Методические указания к лабораторной работе № 8 по курсу
ОСНОВЫ ПРОГРАММИРОВАНИЯ
ГУИМЦ

"Списки - структуры данных в СИ"
(8 часов) -

Москва, МГТУ, СУЦ - 2023 год

СОДЕРЖАНИЕ

ОП ГУИМЦ 2023 ЛР№9 Большаков С.А.

1. Цель лабораторной работы № 8 по дисциплине ОП (Основы программирования) - СУЦ

Целью данной ЛР по дисциплине ОП является получение навыков работы со списками и списковыми структурами при программировании на языке СИ. Студенты используют консольные проекты и отлаживают программы в среде программирования MS VS 2005/2008/2010. Студенты знакомятся с основными операциями при работе со списками, способами их заполнения, распечатки, сортировки, проверяют работу отлаженных примеров и делают контрольные задания. Они выполняют отладку программы по своему варианту и получают исполнимую программу, готовую к выполнению, оформляют отчет по ЛР и защищают его.

2. Порядок выполнения лабораторной работы

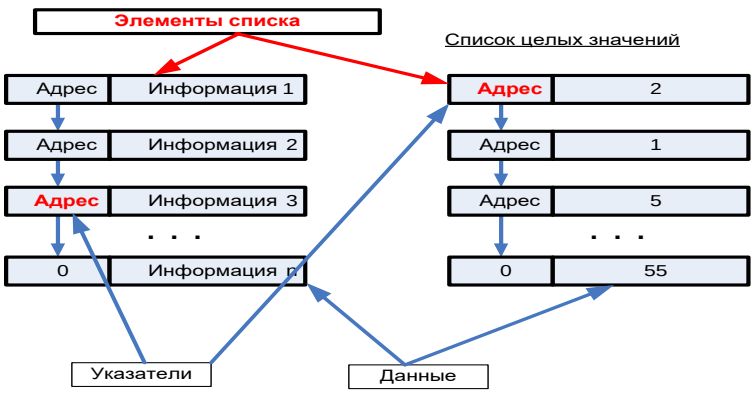
1. Познакомиться с содержанием методических указаний и основными понятиями ЛР (разделы 3 и 4)
2. Проработать порядок выполнения работы (раздел 5).
3. Создать консольные проекты для проверки примеров и выполнения задания ЛР(разделы 3,4).
4. Проверить в данном проекте примеры из методических указаний, выполнив их в отладчике в пошаговом режиме (раздел 4).
5. Написать программу заданий ЛР по варианту, выданному преподавателем и отладить ее (раздел 5).
6. Продемонстрировать работу программы преподавателю в режиме отладчика по шагам и изменяемыми переменными.
7. Подготовить отчет по ЛР по представленному шаблону (раздел 8).
8. Защитить ЛР с предоставлением отчета и ответами на контрольные вопросы (раздел 8,9).
9. Для продвинутых студентов выполнить задания для дополнительных (необязательных) требований и также отобразить их в отчете по ЛР (раздел 7).
10. Подготовить фрагменты программ для выполнения ДЗ по дисциплине ОП для своего варианта по списку.

3. Основные понятия

В теоретической части описания лабораторной работы вводятся основные понятия и рассматриваются принципы для работы со списками на языке программирования СИ.

3.1. Понятие список

Список – это структура данных, в которой данные расположены в определенной последовательности, порядок в которой задается с помощью адресации (указателей). Для этой цели используются переменные типа указатель. Каждый фрагмент, хранящий данные (элемент списка) должен содержать такой указатель и непосредственно информацию списка. На рисунке пред-

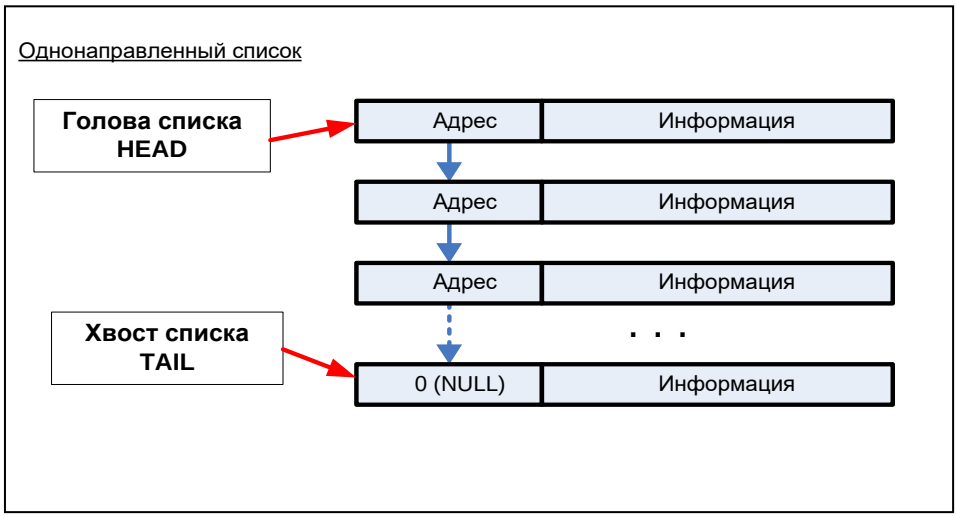


ставлен список, содержащий в качестве данных целые значения (простые переменные). Более подробно структуру списка рассмотрим ниже. Отдельный элемент хранения (на рисунке вытянутый прямоугольник) называется элементом списка. Отдельный элемент списка описывается с помощью структурной переменной или объекта (в СИ++). Для примера, простейший элемент списка может на СИ выглядеть так (информация – простая целая переменная **ListVal**):

```
// Структура самого простого элемента списка
struct ListElem{
ListElem * pNext; // адрес следующего элемента списка (Заметьте, указатель имеет тип (Elem *))
int ListVal; // информация, содержащаяся в списке
  };
  ...
// Описание и инициализация элемента LFirst
ListElem LFirst = { NULL , 3 }; // NULL – нулевой указатель – нет ссылки на другие элементы
// Значение ListVal = 3
СОГЛАШЕНИЕ: Адрес последнего элемента списка равен NULL!!! (нулю)
```

3.2. Голова и хвост списка

Понятия, связанные со списками:
ГОЛОВА СПИСКА (HEAD - первый элемент списка, его адрес)
ХВОСТ СПИСКА (TAIL – последний элемент списка, его адрес).

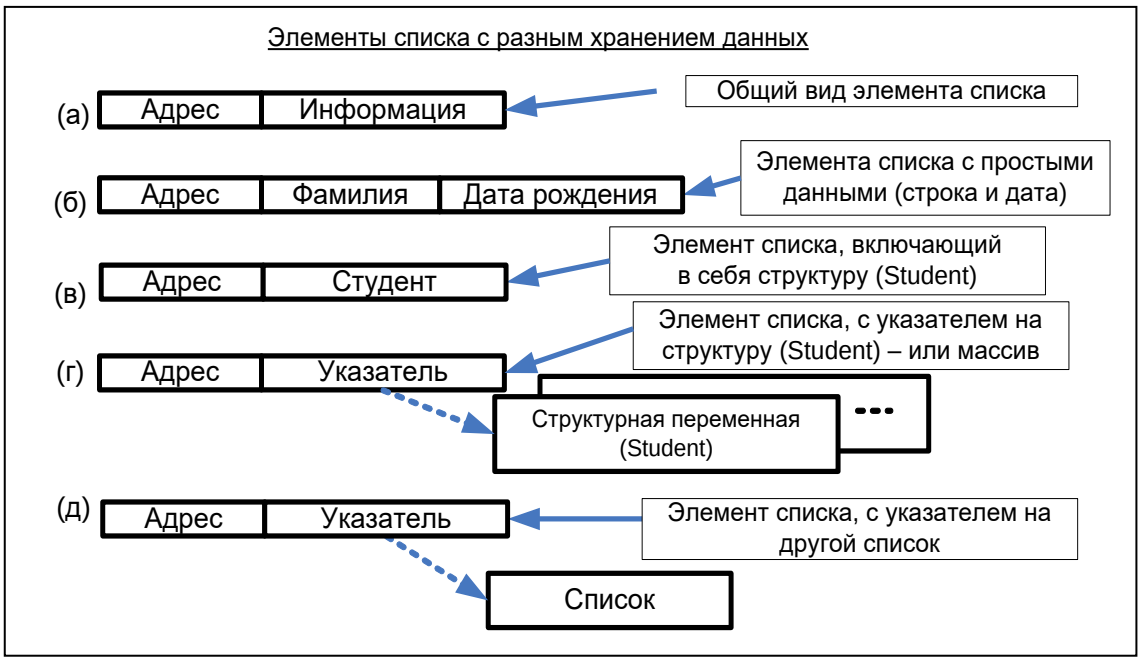


3.3. Особенности структур списков и их назначение

Из определения списка (см. выше) следует, что они предназначены для совместного хранения взаимосвязанной информации. В списки легко добавлять информацию и удалять ее. Список легко очистить и заполнить новыми данными. В качестве информации содержащейся в списке могут быть:

- Простые переменные стандартного типа (int, char , float и т.д.);
- Группы простых переменных (рис. (б));
- Структурные переменные (рис. (в));
- Указатели на структурные переменные (рис. (г));
- Массивы простых и структурных переменных и указатели на них (рис. (г));
- Указатели на другие списки (рис. (д));

На рисунке представлены разные варианты построения элементов списков. Поясним содержание рисунка.



Вариант (а) представляет общий вид отдельного элемента списка. Вариант (б) показывает список с простыми переменными разного типа (Фамилия и дата рождения). Вариант (в) описывает элемент списка с вложенной в него структурой (Student). Вариант (г) иллюстрирует использование указателя на структуру или массивы структур, а вариант (д) может быть использован для построения сложных структур данных со ссылками на списки, например деревьев.

3.4. Особенности списков по сравнению с массивами

Встает вопрос, у нас уже есть структуры данных типа массив, зачем нам понадобились списки? В чем-то массивы оказываются неэффективными! В первую очередь недостатки массивов заключаются в том, что трудно вставлять новые элементы в массивы и удалять существующие элементы из массива. Кроме этого, несмотря на динамические возможности (использование динамической памяти) в любой момент времени размер массива должен быть фиксированным. Списки позволяют более эффективно использовать память компьютера. Списки позволяют более полно использовать механизмы динамической памяти: и список и его элементы могут быть динамическими. Эти ограничения снимаются с использованием структур типа список, хотя добавляются и новые недостатки, в частности: трудно получить доступ к элементу списка по номеру. Самым важным преимуществом списков является возможность хранения разнотипных структурных переменных, если использовать для адресации списков указатели типа **void**. Эти особенности и проблемы мы рассмотрим ниже.

3.5. Простейший список, созданный вручную

В принципе, для построения списка достаточно только отдельных элементов списка и организации связи между ними. Это можно пояснить на примере:

```
// Структура самого простого элемента списка
struct Elem{
```

```
    Elem * pNext; // адрес следующего элемента списка (Заметьте, указатель имеет тип (Elem *))
    int ListVal; // информация, содержащаяся в списке
    };
        …
    // Описание и инициализация элементов списка – трех:
    Elem LE3 = { NULL , 3 }; // NULL – нулевой указатель – нет ссылки на другие элементы
    Elem LE2 = { &LE3 , 2 }; // &LE3 - задание адреса третьего элемента в pNext
    Elem LE1 = { &LE2 , 1 }; // &LE2 - задание адреса второго элемента в pNext
```

С таким списком уже можно работать, например, его распечатать:

```
    // Печать
printf ("Печать списка в цикле: \n" );
Elem * pETemp = &LE1; // Во временную переменную - указатель задаем адрес первого элемента
while ( pETemp != NULL)
{
printf ("Элемент простого списка Elem: %d \n" , pETemp->ListVal);
pETemp = pETemp->pNext; // НАВИГАЦИЯ: Важнейший оператор для работы со списком
};
```

Результат работы фрагмента программы:

```
Печать списка в цикле:
Элемент простого списка Elem: 1
Элемент простого списка Elem: 2
Элемент простого списка Elem: 3
```

Рассмотрим также пример ручного связывания списка:

Описание трех незаполненных элементов:

```
    // Ручное связывание и распечатка списка
    Elem E11; // Первый элемент без инициализации
    Elem E21; // Второй элемент без инициализации
    Elem E31; // Третий элемент без инициализации
    E11.ListVal = 10;
    E11.pNext = &E21;// Ссылается на 2-й элемент
    E21.ListVal = 20;
    E21.pNext = &E31; // Ссылается на 3-й элемент
    E31.ListVal = 30;
    E31.pNext = NULL; // не ссылается ни на кого
    // Печать
    Elem ETemp = {&E11 ,NULL };
    printf ("Содержимое ручного списка: \n");
    while (ETemp.pNext != NULL )
    {
    printf ("Элемент  = %d \n", ETemp.pNext ->ListVal );
    ETemp.pNext = ETemp.pNext ->pNext ; // Навигация
    };
    //
```

