{***

***int data;***

***Node \*next;***

***};***

*class List*

*{*

*private:*

*Node* ***\*head****; //"голова" связанного списка*

*public:*

*List() //конструктор класса без параметров*

*{*

*head = NULL; //первого элемента пока нет*

*}*

*//метод, добавляющий новый узел в список*

*void addNode(int d)*

*{*

*Node \*nd = new Node; //динамически создаем новый узел*

*nd->data = d; //задаем узлу данные*

*nd->next = NULL; //новый узел в конце, поэтому NULL*

*if (head == NULL) //если создаем первый узел*

*head = nd;*

*else //если узел уже не первый*

*{*

*Node \*current = head;*

*//ищем в цикле предшествующий последнему узел*

*while (current->next != NULL)*

*current = current->next;*

*//предшествующий указывает на последний*

*current->next = nd;*

*}*

*}*

*//метод, выводящий связанный список на экран*

*void printList()*

*{*

*Node \*current = head;*

*while (current != NULL)*

*{*

*cout << current->data << endl;*

*current = current->next;*

*}*

*}*

*};*

*int main()*

*{*

***List myList;***

*myList.addNode(25);*

*myList.addNode(20);*

*myList.addNode(15);*

*myList.addNode(10);*

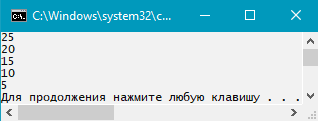
*myList.addNode(5);*

*myList.printList();*

*return 0;*

*}*

Результат работы программы:



**Рисунок 2 – Результат выполнения программы**

Рассмотрим подробнее метод *addNode( )*, добавляющий новый узел в список. Первая строка кода *Node \*nd = new Node;* динамически создает новый объект типа *Node*, т.е. новый узел. После выполнения данной строки, в указатель ***nd***, при успешном создании объекта, записывается адрес созданного объекта в памяти, в неудачном случае будет записано NULL, т.е. объект не создан.

Во 2 и 3-й строках полям вновь созданного узла присваиваются необходимые значения:

*nd->data = d; –* задаем данное – это данное, которое метод   
*void addNode(int d)* принимает в качестве аргумента;

*nd->next = NULL;* – выставляем указатель в NULL, т.к. вновь созданный узел всегда будет последним (в данном варианте программы узлы добавляются в конец связанного списка – классический вариант).

Следующая конструкция выбора служит для определения: создается первый узел в списке или он уже не первый:

*if (head == NULL) //если создаем первый узел*

*head = nd;*

*else //если узел уже не первый*

*{*

*………*

*}*

В случае, если создается только первый узел, то выполнится условие после if – *head* устанавливается в *NUL*, т.е. head присвоится адрес первого узла в памяти. Далее, когда создаются последующие узлы, будут выполняться команды в блоке else:

*else //если узел уже не первый*

***{***

*Node \*current = head;* – создается временная переменная – указатель типа *Node* и присваивается ей указатель на начало списка.

*while (current->next != NULL) –* в цикле последовательно перебираются

*current = current->next; –* все узлы связанного списка до тех пор,

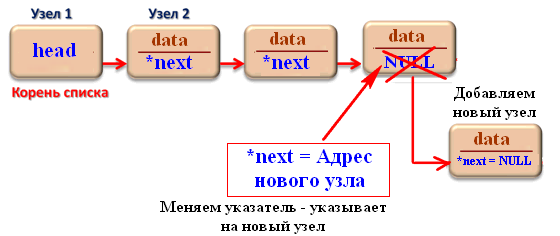
пока не дойдем до последнего созданного узла.

*current->next = nd;*  – присваиваем указателю полученного последнего

узла (был NULL) адрес вновь созданного (нового) узла.

***}***

Описанный процесс показан на рисунке 3.



**Рисунок 3 – Схема добавления элемента в список**

Рассмотрим более подробно действия, выполняемые с односвязным линейным списком (ОЛС).

**Основные действия, производимые над элементами односвязного списка**

Основные действия, производимые над элементами ОЛС:

* инициализация списка;
* добавление узла в список;
* удаление узла из списка;
* удаление корня списка;
* вывод элементов списка;
* взаимообмен двух узлов списка.

**Двунаправленные (двусвязные) списки**

**Двунаправленный (двусвязный) список** – это динамическая структура данных, состоящая из последовательности элементов, каждый из которых содержит (рисунок 1):

* поле – информационное;
* два поля с указателями на соседние элементы: одно поле содержит ссылку на следующий элемент, другое поле – ссылку на предыдущий элемент.

При этом два соседних элемента должны содержать взаимные ссылки друг на друга.

Наличие ссылок на следующее звено и на предыдущее позволяет двигаться по списку от каждого звена в любом направлении: от звена к концу списка или от звена к началу списка, поэтому такой список называют двунаправленным.

Описание простейшего элемента такого списка выглядит следующим образом:

*struct имя\_типа {*

*информационное поле;*

*адресное поле 1;*

*адресное поле 2;*

*};*

где информационное *поле* – это поле любого, ранее объявленного или стандартного, типа;

*адресное поле 1* – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес *следующего элемента списка* ;

*адресное поле 2* – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес *предыдущего элемента списка*.

Обобщаем: каждый элемент (кроме первого и последнего) связан с предыдущим и следующим за ним элементами.



**Рисунок 1 – Двунаправленный список**

Рассмотрим пример сииска:

*struct list*

*{*

*int field; // поле данных*

*list \*next; // указатель на следующий элемент*

*list \*prev; // указатель на предыдущий элемент*

*};*

*list \*headlist;*

где ***field*** – тип информационного поля элемента списка;

***\*next***, ***\*prev*** – указатели на следующий и предыдущий элементы этой структуры соответственно.

Переменная-указатель ***headlist*** задает список как единый программный объект, ее значение – указатель на первый (или заглавный) элемент списка.

Пример списка рассылки, для двусвязного списка структуру address:

*struct address {*

*char name[40];*

*char street[40];*

*char city[20];*

*char state[3];*

*char zip[11];*

*struct address \*next;*

*struct address \*prior;*

*} info;*

Основные операции, выполняемые над двунаправленным списком, те же, что и для однонаправленного списка. Так как двунаправленный список более гибкий, чем однонаправленный, то при включении элемента в список, нужно использовать указатель как на элемент, за которым происходит включение, так и указатель на элемент, перед которым происходит включение. При исключении элемента из списка нужно использовать как указатель на сам исключаемый элемент, так и указатели на предшествующий или следующий за исключаемым элементы. Но так как элемент двунаправленного списка имеет два указателя, то при выполнении операций включения/исключения элемента надо изменять больше связей, чем в однонаправленном списке.

Рассмотрим основные операции, осуществляемые с двунаправленными списками, такие как:

* создание списка;
* печать (просмотр) списка;
* вставка элемента в список;
* удаление элемента из списка;
* поиск элемента в списке;
* проверка пустоты списка;
* удаление списка.

Особое внимание следует обратить на то, что в отличие от однонаправленного списка здесь нет необходимости обеспечивать позиционирование какого-либо указателя именно на первый элемент списка, так как благодаря двум указателям в элементах можно получить доступ к любому элементу списка из любого другого элемента, осуществляя переходы в прямом или обратном направлении. Однако по правилам хорошего тона программирования указатель желательно ставить на заголовок списка.

**ЗАДАНИЯ**

**Студенты**. Ф.И.О., дата поступления, специальность, группа, факультет, средний балл. **Выбор по** году поступления, фамилии, среднему баллу, группе. Факультет реализовать с помощью перечисления.