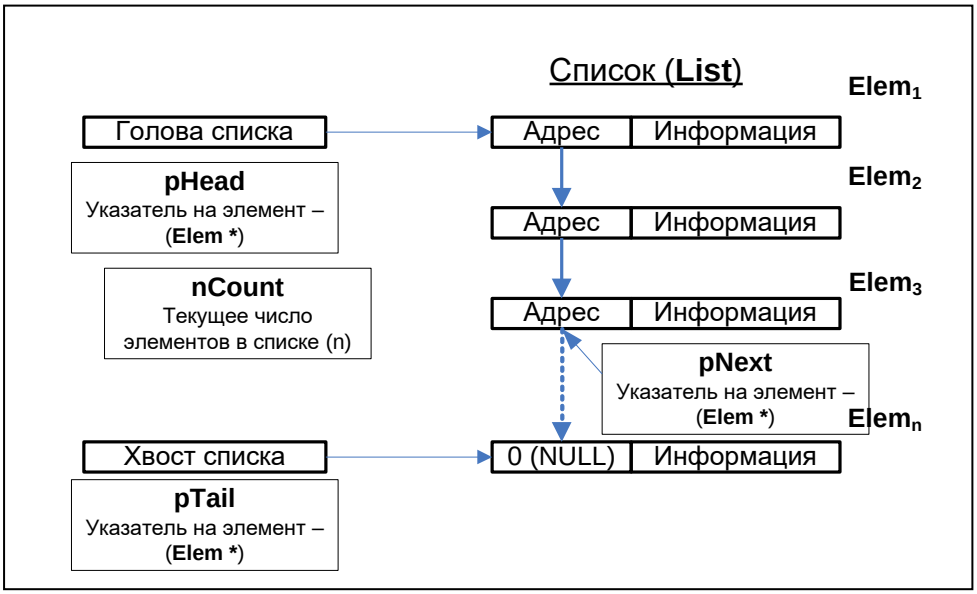
Результат работы фрагмента программы:

```
Содержимое ручного списка:
Элемент  = 10
Элемент  = 20
Элемент  = 30
```

3.5.1. Структуры списков и связь элементов списков

Однако при программировании возникает необходимость нахождения первого элемента списка для начала работы со списком. То есть нужно хранить адрес первого элемента в отдельной переменной. Такая переменная, содержащая адрес первого элемента списка часто называет-

ся головой списка. Она может быть задана типом либо указатель на элемент (**Elem * - pHead**), либо задаваться самим типом элемент списка (**Elem Head**). Выбор способа задания головы списка часто зависит от характера решаемой задачи и удобства работы со списком. В некоторых задачах со списками, необходимо знать какой из элементов списка является последним (хвост). В частности это важно для добавления в конец списка (в хвост) или организации двунаправленных списков (о них речь пойдет позднее). Для этого предусматривают также специальный указатель (**Elem * - pTail)** или специальную переменную - элемент списка (**Elem Tail**). Кроме этого, в некоторых случаях важно знать текущее число элементов списка (списки – динамические структуры данных). Для этого можно ввести специальную переменную целого типа (**nCount**). Для связи списков используется в отдельных элементах специальное поле – указатель на следующий элемент (**Elem * - pNext**). На рисунке, представленном ниже показаны рассмотренные выше элементы списков:



Таким образом, для описания отдельного списка целесообразно выделить специальную структуру данных типа:

```
struct EList{  // Простейшая структура список Elem - элементарный список
Elem * pHead; // Голова списка
Elem * pTail; // Хвост списка
int Count; // Счетчик элементов
};
```

Тогда, в соответствии с описаниями предыдущего примера можно описать сданный список так:

```
//Описание и нициализация простого списка
EList FirstList = { &LE1 , &LE3 , 3}; // 3 - Число элементов в списке
// Печать этого списка
printf ("Печать списка FirstList в цикле: \n" );
pETemp = FirstList.pHead; // Во временную переменную - указатель задаем адрес головы
спика
while ( pETemp != NULL) // Проверка прохождения последнего элемента списка
{
printf ("Элемент простого списка FirstList: %d \n" , pETemp->ListVal);
pETemp = pETemp->pNext; // НАВИГАЦИЯ: Важнейщий оператор для работы со списком
};
```

Результат распечатки такого списка:

```
Печать списка FirstList в цикле:
Элемент простого списка FirstList: 1
```

```
Элемент простого списка FirstList: 2
Элемент простого списка FirstList: 3
```

Важно в дополнение отметить следующее. Данный список является однонаправленным: движение по списку возможно только в одном направлении – от головы к хвосту, так заданы указатели. У последнего элемента списка (хвоста списка), по стандартному соглашению, указатель-адрес задается равным нулю (0, NULL – константа этапа компиляции). Это позволяет проверить завершение цикла печати. В другом случае завершение цикла можно было бы проверить, используя адрес хвоста списка (**pTail**), сравнив его с адресом текущего элемента для печати. Особое внимание уделим выделенной красным цветом строчке с комментарием НАВИГАЦИЯ. В этом операторе присваивания мы с помощью одного указателя (**pETemp**) на текущий элемент списка формируется указатель на следующий элемент списка (**pETemp->pNext**), результат запоминается в первом указателе (**pETemp**). Такой оператор обеспечивает навигацию по списку. Он будет многократно встречаться в наших примерах.

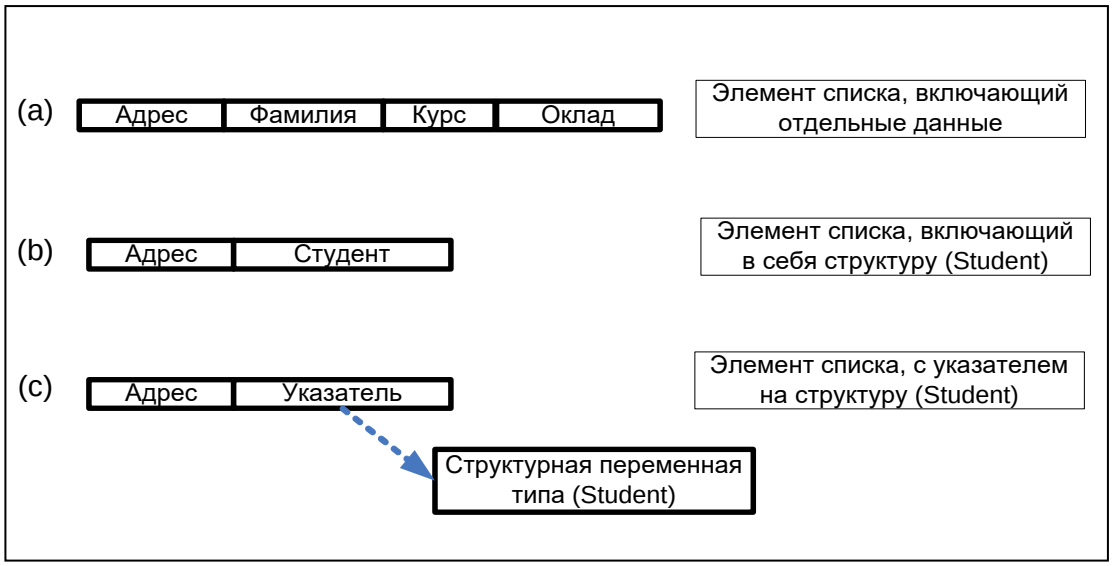
```
pETemp = pETemp->pNext;
```

3.6. Структура элемента списка с вложенными и внешними данными

Выше было отмечено, что информация в элементах списка может храниться разными способами:

- Данные записываются в самой структуре элемента в виде отдельных переменных
- Данные записываются в самой структуре элемента в виде структурной переменной
- Данные записываются в структуре элемента в виде указателя на структурную переменную, память под которую выделяется динамически.

На рисунке, представленном ниже, показаны эти варианты.



Примерами вариантов таких структур могут служить следующие:

```
// Структура для демонстрации
struct Student {
 char Name[20];
 int Num;
 float Oklad;
};
// Структура типа (a)
```

```
struct aElem{
aElem * pNext; // адрес следующего элемента списка
char Name[20]; // информация фамилии, содержащаяся в списке
int Num; // информация о номере - курсе, содержащаяся в списке
float Oklad; // информация об окладе, содержащаяся в списке
};
// Структура типа (b)
struct SNode{
SNode * pNext; // адрес следующего элемента списка
Student Stud; // Структура Student включенная в элемент списка
};
// Структура типа (c)
struct cElem{
cElem * pNext; // адрес следующего элемента списка
Student * pStud; // Указатель на структуру типа Student
};
```

Примеры работа с такими структурами элементов списка:

```
SNode SN3 = {NULL , {"Коля", 3 , 30.0f}};
SNode SN2 = {&SN3 , {"Петя", 2 , 20.0f}};
SNode SN1 = {&SN2 , {"Ваня", 1 , 10.0f}};
// Печать
SNode * pETemp = &SN1; // Во временную переменную - указатель на 1-й элемент
while ( pETemp != NULL) // Проверка прохождения последнего элемента списка
{
printf ("Элемент списка в цикле:Name - %s  Num -  %d   Oklad - %f \n" , pETemp->Stud.Name
, pETemp->Stud.Num , pETemp->Stud.Oklad);
pETemp = pETemp->pNext; // НАВИГАЦИЯ: Важнейший оператор для работы со списком
};
```

Результат распечатки такого списка:

```
Элемент списка в цикле:Name - Ваня  Num -  1   Oklad - 10.000000
Элемент списка в цикле:Name - Петя  Num -  2   Oklad - 20.000000
Элемент списка в цикле:Name - Коля  Num -  3   Oklad - 30.000000
```

3.7. Однонаправленные и двунаправленные списки

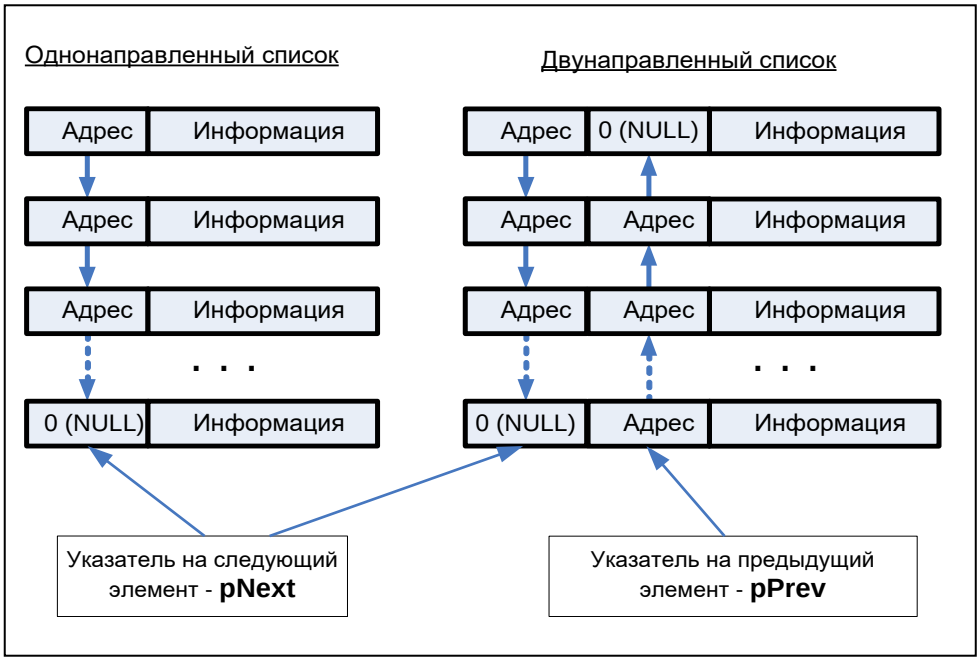
Более удобными и “быстродействующими”, по сравнению с однонаправленными списками, считаются двунаправленные списки. В них навигация по списку может осуществляться как в прямом, так и в обратном направлении. Для этого в структуре каждого элемента списка (**DElem**) предусмотрено два указателя (два адреса): указатель на следующий элемент (**pNext**) и указатель на предыдущий элемент (**pPrev**).

```
struct DElem{
DElem * pNext; // адрес следующего элемента списка (Заметьте, указатель имеет тип (Elem *))
DElem * pPrev; // адрес ghtlsleotuj элемента списка (Заметьте, указатель имеет тип (Elem *))
int ListVal; // информация, содержащаяся в списке
};
```

При этом соглашения для нулевого (NULL) последнего элемента списка (для **pNext**) и первого (для **pPrev**) справедливы. Для единственного элемента списка (он и первый и последний одновременно) справедлива такая инициализация двунаправленного элемента:

```
DElem D1 = {NULL,NULL, 5}; // Оба адреса-указателя равны нулю
```

На рисунке для сравнения представлены варианты организации однонаправленного и двунаправленного списков:



3.8. Статические и динамические списки

Списки могут быть статическими и динамическими. В первом случае и списковая структура и сами элементы списка должны быть описаны в программе предварительно. Заметим, оперативная память также для них выделяется предварительно. Примеры статических списков были рассмотрены в предыдущих разделах. После завершения работы удалять память для этих списков не надо, она освобождается автоматически при завершении программы.

Для динамических списков память под элементы, а, возможно, и под списковую структуру выделяется во время выполнения программы с помощью запросов к ОС (функции **malloc**, **calloc** и др.). После завершения работы выделенная память должна быть возвращена (функция **free**). Пусть для примера мы имеем такую структуру списка и его элементов

```
// Самый простой элемент списка
struct ListElem{
ListElem * pNext; // адрес следующего элемента списка
int ListVal; // информация, содержащаяся в списке
};
//  Структура простого списка
struct List {
ListElem Head; // Голова списка стуктурная переменная а не указатель
ListElem Tail; // Хвост списка
int Count; // Счетчик элементов
};
...
```

Для этих структур (элементов списка и самого списка) выделим динамическую память:

```
#include <malloc.h>
...
// ДИНАМИЧЕСКИЕ СПИСКИ
//Выделение памяти для списка
List * pDList = (List *) malloc (sizeof(List));
```

```
// Выделение памяти для одного элемента списка
ListElem * pTempElem = (ListElem * )malloc (sizeof(ListElem));
// Заполнение элемента списка данными
pTempElem->pNext = NULL; // Всего один элемент
pTempElem->ListVal = 5;
// Занесение элемннгта списка в голову списка
pDList->Head.pNext = pTempElem;
pDList->Tail.pNext = pTempElem;
// Чтение и печать значения первого динамического элемента динамического списка
printf ("Значение первого элемента: %d \n", pDList->Head.pNext->ListVal);
// Освобождение памяти и под элемент и список
free (pDList->Head.pNext);
free (pDList);
```

Результат работы этой программы:

```
Значение первого элемента: 5
```

Здесь для иллюстрации показано добавление только одного элемента списка. В других при мерах мы рассмотрим списки с большим числом элементов.

3.9. Операции для работы со списком

Основные общие операции для работы со списками:

- Создание нового списка
- Добавление нового элемента в список (в голову, в хвост и по номеру)
- Удаление элемента из списка (из головы, с хвоста и по номеру)
- Очистка всего списка
- Распечатка всего списка
- Поиск элемента в списке по номеру или по ключу.
- Замена местами элементов по номерам

Эти операции будут различаться для однонаправленных и двунаправленных списков, для динамических и статических списков, а также для списков, в которых учтена специфика хранимых объектов. В основной теоретической части мы будем рассматривать примеры работы с списками. Удаление элементов по номеру операция очень сложная для однонаправленных списков, поэтому в этой части не рассматривается.

3.10. Ручная работа с однонаправленным списком

Рассмотрим основные операции для работы с однонаправленным списком. Будем использовать следующую структуру элемента:

```
struct ListElem{
ListElem * pNext; // адрес следующего элемента списка
int ListVal; // информация, содержащаяся в списке
};
...
```

Фрагмент программы для ручного связывания статического однонаправленного списка:

```
// Ручное связывание и распечатка списка
ListElem E11;
ListElem E21;
ListElem E31;
E11.ListVal = 1;  // Значение информации 1-го элемента
E11.pNext = &E21;// Ссылается на 2-й элемент
E21.ListVal = 2; // Значение информации 2-го элемента
E21.pNext = &E31; // Ссылается на 3-й элемент
E31.ListVal = 3; // Значение информации 3-го элемента
E31.pNext = NULL; // не ссылается ни на кого
// Печать списка
```

```
        ListElem ETemp = {&E11 ,NULL }; // Временный элемент для навигации
        printf ("Содержимое ручного списка: \n");
        while (ETemp.pNext != NULL )
        {
        printf ("Элемент  = %d \n", ETemp.pNext ->ListVal );
        ETemp.pNext = ETemp.pNext ->pNext ; // Навигация
    };
    …
```

Результат работы программы:

```
Содержимое ручного списка:
Элемент  = 1
Элемент  = 2
Элемент  = 3
```

3.11. Структуры для добавления и удаления (список с указателями)

```
// Самый простой элемент списка
struct Elem{
Elem * pNext; // адрес следующего элемента списка
int ListVal; // информация, содержащаяся в списке
};
// Структура для однонаправленного списка
struct EList{  // Простейшая структура список Elem - элементарный список
Elem * pHead; // Голова списка
Elem * pTail; // Хвост списка
int Count; // Счетчик элементов
};
```

3.12. Описание списка и функция распечатки списка (список с указателями)

```
////////////////////////////////
//// Функции для печати списков
////////////////////////////////
void PrintList( EList * pList )
{
printf ("Печать списка в функции Число = %d \n" ,  pList->Count );
Elem * pETemp = pList->pHead; // Во временную переменную - указатель задаем адрес головы спика
while ( pETemp != NULL) // Проверка прохождения последнего элемента списка
{
printf ("Элемент простого списка в функции: %d \n" , pETemp->ListVal);
pETemp = pETemp->pNext; // НАВИГАЦИЯ: Важнейщий оператор для работы со списком
};
};
///Описание и нициализация простого списка (на основе элементного списка)
Elem LSE3 = { NULL , 3 }; // NULL – нулевой указатель – нет ссылки на другие элементы
Elem LSE2 = { &LSE3 , 2 }; // &LE3 - задание адреса третьего элемента в pNext
Elem LSE1 = { &LSE2 , 1 }; // &LE2 - задание адреса второго элемента в pNext
EList SecondList = { &LSE1 , &LSE3 , 3}; // 3 - Число элементов в списке голова, хвост, число элем
//
PrintList( &SecondList );
```

Результат работы программы:

```
Печать списка в функции Число = 3
Элемент простого списка в функции: 1
Элемент простого списка в функции: 2
Элемент простого списка в функции: 3
```

3.13. Добавление в голову списка (список с указателями)

```
// Новый элемент для добавления в голову
```

```
Elem ENew = {NULL , 5};
// Добавить в голову (в голову )!!!!!!!!!!!!!!
ENew.pNext = SecondList.pHead;
SecondList.pHead = &ENew;
SecondList. Count++;
PrintList( &SecondList );
```

Результат работы программы:

```
Печать списка в функции Число = 4
Элемент простого списка в функции: 5
Элемент простого списка в функции: 1
Элемент простого списка в функции: 2
Элемент простого списка в функции: 3
```

3.14. Добавление в хвост списка (список с указателями)

```
// Элемент для добавления в хвост
Elem ENew2 = {NULL , 10};
///
ENew2.pNext = NULL; // Для надежности
SecondList.pTail->pNext = &ENew2;
SecondList.pTail->pNext = &ENew2;
SecondList. Count++;
PrintList( &SecondList );
```

Результат работы программы:

```
Печать списка в функции Число = 5
Элемент простого списка в функции: 5
Элемент простого списка в функции: 1
Элемент простого списка в функции: 2
Элемент простого списка в функции: 3
Элемент простого списка в функции: 10
```

3.15. Удаление из головы списка (список с указателями)

```
// УДАЛЕНИЕ из головы
SecondList.pHead = SecondList.pHead->pNext;
SecondList. Count--;
PrintList( &SecondList );
```

Результат работы программы:

```
Печать списка в функции Число = 4
Элемент простого списка в функции: 1
Элемент простого списка в функции: 2
Элемент простого списка в функции: 3
Элемент простого списка в функции: 10
```

3.16. Удаление из хвоста списка (список с указателями)

```
// УДАЛЕНИЕ из хвоста (В этом списке сложнее всего)
// Поиск предпоследнего для нового хвоста
Elem * PrevTailElem = SecondList.pHead;
Elem * ETempCurr = SecondList.pHead;
while ( ETempCurr->pNext != NULL)
{
PrevTailElem = ETempCurr;
ETempCurr = ETempCurr->pNext;
};
// Само удаление !!!!!!!!!!!!!!!!!!
SecondList.pTail = PrevTailElem;
PrevTailElem->pNext = NULL;
```

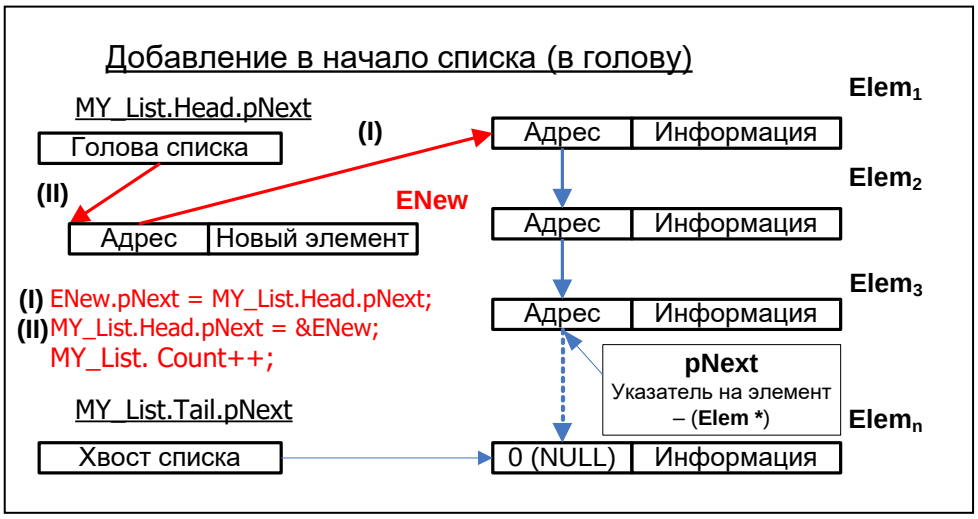
```
SecondList. Count--;
PrintList( &SecondList );
```

Результат работы программы:

```
Печать списка в функции Число = 3
Элемент простого списка в функции: 1
Элемент простого списка в функции: 2
Элемент простого списка в функции: 3
```

3.17. Добавление в голову списка (список без указателей)

Для добавления элемента в голову списка объявим структурную переменную список (**MY_List** типа **List**) и вручную запомним значения для головы и хвоста списка.




```
struct Elem{
Elem * pNext; // адрес следующего элемента списка
int ListVal; // информация, содержащаяся в списке
};
struct List {
Elem Head; // Голова списка без указателей
Elem Tail; // Хвост списка без указателей
int Count; // Счетчик элементов
};

// Описание и инициализация простого списка
List MY_List;
MY_List.Head.pNext = &E11; // Первый элемент списка
MY_List.Tail.pNext = &E31; // Последний элемент списка
// Новый элемент для добавления
ListElem ENew = {NULL , 5};
// Добавить элемент списка в голову
ENew.pNext = MY_List.Head.pNext;
MY_List.Head.pNext = &ENew;
MY_List. Count++;
```

Распечатка списка после добавления:

```
ETemp.pNext = MY_List.Head.pNext; // Временный элемент на начало списка
printf ("Содержимое ручного списка после добавления в голову: \n");
while (ETemp.pNext != NULL )
{
printf ("Элемент  = %d \n", ETemp.pNext ->ListVal );
```

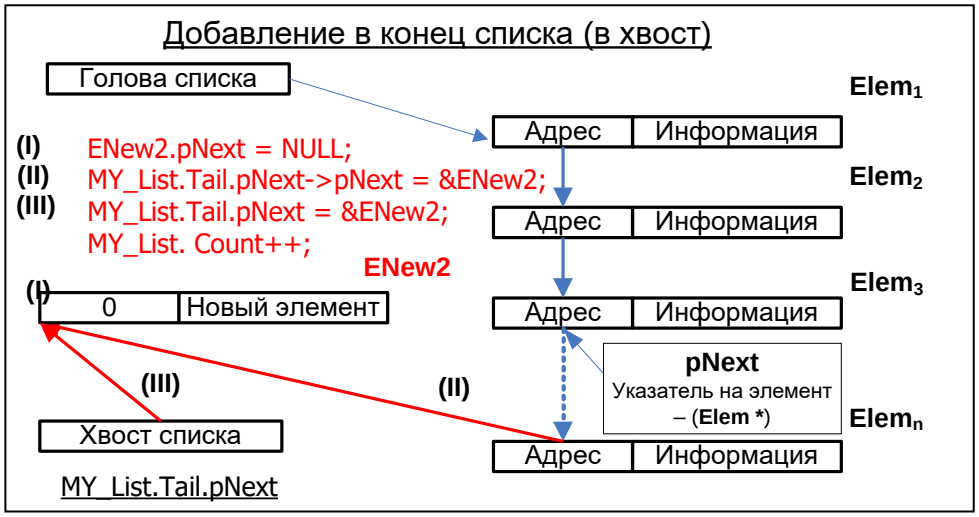
```
        ETemp.pNext = ETemp.pNext ->pNext ; // Навигация
    };
```

Результат распечатки:

```
Содержимое ручного списка после добавления в голову:
Элемент  = 5
Элемент  = 1
Элемент  = 2
Элемент  = 3
```

3.18. Добавление в конец списка (список без указателей)

При добавлении в конец (в хвост) списка для изменения адреса используется значение указателя, которое записано в поле Tail.pNext структуры список. Адрес нового элемента



(**&ENew2**) записывается поле адреса предыдущего последнего элемента списка и поле хвоста (**Tail**) структуры списка. Адресное поле нового элемента должно быть нулевым, в нашем случае мы использовали инициализацию эдемента.

```
// Добавление в хвост
ListElem ENew2 = {NULL , 10};
ENew2.pNext = NULL;
MY_List.Tail.pNext->pNext = &ENew2;
MY_List.Tail.pNext = &ENew2;
MY_List. Count++;
// Распечатка списка … (как в предудущем случае)
```

Результат распечатки:

```
Содержимое ручного списка после добавления в хвост:
Элемент  = 5
Элемент  = 1
Элемент  = 2
Элемент  = 3
Элемент  = 10
```

3.19. Функция распечатки однонаправленного элементного списка

Ниже приведен фрагмент программы, на котором расположен цикл распечатки списка. Список в данном случае задается передачей параметра первого элемента списка. Отметем, что

можно было бы передавать и указатель на первый элемент, такой алгоритм мы рассмотрим позднее.

```
// Распечатка списка, построенного на элементах ListElem
void PrintElemList (  ListElem H )
{
    printf ("Печать списка: \n");
  ListElem * pE = H.pNext ;
  while ( pE != 0)
  {
     printf ("Элемент  = %d \n", pE->ListVal );
     pE = pE->pNext; // Очень важно - навигация по списку
  };
};
```

Сначала печатается заголовок печати, а затем последовательно распечатываются все элементы списка, для перебора элементов списка используется оператор навигации, а для проверки конца цикла значение текущего адреса (указатель - **PE**). Вызов этой функции через структуру список (**List - MY_List**) выглядит так (полученный адрес первого элемента списка преобразуется в сам элемент с помощью операции разыменования):

…

```
PrintElemList (  *(MY_List.Head.pNext) );
```

…

Такая запись эквивалентно следующей записи для нашего примера:

```
PrintElemList (  E11 );
```

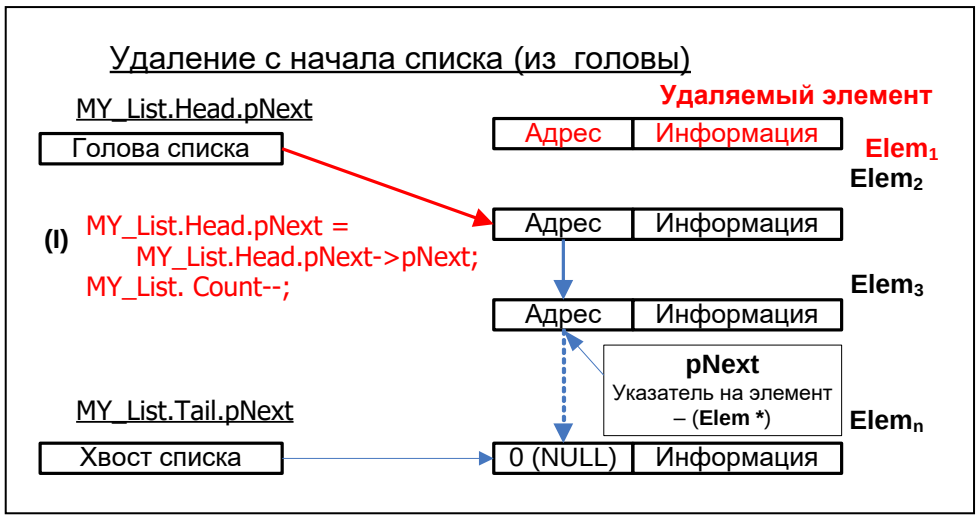
…

Результат вывода:

```
Печать списка:
Элемент  = 1
Элемент  = 2
Элемент  = 3
```

3.20. Удаление из головы списка (список без указателей)

Операция удаления из головы намного проще, чем операции добавления. Для удаления из головы (отметим, что список однонаправленный) достаточно изменить значение адреса, содержащегося в голове списка:



Текст программы удаления из головы выглядит так:

```
// Удаление из головы списка
printf ("Содержимое  списка до удаления из головы: \n");
```

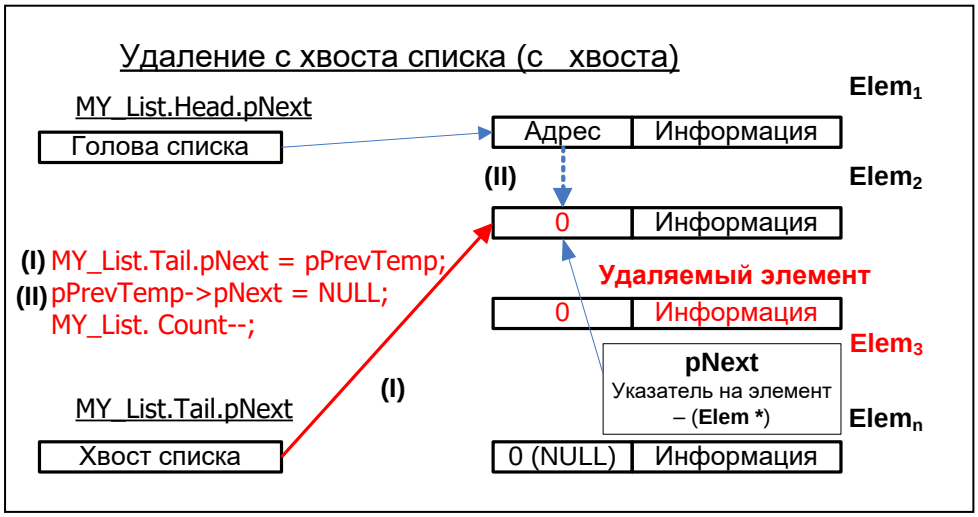
```
        PrintElemList (  *(MY_List.Head.pNext) );
        MY_List.Head.pNext = MY_List.Head.pNext->pNext;
        MY_List. Count--;
        printf ("Содержимое  списка после удаления из головы: \n");
        PrintElemList (  *(MY_List.Head.pNext) ); // Вызов функции печати списка
```

Результат вывода:

```
Содержимое  списка до удаления из головы:
Печать списка:
Элемент  = 5
Элемент  = 1
Элемент  = 2
Элемент  = 3
Элемент  = 10
Содержимое  списка после удаления из головы:
Печать списка:
Элемент  = 1
Элемент  = 2
Элемент  = 3
Элемент  = 10
```

3.21. Удаление из хвоста списка (список без указателей)

Удаление с хвоста списка для однонаправленного списка намного сложнее, так как адрес предпоследнего элемента с хвоста неизвестен. В этом случае первоначально определяется адрес



этого элемента: нужно “добежать” до хвоста, запомнив предварительно адрес предыдущего элемента (**pPrevTemp**), а затем выполнить операцию удаления (изменить адрес хвоста списка и обнулив адрес у найденного предпоследнего элемента).

```
// Удаление с хвоста списка
printf ("Содержимое  списка до удаления с хвоста: \n");
PrintElemList  (  *(MY_List.Head.pNext) );
// Поиск предпоследнего элемента списка!!!!!!(в pPrevT)
ListElem * pPrevT =  MY_List.Head.pNext;
// ETemp.pNext = MY_List.Head.pNext; // временный указатель для цикла
ListElem * pPrevTemp = MY_List.Head.pNext; // временный указатель для предыдущего элемента
if ( pPrevT != NULL ) // Элементы в списке есть
{
  while ( pPrevT != NULL)
  {
   pPrevTemp = pPrevT; // Запоминание предыдущего элемента
   pPrevT = pPrevT ->pNext ; // Навигация
```

```
        if (pPrevT->pNext == NULL ) break;
        }
      // Удаление последнего элемента
      MY_List.Tail.pNext = pPrevTemp;
      pPrevTemp->pNext = NULL;
      MY_List. Count--;
      // Печать измененного списка списка
      printf ("Содержимое  списка после удаления с хвоста: \n");
      PrintElemList  (  *(MY_List.Head.pNext) );
};

   ...
```

Результат вывода:

```
Содержимое  списка до удаления с хвоста:
Печать списка элементов:
Элемент  = 1
Элемент  = 2
Элемент  = 3
Элемент  = 10
Содержимое  списка после удаления с хвоста:
Печать списка элементов:
Элемент  = 1
Элемент  = 2
Элемент  = 3
```

Примечание 1: Рассмотрены основные операции добавления и удаления элементов в/из списков. К сожалению, они в таком виде применимы только для частных случаев, так как обычно текущее состояние списка заранее неизвестно. В окончательном виде при добавлении и удалении необходимо учитывать следующие факторы: текущее состояние списка (пуст ли он?), каким он будет после выполнения операции, как нужно установить основные поля структуры списка после завершения операции. В разделе примеров данных МУ представлены функции более универсального характера для выполнения этих операций.

3.22. Очистка статического списка (список без указателей)

Для очистки статического списка, все элементы которого являются также статическими (описаны в программе) достаточно установить указатели головы и хвоста в нуль, при построении списка с полями в виде элементов списка:

```
struct List {
ListElem Head; // Голова списка
ListElem Tail; // Хвост списка
int Count; // Счетчик элементов
};
...
MY_List.Head.pNext = NULL;
MY_List.Tail.pNext = NULL;
```

Если список организован по-другому, для головы и хвоста используются указатели на элементы списка, то очистка списка будет выглядеть иначе:

```
 struct PList {
ListElem * pHead; // Голова списка
ListElem * pTail; // Хвост списка
int Count; // Счетчик элементов
```

```
};
...
MY_List. pHead = NULL;
MY_List. pTail = NULL;
```

Примечание 2: В этом случае значительно поменяются также многие действия для выполнения операций над списками рассмотренные выше. Примеры таких списков мы рассмотрим ниже на основе двунаправленных списков (см. ниже в этой главе.).

3.23. Простая ручная работа с динамическим списком

Перед каждым фрагментом программы дано пояснение сути действий. Для работы используются: структура однонаправленного списка (**List**) и элемента списка(**ListElem**). Для работы используется библиотека (<**malloc.h**>). Красным цветом выделены операции навигации по списку и операция работы со списклм.

Создание динамического списка:

```
// Динамический список -  пока только ручная работа
////Создание динамического списка
List * pList = (List *) malloc ( sizeof(List));
// Начальная инициализация списка
pList->Count = 0;
pList->Head.pNext = NULL;
pList->Tail.pNext = NULL;
```

Создание элементов списка и связывание списка (три элемента):

```
////Создание и добавление элементов
ListElem *pDETemp;
// 1 = й
pDETemp = (ListElem *) malloc ( sizeof(ListElem));
pDETemp->pNext = NULL;
pDETemp->ListVal = 1 ;
pList->Count  = 1;
pList->Head.pNext = pDETemp; // Добавить первый
pList->Tail.pNext = pDETemp;
// 2 - й в голову
pDETemp = (ListElem *) malloc ( sizeof(ListElem));
pDETemp->ListVal = 2 ;
pDETemp->pNext = pList->Head.pNext;// Добавить второй в голову
pList->Head.pNext = pDETemp;
// 3 - й в хвост
pDETemp = (ListElem *) malloc ( sizeof(ListElem));
pDETemp->ListVal = 3 ;
pDETemp->pNext = NULL ;
pList->Tail.pNext->pNext =  pDETemp;
pList->Tail.pNext = pDETemp;// Добавить третий в хвост
////Печать списка
PrintElemList  (  *(pList->Head.pNext) );
```

Удаление ч головы списка:

```
////Удаление с головы
pDETemp = pList->Head.pNext ; // Запомним для очистки памяти
pList->Head.pNext = pList->Head.pNext->pNext;
free( pDETemp ); // Очистить память
```

Удаление с хвоста списка:

```
////Удаления с хвоста
pDETemp = pList->Tail.pNext ; // Запомним для очистки памяти
// Поиск предпоследнего для укорочения списка
ListElem * pFind = pList->Head.pNext ;
```

```
while( pFind != NULL)
{
if ( pFind ->pNext->pNext == NULL) break;
pFind = pFind->pNext;
};
pFind->pNext = NULL; // Укоротить список
free( pDETemp ); // Очистить память
//
printf ("Печать после удаления с головы и хвоста: \n");
PrintElemList  (  *(pList->Head.pNext) );
```

Цикл занесения динамических элементов в список:

```
//  Цикл динамического заполнения списка в голову (5 элементов)
for ( int i = 0 ;  i< 3  ; i++ )
{
pDETemp = (ListElem *) malloc ( sizeof(ListElem));
pDETemp->ListVal = i + 10;
pDETemp->pNext = pList->Head.pNext;// Добавить текущий элемент в голову
pList->Head.pNext = pDETemp;
};
```

Печать списка в цикле:

```
// Печать списка
pDETemp = pList->Head.pNext;
printf ("Печать списка после динамического добавления: \n");
while( pDETemp != NULL)
{
printf ("Элемент списка: - %d \n" , pDETemp->ListVal );
pDETemp = pDETemp->pNext; // Навигация
};
```

Очистка динамического списка (в цикле удаляем все элементы – **pTemp**, а затем структуру списка **pList**, созданного динамически выше)

```
////Очистка динамического списка (с головы?)
pDETemp = pList->Head.pNext;
k=1;
while( pDETemp != NULL)
{
ListElem * pTemp = pDETemp;
pDETemp = pDETemp->pNext; // Навигация
free (pTemp);
};
free (pList);
...
```

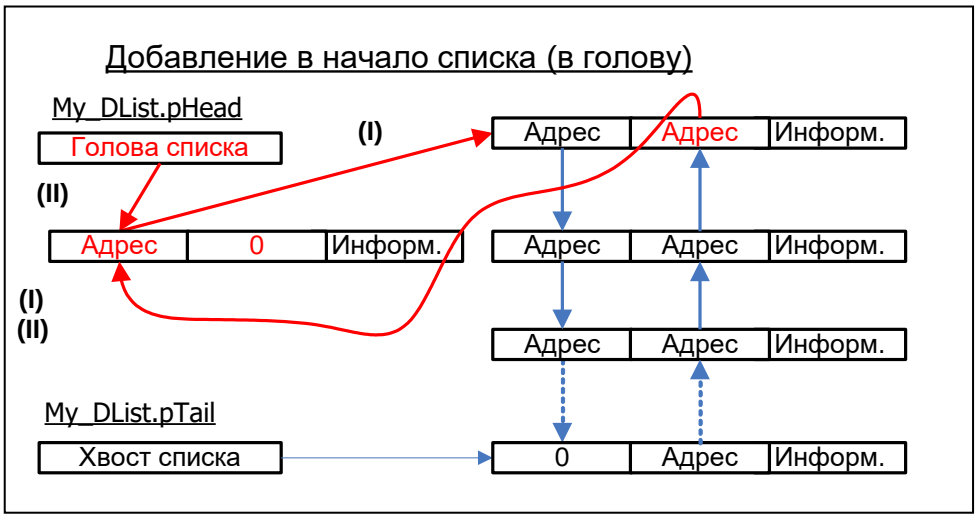
Работы программы для динамических списков:

```
Печать списка элементов:
Элемент  = 2
Элемент  = 1
Элемент  = 3
Печать после удаления с головы и хвоста:
Печать списка элементов:
Элемент  = 1
Печать списка после динамического добавления:
Элемент списка: - 12
Элемент списка: - 11
Элемент списка: - 10
Элемент списка: - 1
```

После удаления с головы и хвоста в нашем списке остается только один элемент. При добавлении в голову в цикле трех элементов, общее число элементов увеличивается до четырех.

3.24. Простая ручная работа с двунаправленным списком

На рисунке покажем, что должно происходить при добавлении элемента в голову для двунаправленного списка. Для других операций рисунки в этом разделе приводить не будем. Их можно построить по аналогии. В тексте операции добавления и удаления выделены красным цветом.



Описание двунаправленного списка и его элементов:

```
// Структура данных для демонстрации
struct Student {
 char Name[20];
 int Num;
 float Oklad;
};
// Структура элемента двунаправленного списка для демонстрации
struct DElem {
DElem * pNext;
DElem * pPrev;
Student * pSTud;
};
// Структура двунаправленного списка для демонстрации
struct DoubleList {
DElem * pHead; // Голова списка - указатель
DElem * pTail;   // Хвост списка - указатель
int Count;    // Число элементов
};
```

Инициализация и заполнение списка (**DoubleList**) статическими элементами списка (**DElem**):

```
// Список
DoubleList My_DList;
// Инициализация двунаправленного списка
My_DList.Count = 0;
My_DList.pHead = NULL;
My_DList.pTail = NULL;
// Первый элемент двунаправленного списка
DElem D1 = { NULL , NULL , NULL};
D1.pSTud = (Student *) malloc ( sizeof(Student));
D1.pSTud->Num = 1;
D1.pSTud->Oklad = 10.0f;
```

```
strcpy( D1.pSTud->Name,"Петров");
// Второй элемент двунаправленного списка
DElem D2 = { NULL , NULL , NULL};
D2.pSTud = (Student *) malloc ( sizeof(Student));
D2.pSTud->Num = 2;
D2.pSTud->Oklad = 20.0f;
strcpy( D2.pSTud->Name,"Сидоров");
// Третий элемент двунаправленного списка
DElem D3 = { NULL , NULL , NULL};
D3.pSTud = (Student *) malloc ( sizeof(Student));
D3.pSTud->Num = 3;
D3.pSTud->Oklad = 30.0f;
strcpy( D3.pSTud->Name,"Иванов");
// Добавление в список трех
My_DList.pHead = &D1;
My_DList.pTail = &D3;
// Прямая адресация
D1.pNext = &D2; // Текущий (первый) ссылается на следующий (второй) …
D2.pNext = &D3;
D3.pNext = NULL;
//Обратная адресация
D1.pPrev = NULL;
D2.pPrev = &D1;
D3.pPrev = &D2; // Последний ссылается на предпоследний …
My_DList.Count = 3;
```

Распечатка списка в цикле в обратном порядке для примера:

```
// Печать списка в обратном порядке для примера
printf ("Печать двунаправленного списка в обратном порядке: \n");
DElem * pWorkDE = My_DList.pTail ;
while ( pWorkDE != NULL)
{
printf (" Фам -> %s  Курс -> %d   Оклад -> %f \n", pWorkDE->pSTud->Name,
                pWorkDE->pSTud->Num, pWorkDE->pSTud->Oklad );
pWorkDE = pWorkDE ->pPrev; // Навигация в обратном порядке
};
```

Добавление в голову в двунаправленном списке:

```
// Новый элемент для добавления в голову
DElem DHEAD = { NULL , NULL , NULL};
DHEAD.pSTud = (Student *) malloc ( sizeof(Student));
DHEAD.pSTud->Num = 5;
DHEAD.pSTud->Oklad = 50.0f;
strcpy( DHEAD.pSTud->Name,"ГОЛОВА");
//  Добавление в голову (см. рисунок)
DHEAD.pNext = My_DList.pHead;
DHEAD.pPrev = NULL;
My_DList.pHead->pPrev = &DHEAD;
My_DList.pHead = &DHEAD;
My_DList.Count++;
```

Добавление в хвост в двунаправленном списке:

```
// Новый элемент для добавления в хвост
DElem DTAIL = { NULL , NULL , NULL};
DTAIL.pSTud = (Student *) malloc ( sizeof(Student));
DTAIL.pSTud->Num = 10;
DTAIL.pSTud->Oklad = 100.0f;
strcpy( DTAIL.pSTud->Name,"ХВОСТ");
//Добавление в  хвост
DTAIL.pNext = NULL;
DTAIL.pPrev =  My_DList.pTail;
```

```
My_DList.pTail->pNext = &DTAIL;
My_DList.pTail = &DTAIL;
My_DList.Count++;
```

Распечатка списка в цикле в прямом порядке:

```
// Распечатка
printf ("\nПечать двунаправленного списка после добавления в голову и хвост: \n");
pWorkDE = My_DList.pHead ;
while ( pWorkDE != NULL)
{
printf (" Фам -> %s  Курс -> %d   Оклад -> %f \n", pWorkDE->pSTud->Name,
                pWorkDE->pSTud->Num, pWorkDE->pSTud->Oklad );
pWorkDE = pWorkDE ->pNext; // Навигация
};
```

Удаление из головы в двунаправленном списке:

```
// Удаление из головы
My_DList.pHead =My_DList.pHead->pNext;
My_DList.pHead->pPrev = NULL;
My_DList.Count--;
```

Удаление из хвоста в двунаправленном списке:

```
// Удаление из хвоста
My_DList.pTail = My_DList.pTail->pPrev;
My_DList.pTail ->pNext = NULL;
My_DList.Count--;
```

Распечатка списка в цикле в прямом порядке:

```
// Распечатка
printf ("\nПечать двунаправленного списка после удаления из головы и хвоста: \n");
pWorkDE = My_DList.pHead ;
while ( pWorkDE != NULL)
{
printf (" Фам -> %s  Курс -> %d   Оклад -> %f \n", pWorkDE->pSTud->Name,
                pWorkDE->pSTud->Num, pWorkDE->pSTud->Oklad );
pWorkDE = pWorkDE ->pNext;
};
```

Очистка списка двунаправленного статического списка

```
// Очистка списка
My_DList.pHead= NULL;
My_DList.pTail= NULL;
My_DList.Count = 0;
```

Результаты работы фрагмента программы с двунаправленным списком:

```
Печать двунаправленного списка в обратном порядке:
 Фам -> Иванов  Курс -> 3   Оклад -> 30.000000
 Фам -> Сидоров  Курс -> 2   Оклад -> 20.000000
 Фам -> Петров  Курс -> 1   Оклад -> 10.000000

Печать двунаправленного списка после добавления в голову и хвост:
 Фам -> ГОЛОВА  Курс -> 5   Оклад -> 50.000000
 Фам -> Петров  Курс -> 1   Оклад -> 10.000000
 Фам -> Сидоров  Курс -> 2   Оклад -> 20.000000
 Фам -> Иванов  Курс -> 3   Оклад -> 30.000000
 Фам -> ХВОСТ  Курс -> 10   Оклад -> 100.000000

Печать двунаправленного списка после удаления из головы и хвоста:
 Фам -> Петров  Курс -> 1   Оклад -> 10.000000
 Фам -> Сидоров  Курс -> 2   Оклад -> 20.000000
 Фам -> Иванов  Курс -> 3   Оклад -> 30.000000
```

Знакомство с выполнением операций для двунаправленного списка позволяет сделать вывод, что операции добавления и удаления выполняются более просто, в частности отсутствуют циклы для поиска хвоста списка (предыдущего элемента от хвоста), которые необходимы для выполнения операций с однонаправленными списками, проще выполнить распечатку списка в обратном порядке.

4. Примеры программы с использованием файлового ввода и вывода

Вторая часть задания, помимо первой связанной с изучением теоретического раздела заключается в том, чтобы испытать в проекте СИ уже отлаженные программы и фрагменты программ. Возможно, что, осваивая теоретическую часть работы, вы уже на компьютере проверили выполнение фрагментов текста и применения различных операторов и алгоритмов (из раздела 3), тогда вам будет проще продемонстрировать их работу преподавателю. В дополнение к примерам, расположенным выше нужно испытать и изучить примеры расположенные ниже. Эти действия желательно сделать в отладчике. В приложении также представлены интересные примеры по теме лабораторной работы.

Для этого нужно создать пустой проект в MS VS (Test_LR2), как описано выше, скопировать через буфер обмена в него текст данных примеров, отладить его и выполнить.

4.1. Примеры, описанные в теоретической части ЛР

Нужно внимательно изучить и проверить работу всех примеров из теоретической части ЛР. Эти примеры расположены выше. Все примеры можно скопировать в свой проект. Все эти задания выполняются обязательно, они не требуют дополнительной отладки и легко (через буфер обмена -**Clipboard**) переносятся в программу. Все фрагменты должны демонстрироваться преподавателю. В частности, в первой, теоретической части представлены следующие примеры:

1. Создание простейшего списка вручную.
2. Распечатка простейшего списка в цикле (**FirstList**).
3. Простейший динамический список (**pDList**).
4. Ручная работа с однонаправленным списком на основе элементов.
5. Ручная работа с однонаправленным списком на основе структуры (**MY_List**).
6. Ручное добавление и удаление в голову и хвост.
7. Распечатка однонаправленного списка с помощью функции.
8. Очистка статического и динамического списков.
9. Работа с динамическим списком (указатель - **pList**).
10. Работа с двунаправленным списком (**My_DList**).

Кроме этого ниже представлены примеры, которые могут быть полезными, в том числе и при выполнении контрольных заданий. Их тоже желательно изучить и проверить в программном проекте. Выполнение этих задач не обязательно.

4.2. Структуры данных для примеров с однонаправленными списками

Структуры данных для элемента списка (**ListElem**) и самого списка (**List**) представлены ниже. Они используются для всех примеров с однонаправленными списками.

```
// Элементы списка
struct ListElem{
ListElem * pNext; // адрес следующего элемента списка
int ListVal; // информация, содержащаяся в списке
};
// Структура списка
struct List {
ListElem Head; // Голова списка
ListElem Tail; // Хвост списка
int Count; // Счетчик элементов
};
```

4.3. Функции для работы с однонаправленным списком

Ниже приведены универсальные функции для работы с однонаправленными списками. Эти функции используются в примерах представленных в следующих разделах данных методических указаний.

```
// Добавление элементов в однонаправленный список (в голову)
void AddList( List * pL , ListElem * pE)
{
 if (pL->Head.pNext == NULL) {pL->Head.pNext = pE; pL->Tail.pNext = pE;}
else
{
 pE->pNext = pL->Head.pNext;
 pL->Head.pNext = pE;
};
(pL->Count)++;
};
// Добавление элементов в однонаправленный список (в хвост)
void AddTailList( List * pL  , ListElem * pE)
{
 if (pL->Head.pNext == NULL) {pL->Head.pNext = pE; pL->Tail.pNext = pE;}
else
{

// Поиск хвоста списка
 ListElem * pELast;
 ListElem * pCurr;
 pCurr = &(pL->Head) ;
  // Цикл поиска хвоста
    while ( pCurr->pNext != NULL )
        {
   pELast = pCurr;
   pCurr = pCurr->pNext;
        };
//
pELast->pNext = pCurr;
pCurr->pNext = pE;
pL->Tail.pNext = pE;
pE->pNext = NULL;
 };
 (pL->Count)++ ;
};
// Удаление элементов из однонаправленного списка (из головы)
void DelList( List * pL )
{
```

```
 if (pL->Head.pNext == NULL) { return; }
 else
 {
 ListElem * pE;
    pE = pL->Head.pNext ;
 pL->Head.pNext = pL->Head.pNext->pNext;
 pE->pNext = NULL;
    if (pL->Head.pNext == NULL) { pL->Tail.pNext = NULL; };
 (pL->Count)--;
 }; };
 // Удаление элементов из однонаправленного списка (из хвоста)
 void DelLastList( List * pL  )
 {
 if (pL->Head.pNext == NULL) return ;
 // Цикл поиска хвоста списка
 ListElem * pE;
    ListElem * pTemp;
 ListElem * pELast;
 // Один элемент в списке
 if (pL->Head.pNext->pNext == NULL) { pL->Head.pNext = NULL  ;
 pL->Tail.pNext = NULL;
 pL->Count = NULL;return ; } ;
     pTemp = &(pL->Head) ;
        pE = pL->Head.pNext ;
 // Поиск хвоста
    while ( pE->pNext != NULL )
        {
   pELast = pE;
   pTemp = pTemp->pNext;
   pE = pE->pNext;
        };
 // Найден конец списка и главное предыдущийперед концом
 pELast->pNext = NULL;
 pL->Tail.pNext = pELast;
 pTemp->pNext = NULL;
 (pL->Count)-- ;
 //
};
 // Печать списка
 void PrintList( List L)
 {
 if (L.Head.pNext == NULL) {
           printf ("Список List пуст! \n");
           return;};
 printf ("Печать списка List(Счетчик = %d): \n", L.Count);
  ListElem * pE = L.Head.pNext ;
  while ( pE != 0)
  {
 printf ("Элемент  = %d \n", pE->ListVal );
 pE = pE->pNext; // Очень важно - навигация по списку
  };
};
```

4.4. Функции вставки и удаления по номеру в однонаправленном списке

Ниже приведены универсальные функции для работы с однонаправленными списками. Эти функции могут использованы для добавления и удаления элемента по номеру, что в общем не очень характерно для списковых структур данных.

```
        // Добавление по номеру (номером непосредственно) Нумерация с нуля
        void AddListNum( List * pL , ListElem * pE , int Num)
        {
        //
                if ( Num <= NULL) {AddList( pL , pE ); return;}// В голову
                if ( Num >= pL->Count ) {  AddTailList( pL  , pE); return;}// В хвост
        // Добавить в середину Num > 0 и Num  < Count
        // Найти указатель заданного номера
        ListElem * pTemp = pL->Head.pNext;
        int nCur ;
        for ( nCur= 1  ;  nCur < Num ; nCur++ )
        pTemp = pTemp->pNext;
        // Добавление
        pE->pNext = pTemp->pNext;
        pTemp->pNext = pE;
        //Выход
        return;
    }
    ...
```

Фрагмент текста программы для добавления в список:

```
        //ФУНКЦИИ ДОБАВЛЕНИЯ для однонаравленного списка
        ListElem E55 = {NULL , 55};
        ListElem E77 = {NULL , 77};
        ListElem E99 = {NULL , 99};
        AddList( &MY_List , &E55 ); // В голову
        AddTailList (  &MY_List , &E77 ); // В хвост
        AddListNum( &MY_List , &E99 ,3 ); // По номеру 3 с нуля (0 - 1 - 2 -3)
    PrintElemList  (  *(MY_List.Head.pNext) );
    ...
```

Результаты работы программы.

```
Печать списка элементов:
Элемент  = 55
Элемент  = 1
Элемент  = 2
Элемент  = 99
Элемент  = 3
Элемент  = 77
```

Функция для удаления элемента из списка:

```
        //Удаление из списка по номеру
        void DelListNum( List * pL ,  int Num)
        {
        // Проверка на правильность номера для удаления
                if (( Num <  NULL) || ( Num >  pL->Count)) return; // Нет удаления
        // Проверка на первый и последний
                if ( Num <= NULL) {DelList( pL ); return;}// В голову
                if ( Num >= pL->Count ) {  DelLastList( pL  ); return;}// В хвост
        // Удалить в середину Num > 0 и Num  < Count
        // Найти указатель заданного номера
        ListElem * pTemp = pL->Head.pNext;
        int nCur ;
        for ( nCur = 1  ;  nCur < Num ; nCur++ )
        pTemp = pTemp->pNext;
        // Удаление
        pTemp->pNext = pTemp->pNext->pNext;
        //Выход
        return;
    }
```

...

Фрагмент текста программы для удаления из списка:

```
//ФУНКЦИИ УДАЛЕНИЯ  для однонаравленного списка
printf ("Содержимое списка до удаления функциями: \n");
PrintElemList  (  *(MY_List.Head.pNext) );
DelList( &MY_List); // Из головы
DelLastList(  &MY_List ); // Из хвоста
DelListNum ( &MY_List , 2 ); // По номеру
printf ("Содержимое списка после удаления функциями: \n");
PrintElemList  (  *(MY_List.Head.pNext) );
...
```

Результаты работы программы при удалении.

```
Содержимое списка до удаления функциями:
Печать списка элементов:
Элемент  = 55
Элемент  = 1
Элемент  = 2
Элемент  = 99
Элемент  = 3
Элемент  = 77
Содержимое списка после удаления функциями:
Печать списка элементов:
Элемент  = 1
Элемент  = 2
Элемент  = 3
```

4.5. Замена двух элементов по номеру в однонаправленном списке

Функция для замены двух элементов списке (**SwapNum**). Данная функция может быть использована для различных алгоритмов сортировки списков. Для замены указываются два номера. Для удобства и прозрачности замен используется вспомогательная функция (**GetListNum**) позволяющая пол номеру получить адрес элемента списка.

```
// Получение элемента по номеру без удаления
int GetListNum( List * pL ,  ListElem ** ppE , int Num , ListElem ** ppPrev = NULL)
{
        *ppE = NULL;
         ListElem ETemp = {NULL , 33};
    ListElem * pETemp = &ETemp;
    ListElem ** ppTemp = &pETemp ;
        if ( ppPrev == NULL )  ppPrev = ppTemp;
// Проверка на правильность номера для удаления
        if (( Num <  NULL) || ( Num >  pL->Count)) return -1 ; // Нет удаления
// Проверка на первый и последний
        if ( Num <= NULL) { *ppE = pL->Head.pNext ;
        *ppPrev = &(pL->Head);
        return 0;}// ГОЛОВА
        if ( Num >= pL->Count ) {  *ppE = pL->Tail.pNext ;
        *ppPrev = &(pL->Tail);
        return 0;}// ХВРСТ
// Удалить в середину Num > 0 и Num  < Count
// Найти указатель заданного номера
ListElem * pTemp = pL->Head.pNext;
*ppPrev = & (pL->Head) ;
int nCur ;
for ( nCur = 0  ;  nCur < Num ; nCur++ )
```

```
        {
          *ppPrev = pTemp;
          pTemp = pTemp->pNext;
        }
        //
        *ppE = pTemp;
        //Выход
        return 0;
        }
```

Функция замены по номеру

```
        // Замена в списке по номеру
        void SwapNum ( List * pL ,  int Num1 , int Num2)
        {
        ListElem * pE1;
        ListElem * pE2;
        ListElem * pPrevE1;
        ListElem * pPrevE2;
        ListElem * pTemp;
        GetListNum( pL ,  &pE1 , Num1, &pPrevE1);
        GetListNum( pL ,  &pE2 , Num2 ,&pPrevE2);
        if ( pE1 != pE2) // Равны элементы
        {
        //
        pTemp = pPrevE1->pNext;
        pPrevE1->pNext = pPrevE2->pNext;
        pPrevE2->pNext = pTemp;
        //
        pTemp = pE1->pNext;
        pE1->pNext = pE2->pNext;
        pE2->pNext = pTemp;
        }
        return;
    };
```

Пример использования функции замены по номерам в списке.

```
        printf ("Содержимое списка после SWAP по номеру: \n");
        SwapNum (  &MY_List,  1 , 5);
    PrintElemList  (  *(MY_List.Head.pNext) );
```

Результат работы функции замены:

```
Печать списка элементов:
Элемент  = 55
Элемент  = 1
Элемент  = 2
Элемент  = 99
Элемент  = 3
Элемент  = 77
Содержимое списка после SWAP по номеру:
Печать списка элементов:
Элемент  = 55
Элемент  = 77
Элемент  = 2
Элемент  = 99
Элемент  = 3
Элемент  = 1
```

4.6. Замена двух элементов по адресу в однонаправленном списке

Функция для замены двух элементов списке (**SwapPtr**). Данная функция может быть использована для различных алгоритмов сортировки списков. Для замены указываются два адреса. Для удобства и прозрачности замен используется вспомогательная функция (**GetListPNT**) позволяющая по адресу получить номер элемента списка.

```
int GetListPNT( List * pL , ListElem * pE , int * Num)
{
        *Num = 0;
ListElem * pTemp = pL->Head.pNext;
int Flag = false;
while ( pTemp != NULL)
{
        if ( pTemp == pE ) { Flag = true; break;}
 pTemp = pTemp->pNext;
  (*Num)++;
};
return Flag;
};
```

Пример использования функции замены по адресам в списке.

```
/// Замена в списке на основе элументов  -  указателей на них
void SwapPtr ( List * pL ,   ListElem * pE1,  ListElem * pE2)
{
        int Num1;
        int Num2;
if ((GetListPNT(  pL , pE1 , &Num1) == true) && (GetListPNT(  pL , pE2 , &Num2) == true ))
    SwapNum ( pL , Num1 , Num2);
  return;
};
...
```

Вызов функции замены:

```
printf ("Содержимое списка после SWAP по адресу: \n");
SwapPtr (&MY_List, &E77 , &E99);
PrintElemList  (  *(MY_List.Head.pNext) );
...
```

Результат работы функции:

```
Печать списка элементов:
Элемент  = 55
Элемент  = 77
Элемент  = 2
Элемент  = 99
Элемент  = 3
Элемент  = 1
Содержимое списка после SWAP по адресу:
Печать списка элементов:
Элемент  = 55
Элемент  = 99
Элемент  = 2
Элемент  = 77
Элемент  = 3
Элемент  = 1
```

4.7. Описание структур и основных функций для двунаправленного списка

Структуры данных, необходимые для демонстрации работы с двунаправленными списками в следующих примерах.

```
// Простой элемент двунаправленного списка - структура
struct Node {
Node * pNext;
Node * pPrev;
int ListVal;
};
// Струкрура для двунаправленного списка
struct DList {
Node Head; // Голова списка
Node Tail;   // Хвост списка
int Count;    // Число элементов
};
...
```

Функции, необходимые для демонстрации работы с двунаправленными списками в следующих примерах.

```
// Инициализация элемента двунаправленнного списка
void InitNode ( Node * pNode , Node * pN, Node * pP, int Val)
{
if ( pN != NULL) pNode->pNext = pN;
else pNode->pNext = NULL;
if ( pP != NULL) pNode->pPrev = pP;
else pNode->pPrev = NULL;
pNode->ListVal  = Val;
};
// Инициализация двунаправленнного списка
void InitList( DList * pL  )
{
pL->Head.pNext = NULL;
pL->Tail.pNext = NULL;
//
pL->Head.pPrev = NULL;
pL->Tail.pPrev = NULL;
pL->Count =NULL ;
};
// Распечатка  двунаправленнного списка
void DListPrint (  DList L )
{
printf ("Содержимое списка DList: \n");
   Node * pE = L.Head.pNext;
   if ( pE == NULL) printf ("Список пуст! \n");
  while ( pE != NULL )
  {
 printf ("Элемент  = %d \n", pE->ListVal );
 pE = pE->pNext; // Очень важно - навигация по списку
  };
};
// Добавление в голову двунаправленнного списка
void AddDList( DList *  pL  , Node * pNode ){
// В голову
 if ( pL->Head.pNext == NULL)
    { pL->Head.pNext = pNode;
     pL->Tail.pNext = pNode;
        pNode->pNext = NULL;
        pNode->pPrev = NULL;
           ( pL->Count ) ++ ;
              return;  };
// Для новой
pNode->pNext = pL->Head.pNext;
```

```
    pNode->pPrev = NULL;
    // Для старой первой
    pL->Head.pNext->pPrev = pNode;
    // Для головы
    pL->Head.pNext = pNode;
    //
    ( pL->Count ) ++ ;
};
    // Добавление в хвост двунаправленнного списка
    void AddTailDList( DList *  pL  , Node * pNode ){
     // Проверка пустого списка
     if ( pL->Tail.pNext == NULL)
          {
     //до добавления список был пуст
            pL->Head.pNext = pNode;
            pL->Tail.pNext = pNode;
            pNode->pNext = NULL;
            pNode->pPrev = NULL;
               ( pL->Count ) ++ ;
               return;  };
     // Для новой в хвост
       pNode->pPrev = pL->Tail.pNext ;
     pNode->pNext = NULL ;
     // Для бывшей последней
       pL->Tail.pNext->pNext = pNode;
     // Для хвоста
     pL->Tail.pNext = pNode;
     // увеличим счетчик элементов
     ( pL->Count ) ++ ;
};
...
```

Если описания функция для работы со списком вынесены в другой исходный модуль, то нужны прототипы в главном модуле.

```
...
// Прототипы функций для  двунаправленнного списка
  void InitNode ( Node * pNode , Node * pN, Node * pP, int Val);
  void InitList( DList * pL  );
  void DListPrint(  DList L );
  void AddDList( DList *  pL  , Node * pNode );
  void AddTailDList( DList *  pL  , Node * pNode );
```

4.8. Создание, заполнение и распечатка двунаправленного списка

Рассмотрим пример программы, в который включено следующее: описание списка (**DList**) и его элемента(**Node**); инициализацию двунаправленного списка и его элементов (**InitList** , **InitNode**); ручное добавление элементов в список (**AddDList**) и распечатку списка (**DListPrint**).

```
...
    // Динамические двунаправленные списки
    DList L1;
    // Начальная настройка списка
InitList( &L1  );
    DListPrint (  L1 ); // печать пустого списка
    // Указатель на динамический элемент двунаправленного списка
    Node * pNode;
    printf ("Добавление в голову: \n");
    // Выделение памяти, заполнение первого элемента и добавление в список
    pNode = (Node *) malloc (sizeof(Node));
```

```
    InitNode ( pNode , NULL,NULL, 1 );
    AddDList( &L1 , pNode );
    DListPrint (  L1 ); // печать списка с одним элементом
...
```

После выполнения фрагмента программы, расположенного выше, получим следующий результат:

```
Содержимое списка DList:
Список пуст!
Добавление в голову:
Содержимое списка DList:
Элемент  = 1
```

Добавим еще несколько элементов в список и получим результат, расположенный ниже:

```
// Выделение памяти, заполнение второго элемента
  pNode = (Node *) malloc (sizeof(Node));
  InitNode ( pNode , NULL,NULL, 2 );
  AddDList( &L1 , pNode );
  pNode = (Node *) malloc (sizeof(Node));
  InitNode ( pNode , NULL,NULL, 3 );
  AddDList( &L1 , pNode );
  DListPrint (  L1 ); // добавлено всего  3 элемента
  printf ("Добавление в хвост и в голову: \n");
  pNode = (Node *) malloc (sizeof(Node));
  InitNode ( pNode , NULL,NULL, 55 );
  AddTailDList( &L1 , pNode );  // добавлен элемент в хфост
  DListPrint (  L1 ); // печать списка
  Результат:
Добавление в хвост и в голову:
Содержимое списка DList:
Элемент  = 3
Элемент  = 2
Элемент  = 1
Элемент  = 55
...
```

4.9. Сумма целочисленных переменных списка

Заполним список элементами, как в предыдущем примере, и подсчитаем сумму его целочисленных значений (**pTemp->ListVal**).

```
// Динамический список
DList L1;
// Инициализация списка и его заполнение
InitList( &L1  );
pNode = (Node *) malloc (sizeof(Node));
InitNode ( pNode , NULL,NULL, 1 );
AddDList( &L1 , pNode );
pNode = (Node *) malloc (sizeof(Node));
InitNode ( pNode , NULL,NULL, 2 );
AddDList( &L1 , pNode );
pNode = (Node *) malloc (sizeof(Node));
InitNode ( pNode , NULL,NULL, 3 );
AddDList( &L1 , pNode );
DListPrint (  L1 ); // добавлено всего  3 элемента
// Сумма данных в списке
int Sum;
Sum = 0;
Node * pTemp;
pTemp = L1.Head.pNext;
```

```
while ( pTemp != NULL)
{
Sum = Sum + pTemp->ListVal;
pTemp = pTemp->pNext; // Очень важный оператор -- навигация по списку!!!
};
printf ("Результат суммирования в списке: Sum = %d \n\n", Sum);
```

Навигация по списку выполняется через указатели на следующий элемент списка(**pTemp = pTemp->pNext**). Проверка завершения цикла выполняется сравнением этого временного указателя с нулевым значением (**pTemp != NULL**).

Результат печати списка и суммы его целочисленных элементов:

```
Содержимое списка DList:
Элемент  = 3
Элемент  = 2
Элемент  = 1
Результат суммирования в списке: Sum = 6
```

4.10. Сумма и печать переменных списка с вложенной структурой

Вычисление суммы и печать переменных списка с вложенными структурами.

```
// Структура для демонстрации
struct Student {
 char Name[20];
 int Num;
 float Oklad;
};
// Для однонаправленного списка с включенными данными
struct SNode {
   SNode * pNext;
   Student Stud;
};

SNode S3 ={ NULL, {"Сидоров" , 3 , 30.00f}}; // Инициализация структуры при описании
SNode S2 ={ &S3,  {"Петров" , 2 , 20.00f}}; // Инициализация структуры при описании
SNode S1 ={ &S2,  {"Иванов" , 1 , 10.00f}};// Инициализация структуры при описании
SNode *pSNode = &S1;
// Для суммы окладов
float fSum = 0.0f;
while(pSNode != NULL)
{
printf ("Элемент простого списка: %s  %d  %f \n" , pSNode->Stud.Name,
       pSNode->Stud.Num, pSNode->Stud.Oklad       );
fSum = fSum + pSNode->Stud.Oklad ;
pSNode = pSNode->pNext;  // Навигация
};
// Сумма
printf ("Результат суммирования (оклады): Sum= %f \n\n", fSum);
```

Результат печати списка и суммы по структурным переменным:

```
Элемент простого списка: Иванов  1  10.000000
Элемент простого списка: Петров  2  20.000000
Элемент простого списка: Сидоров  3  30.000000
Результат суммирования (оклады): Sum= 60.000000
```

4.11. Печать переменных списка с внешними данными (указатель на структуру)

```
struct DSNode {
   DSNode * pNext;
```

```
        Student * pStud; // Ссылка на данные, выделяемые динамически
};
…
    // Описание переменных списка (связь ручная)
    DSNode DS3 ={ NULL, NULL };  //  {"Сидоров" , 3 , 30.00f}
    DSNode DS2 ={ &DS3,  NULL }; // {"Петров" , 2 , 20.00f}
    DSNode DS1 ={ &DS2,  NULL }; // {"Иванов" , 1 , 10.00f}
    // Заполнения переменных списка
    Student Stud1 = {"Иванов2" , 1 , 10.00f};
    DS1.pStud = &Stud1;
    Student Stud2 =  {"Петров2" , 2 , 20.00f};
    DS2.pStud = &Stud2;
    Student Stud3 = {"Сидоров2" , 3 , 30.00f};
    DS3.pStud = &Stud3;
    …
```

Цикл печати списка:

```
    DSNode * pDSNode = &DS1;
    //
    while(pDSNode != NULL)
    {
    printf ("Элемент простого списка: %s  %d  %f \n" , pDSNode->pStud->Name,
            pDSNode->pStud->Num, pDSNode->pStud->Oklad     );
    pDSNode = pDSNode->pNext;
};
…
```

Результат печати списка:

```
Элемент простого списка: Иванов2  1  10.000000
Элемент простого списка: Петров2  2  20.000000
Элемент простого списка: Сидоров2  3  30.000000
```

4.12. Печать переменных списка с динамическими данными void

Структуры данных с универсальными указателями (типа **void**). При работе с указателями универсального вида нужно вручную следить за преобразованием типов данных и обеспечивать корректную компиляцию с явным указанием типов (С помощью каст-выражений). Такие списки привлекательны тем, что могут хранить разные типы структурных переменных, но при этом написание программ и отладка усложняются, а вероятность ошибок возрастает. Возможность такого использования указателей базируется на правиле: любой тип указателя может быть преобразован к указателю типа void. Примеры универсальных структур:

```
// Минимально с универсальными указателями
struct VNode {
void * pData;
void * pNext;
};
//
struct VSList {
void * pHead; // Голова списка
void * pTail; // Хвост списка
int Count; // Счетчик студентов
};
```

Заполнений списков с **void**-указателями и каст-выражением (**(Student *)**):

```
// Ручное заполнение списка
VNode V3 = { NULL , NULL};
VNode V2 = { NULL , &V3};
```

```
VNode V1 = { NULL , &V2};
// 1 - й
V1.pData = (void *) malloc ( sizeof (Student));
strcpy( (char *)((Student * )V1.pData)->Name , "Первый");
((Student * )V1.pData)->Num = 1;
((Student * )V1.pData)->Oklad = 1000.0f;
// 2 - й
V2.pData = (void *) malloc ( sizeof (Student));
strcpy( (char *)((Student * )V2.pData)->Name , "Второй");
((Student * )V2.pData)->Num = 2;
((Student * )V2.pData)->Oklad = 2000.0f;
// 3 - й
V3.pData = (void *) malloc ( sizeof (Student));
strcpy( (char *)((Student * )V3.pData)->Name , "Третий");
((Student * )V3.pData)->Num = 3;
((Student * )V3.pData)->Oklad = 3000.0f;
// Список в нашем случае статический
VSList  VL = { &V1, &V3, 3};
// Печать
void * pV;
pV = VL.pHead;
// Цикл печати списка вручную
while(pV != NULL)
{
printf ("Печать элемента списка: %s  %d  %f \n" ,
      ((Student * )(((VNode *)pV)->pData))->Name , ((Student * )(((VNode *)pV)->pData))-
>Num ,
            ((Student * )(((VNode *)pV)->pData))->Oklad );
pV = ((VNode *)pV)->pNext; // Навигация по списку
};
```

Печать результатов для универсальных списков:

```
Печать элемента списка: Первый  1  1000.000000
Печать элемента списка: Второй  2  2000.000000
Печать элемента списка: Третий  3  3000.000000
```

Нужно создать пустой проект в **MS VS**, как описано выше, скопировать через буфер обмена в него текст данного примера, отладить его и выполнить.

5. Контрольные задания ЛР №8.

5.1. Создать однонаправленный список вручную

Описать элемент однонаправленного списка. Описать структуру для однонаправленного списка (см. примеры). Создать однонаправленный список с вложенной структурой для своего варианта ДЗ/КЛР (см. таблицу ниже). Список создается вручную и частично инициализируется при описании элементов. Число элементов списка не менее 5-ти. Проверка правильного создания и связывания списка выполняется в отладчике (окно “Local”).

Шаблон элемента списка:

struct FElem{

Elem * **pNext**; // адрес следующего элемента списка

int **ListVal**; }; // информация, содержащаяся в списке

// Описание и инициализация элементов списка – трех:

FElem LE3 = { **NULL** , 3 }; // **NULL** – нулевой указатель – нет ссылки на другие

FElem LE2 = { &LE3 , 2 }; // &LE3 - задание адреса третьего элемента в pNext
FElem LE1 = { &LE2 , 1 }; // &LE2 - задание адреса второго элемента в pNext

Шаблон структуры списка:

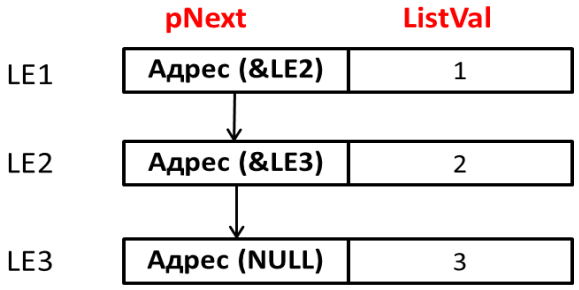
```
struct FastList{
	FElem * HEAD; // Содержит адрес первого элемента списка
	FElem * TAIL;  // Содержит адрес последнего элемента списка
	};
```

5.2. Распечатать однонаправленный список

Распечатать созданный в предыдущем пункте однонаправленный список. Для навигации по списку использовать указатели и оператор навигации по списку. Список напечатать в прямом порядке.

Фрагмент печати ручного списка:

```
//Список для примера
FastList  Flist1;
// Ручное задание параметров списка
Flist1.HEAD = &FLE1;
Flist1.TAIL = &FLE3;
printf ("Печать списка в цикле: \n" );
////////////////////////////////
//Контрольное задание № 5.2 Печать списка - фраг-
мент
////////////////////////////////
FElem * pFETemp = Flist1.HEAD; // Во временную пе-
ременную - указатель задаем адрес первого элемента
// Печать Простого списка
while ( pFETemp != NULL)
{
printf ("Элемент простого списка FElem: %d \n" , pFETemp->ListVal);
pFETemp = pFETemp->pNext; // НАВИГАЦИЯ: Важнейший оператор для работы со списком
	};
```



В результате получим:

```
Печать списка в цикле:
Элемент простого списка FElem: 1
Элемент простого списка FElem: 2
Элемент простого списка FElem: 3
```

5.3. Создать функцию для распечатки списка

Создать функцию для распечатки списка со своими структурами. В качестве параметра должен передаваться указатель на структуру списка. Продемонстрировать работу функции на своем списке с вызовом из основной программы.

Функция печати ручного списка:

```
/// Функция печати списка
void PrintFList(FastList * L)
{
printf ("ФУНКЦИЯ: Печать списка в цикле: \n" );
FElem * pETemp = L->HEAD ; // Во временную переменную - указатель задаем адрес первого эле-
мента
if(L->HEAD == NULL)
	printf ("ФУНКЦИЯ: Список пуст: \n" );
while ( pETemp != NULL)
{
printf ("Элемент простого списка Elem: %d \n" , pETemp->ListVal);
pETemp = pETemp->pNext; // НАВИГАЦИЯ: Важнейший оператор для работы со списком
```

```
 };
return; };
```

Вызов функции:

```
PrintFList (&LE1);
```

В результате получим:

```
ФУНКЦИЯ: Печать списка в цикле:
Элемент простого списка Elem: 1
Элемент простого списка Elem: 2
Элемент простого списка Elem: 3
```

5.4. Написать программу для добавления элементов в однонаправленный список (в голову списка)

Добавим в список в голову новый элемент (33) - FLE33:

```
////////////////////////////////
//Контрольное задание № 5.4а добавление в голову - Фрагмент
////////////////////////////////
FElem FLE33 = { NULL , 33 };
	FLE33.pNext = Flist1.HEAD ;
	Flist1.HEAD = &FLE33;
	PrintFList(&Flist1);
```

В результате получим:

```
ФРАГМЕНТ: Добавление 33:
ФУНКЦИЯ: Печать списка в цикле:
Элемент простого списка Elem: 33
Элемент простого списка Elem: 1
Элемент простого списка Elem: 2
Элемент простого списка Elem: 3
```

5.5. Создать функцию для добавления элементов в однонаправленный список (в голову списка)

Функция добавления в голову:

```
// Добавить в голову списка
// Добавить в голову списка
void AddHead(  FastList * L  , FElem * AddElem)
{
	AddElem->pNext = L->HEAD ;
	L->HEAD = AddElem;
	return; };
```

Добавим в голову на основе функции:

```
FElem FLE44 = { NULL , 44 }; // NULL – нулевой указатель (конец) – нет ссылки на другие
//
AddHead( &Flist1  , &FLE44);
// Проверка добавления
PrintFList(&Flist1);
```

В результате получим:

```
ФУНКЦИЯ: Печать списка в цикле:
Элемент простого списка Elem: 44
Элемент простого списка Elem: 33
Элемент простого списка Elem: 1
Элемент простого списка Elem: 2
Элемент простого списка Elem: 3
```

Данные добавляются в голову списка. Продемонстрировать работу функции на своем списке. Распечатать список после добавления с помощью своей функции.

5.6. Написать программу для добавления элементов в однонаправленный список (в хвост списка)

Добавим в список в хвост новый элемент (55) – LENEW55 :

```
//Контрольное задание № 5.5 Добавление в хвост фрагмент
////////////////////////////////
FElem FLE55 = { NULL , 55 };
// Добавление в хвост списка
Flist1.TAIL->pNext=&FLE55 ;
Flist1.TAIL = &FLE55;
// Проверка добавления
PrintFList(&Flist1);
```

В результате получим:

```
Фрагмент : Добавление в хвост 55:
ФУНКЦИЯ: Печать списка в цикле:
Элемент простого списка Elem: 33
Элемент простого списка Elem: 1
Элемент простого списка Elem: 2
Элемент простого списка Elem: 3
Элемент простого списка Elem: 55
```

5.7. Создать функцию для добавления элементов в однонаправленный список (голова и хвост)

Данные добавляются в хвост списка. Продемонстрировать работу функции на своем списке. Распечатать список после добавления с помощью своей функции.

Функция добавления в хвост:

```
// Добавление в хвост
void AddTail(  FastList * L  , FElem * AddElem)
{
L->TAIL->pNext=AddElem ;
L->TAIL = AddElem;
return; };
```

Прототип и вызов:

```
void AddTail(  FastList * L  , FElem * AddElem);
////
FElem FLE66 = { NULL , 66 };
AddTail( &Flist1  , &FLE66);
PrintFList(&Flist1);
```

В результате получим:

```
Функция  : Добавление в хвост 44:
ФУНКЦИЯ: Печать списка в цикле:
Элемент простого списка Elem: 77
…
Элемент простого списка Elem: 33
Элемент простого списка Elem: 66
```

5.8. Выполнить удаление элементов из списка (голова списка)

Написать фрагмент программы для удаления из однонаправленного списка с элементами своего задания. Вид удаления задается вариантом (голова или хвост). Продемонстрировать работу функции на своем списке. Распечатать список после добавления с помощью своей функции.

Для удаления из головы нужно изменить ссылку из головы списка:

```
Flist1.HEAD = Flist1.HEAD->pNext;
PrintFList(&Flist1);
```

В результате получим:

```
Элемент простого списка Elem: 77
…
Элемент простого списка Elem: 33
Элемент простого списка Elem: 66
```

Для удаления из хвоста нужно адресу предпоследнего элемента списка (pNext) задать NULL:

```
if ( Flist1.HEAD != NULL )
// Список не пуст
 {  FElem * pTemp = Flist1.HEAD ;
 FElem * pTempPrev = Flist1.HEAD;
 while (pTemp->pNext != NULL)
 { pTempPrev = pTemp;
   pTemp = pTemp->pNext ;}
 if (pTempPrev != NULL)
 {  pTempPrev->pNext = NULL; // Удаление
};  };
```

```
Элемент простого списка Elem: 77
…
Элемент простого списка Elem: 33
Элемент простого списка Elem: 1
Элемент простого списка Elem: 2
Элемент простого списка Elem: 3
Элемент простого списка Elem: 33
```

Или для списка с известными элементами: для удаления из хвоста нужно обнулить (NULL) ссылку из последнего элемента списка:

```
printf ("Функция  : Удаление из [хвоста 33 : \n" );
LENEW4.pNext= NULL;
PrintList(&LENEW2);
```

В результате получим:

```
Элемент простого списка Elem: 77
Элемент простого списка Elem: 1
Элемент простого списка Elem: 2
Элемент простого списка Elem: 3
Элемент простого списка Elem: 33
Элемент простого списка Elem: 44
Функция  : Удаление из хвоста 44 :
ФУНКЦИЯ: Печать списка в цикле:
Элемент простого списка Elem: 77
Элемент простого списка Elem: 1
Элемент простого списка Elem: 2
Элемент простого списка Elem: 3
Элемент простого списка Elem: 33
```

5.9. Создать функцию очистки списка

Создать специальную функцию для очистки списка. Продемонстрировать ее работу в основной программе.

```
//Контрольное задание № 5.7 Очистка списка
Flist1.HEAD = NULL;
Flist1.TAIL = NULL;
// Ручная очистка
Flist1.HEAD = &FLE1;  // Можно и так по - договоренности о нулевом списке
```

```
Flist1.TAIL = &FLE1;
```

Функция очистки списка:

```
void ClearList(FastList * L)
{
L->HEAD = NULL;
L->TAIL = NULL;
      };
// Очистка списка
ClearList(&Flist1);
PrintFList(&Flist1);
```

В результате получим:

```
ФУНКЦИЯ: Список пуст:
```

5.10. Написать программу для поиска экстремального элемента списка

Искать по варианту минимум или максимум для одного из целочисленных элементов своей структуры данных по варианту. Весь список, номер экстремального элемента и его значение этого элемента (распечатать предварительно весь список и найденный элемент - всю структуру).

```
//Поиск максимума
////////////////////////////////
// контрольные задания ЛР №8  5.10  !!! Поиск максимума в списке
////////////////////// Блок программы
{ FastList MinMaxList;  // Список для поиска максимума
FElem FLEM1 = { NULL , 100 };
FElem FLEM2 = { &FLEM1 , 400 };
FElem FLEM3 = { &FLEM2 , 30 };
FElem FLEM4 = { &FLEM3 , 300 };
MinMaxList.HEAD = &FLEM4;
MinMaxList.TAIL = &FLEM4;
PrintFList(&MinMaxList);
// Цикл поиска максимума
int ListMax=MinMaxList.HEAD->ListVal ; // Начальное значение Максимума
FElem * pTemp = MinMaxList.HEAD;
while (pTemp != NULL)
{ if( ListMax < pTemp->ListVal)
ListMax = pTemp->ListVal; // Текущий максимум
      pTemp = pTemp->pNext; // Навигация по списку
}; //
printf ("Значение Максимума = %d \n", ListMax); };
```

В результате получим:

```
      Значение Максимума = 400
```

5.11. Выполнить сложение двух списков

Создать вручную два однонаправленных или двунаправленных списка из трех элементов (тип списка задается по варианту), сложить эти списки (в общий список последовательно включаются элементы первого, а затем второго списка), распечатать отдельно оба списка и полученный результат после сложения списков. Ручное создание списков представлено в теоретической части данных МУ.

Сложение списков

```
////////////////////////////////
FastList  Flist101;
FElem FLE63 = { NULL , 103 }; // NULL – нулевой указатель (конец) – нет ссылки на другие
FElem FLE62 = { &FLE63 , 102 }; // &LE3 - задание адреса третьего элемента в pNext
FElem FLE61 = { &FLE62 , 101 }; // &LE2 - задание адреса второго элемента в pNext
Flist101.HEAD = &FLE61;
Flist101.TAIL = &FLE63;
printf ("Первый список \n");
```

```
PrintFList(&Flist101);
// контрольные задания ЛР №8  5.9  !!!
FastList  Flist201;
FElem FLE43 = { NULL , 203 }; // NULL – нулевой указатель (конец) – нет ссылки на другие
FElem FLE42 = { &FLE43 , 202 }; // &LE3 - задание адреса третьего элемента в pNext
FElem FLE41 = { &FLE42 , 201 }; // &LE2 - задание адреса второго элемента в pNext
Flist201.HEAD = &FLE41;
Flist201.TAIL = &FLE43;
printf ("Второй список \n");
PrintFList(&Flist201);
//////////////////////
// Сложение списков Flist101 + Flist201
//////////////////////
FastList  FlistREZ;
// голова и хвост нового списка
FlistREZ.HEAD = Flist101.HEAD;
FlistREZ.TAIL = Flist201.TAIL;
// связывание
Flist101.TAIL->pNext = Flist201.HEAD; /// Последний элемент Flist101 ссылается на первый
элемент Flist201
printf ("Результирующий список \n");
      PrintFList(&FlistREZ);
```

В результате получим:

```
Первый список
ФУНКЦИЯ: Печать списка в цикле:
Элемент простого списка Elem: 101
Элемент простого списка Elem: 102
Элемент простого списка Elem: 103
Второй список
ФУНКЦИЯ: Печать списка в цикле:
Элемент простого списка Elem: 201
Элемент простого списка Elem: 202
Элемент простого списка Elem: 203
Результирующий список
ФУНКЦИЯ: Печать списка в цикле:
Элемент простого списка Elem: 101
Элемент простого списка Elem: 102
Элемент простого списка Elem: 103
Элемент простого списка Elem: 201
Элемент простого списка Elem: 202
Элемент простого списка Elem: 203
```

6. Варианты заданий для студентов СУЦ.

Варианты заданий приведены ниже. Номер варианта должен соответсвовать номеру студента в групповом журнале.

п/п	Структура ДЗ 5 полей, 3 первых из них числовые	Поиск экстренума	Удаление элементов из списка	Для сложения списков	Сортировка (д.т.) по числовому параметру
1.	Кафедра	Минимум	Из хвоста	Однонаправленные списки	По убыванию

п/п	Структура ДЗ 5 полей, 3 первых из них числовые	Поиск экстренума	Удаление элементов из списка	Для сложения списков	Сортировка (д.т.) по числовому параметру
2.	Книга	Максимум	Из головы	Двунаправленные списки	По возрастанию
3.	Файл	Минимум	Из хвоста	Однонаправленные списки	По убыванию
4.	Автомобиль	Максимум	Из головы	Двунаправленные списки	По возрастанию
5.	Компьютер	Минимум	Из хвоста	Однонаправленные списки	По убыванию
6.	Группа	Максимум	Из головы	Двунаправленные списки	По возрастанию
7.	Человек	Минимум	Из хвоста	Однонаправленные списки	По убыванию
8.	Стеллаж	Максимум	Из головы	Двунаправленные списки	По возрастанию
9.	Дом	Минимум	Из хвоста	Однонаправленные списки	По убыванию
10.	Студент	Максимум	Из головы	Двунаправленные списки	По возрастанию

7. Дополнительные требования для студентов СУЦ (д.т.).

Для продвинутых студентов, по желанию, можно построить программу с дополнительными требованиями. Дополнительные требования выполняются в дополнение основным требованиям ЛР.

7.1. Создать функцию для добавления по номеру в двунаправленном списке

Внешние динамические данные, свои данные. Продемонстрировать с распечаткой.

7.2. Создать функцию для обмена элементов в двунаправленном списке

Внешние динамические данные, свои данные. Продемонстрировать с распечаткой.

7.3. Отсортировать список по числовому параметру из структуры

Тип сортировки задан вариантом. Внешние динамические данные, свои данные. Продемонстрировать с распечаткой.

7.4. Создать функцию для сортировки элементов по номеру

Тип сортировки задан вариантом. Внешние динамические данные, свои данные. Продемонстрировать с распечаткой.

7.5. Создать двунаправленный список вручную – вложенные в элемент данные

Данные вложены в элемент списка. Минимально записать 5-ть элементов в цикле, можно задать случайные данные. Распечатать список.

7.6. Записать данные из двунаправленного списка в файл

Написать фрагмент программы для записи списка, в двоичный файл. Список использовать из предыдущего примера. Свои данные записаны в структуре элемента списка. Результат записи в файл проверить в файл менеджере.

7.7. Прочитать данные из файла в двунаправленный список

Прочитать данные из двоичного файла в список. Файл записан в предыдущем задании. Свои данные записаны в структуре элемента списка. Полученный список распечатать и сравнить результат с файлом (посмотреть значения в шестнадцатеричном виде в файл менеджере).

8. Демонстрация, защита ЛР и отчет по ЛР.

После выполнения всех необходимых шагов по ЛР, работающую программу нужно продемонстрировать преподавателю, проводящему ЛР, о чем он в журнале делает отметку. Далее студент на основе шаблона и примера оформляет отчет по ЛР. После оформления отчета, который может быть представлен преподавателю в электронном виде, выполняется защита ЛР. Студент дает ответы на вопросы по отчету и на контрольные вопросы приведенные ниже. ЛР считается полностью зачтенной, если выполнены все перечисленные требования и действия: демонстрация, отчет и защита ЛР.

9. Контрольные вопросы по ЛР.

1. Что такое список как структура данных? Дайте определение.
2. Какие разновидности списков вы знаете?
3. Что такое элемент списка? Какие структуры элемента списка вы знаете?
4. Какие особенности списков вы знаете по сравнению с массивами?
5. Что такое статические и динамические списки?
6. Что такое однонаправленные и двунаправленные списки? Их преимущества?
7. Как строится и используется структура типа список? Что такое хвост и голова списка
8. Как выполняется движение (навигация) по списку?
9. Как может записываться информация в элементах списка?
10. В каких списках лучше выполнять операции добавления и удаления? Почему?
11. Как создаются динамические списки?
12. Какие основные операции над списками вы знаете?
13. Поясните работу с однонаправленным списком по МУ?
14. Поясните работу с двунаправленным списком по МУ?
15. Поясните работу со статическим списком по МУ?
16. Поясните работу с динамическим списком по МУ?
17. Поясните операцию добавления в голову списка?
18. Поясните операцию добавления в хвост списка?
19. Поясните операцию удаления из головы списка?
20. Поясните операцию удаления из хвоста списка?
21. Поясните алгоритм распечатки однонаправленного списка?
22. Как выполняется очистка статического списка?
23. Как выполняется очистка динамического списка?
24. Как записать список в файл, прочитать из файла?

10. Литература.

Основная литература

1. Список литературы, доступные книги и необходимые пособия для ЛР ОП размещены на сайте www.sergebolshakov.ru на страничке “2-й к СУЦ”. Пароль для доступа можно взять у преподавателя или старосты группы.
2. Керниган Б., Ритчи Д. К Язык программирования С, 2-е издание: Пер. с англ. – М. : Издательский дом “Вильямс”, 2009. – 304с.: ил. – Пар. Тит. англ.
3. Касюк, С.Т. Курс программирования на языке Си: конспект лекций/С.Т. Касюк. — Челябинск: Издательский центр ЮУрГУ, 2010. — 175 с.
4. MSDN Library for Visual Studio 2005 (Vicrosoft Document Explorer – входить в состав дистрибутива VS. Нужно обязательно развернуть при установке VS VS или настроить доступ через Интернет.)
5. Фридланд А.Я. Информатика и компьютерные технологии: Основные термины: Толк.слов.: Более 1000 базовых понятий и терминов. – 3-е изд., испр. и доп./ А.Я Фридланд, Л.С. Ханамирова, И.А. Фридланд – М.:ООО «Издательство Астрель»: ООО «Издательство АСТ», 2003. - 272 с.

Дополнительная литература

6. Общее методическое пособие по курсу для выполнения ЛР и ДЗ (см. на сайте 1-й курс www.sergebolshakov.ru) – см. кнопку в конце каждого раздела сайта!!!
7. Керниган Б., Ритчи Д. К36 Язык программирования Си.\Пер. с англ., 3-е изд., испр. - СПб.: "Невский Диалект", 2001. - 352 с.: ил.
8. Другие методические материалы по дисциплине с сайта www.sergebolshakov.ru.
9. Конспекты лекций по дисциплине “Основы программирования”.
10. Подбельский В.В. Язык Си++: Учебное пособие. – М.: Финансы и статистика, 2003.
11. 5. Подбельский В.В. Стандартный СИ++: Учебное пособие. – М.: Финансы и статистика, 2008.
12. Г. Шилдт “C++ Базовый курс”: Пер. с англ.- М., Издательский дом “Вильямс”, 2011 г. – 672с
13. Фридланд А.Я. Информатика и компьютерные технологии. Основные термины: толковый слов. : 3-е изд. Испр. и доп./ А.Я. Фридланд, Л.С. Хаааамирова, И.А. Фридланд. – М.:ООО «Издательство Астрель»: ООО «Издательство АСТ». 2003 – 272с.
14. Г. Шилдт “C++ Руководство для начинающих” : Пер. с англ. - М., Издательский дом “Вильямс”, 2005 г. – 672с
15. Г. Шилдт “Полный справочник по C++”: Пер. с англ.- М., Издательский дом “Вильямс”, 2006 г. – 800с
16. Бьерн Страуструп "Язык программирования C++"- М., Бином, 2010 г.

10.1. Приложение: фрагменты программ для работы со списками

Если хотите, сделайте это самостоятельно.

В данном разделе приведем два варианта записи массива структур в файл: низкоуровневый и на основе потоков. Используем структуру **Student**:

10.2. Структуры данных для примеров

```
// Простой элемент списка двунаправленного списка - структура
struct Node {
Node * pNext;
Node * pPrev;
int ListVal; // только целая переменная
};
// Струкрура для двунаправленного списка
struct DList {
Node Head; // Голова списка
Node Tail;   // Хвост списка
int Count;    // Число элементов
};
```

10.3. Функция инициализации элемента и списка

Функции начальной настройка списка и его элементов:

```
// Инициализация структуры двунаправленнного списка
void InitList( DList * pL  )
{
pL->Head.pNext = NULL;
pL->Tail.pNext = NULL;
//
pL->Head.pPrev = NULL;
pL->Tail.pPrev = NULL;
pL->Count =NULL ;
};
// Инициализация элемента списка двунаправленнного списка
void InitNode ( Node * pNode , Node * pN, Node * pP, int Val)
{
if ( pN != NULL) pNode->pNext = pN;
else pNode->pNext = NULL;
if ( pP != NULL) pNode->pPrev = pP;
else pNode->pPrev = NULL;
pNode->ListVal  = Val;
};
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

10.4. Распечатка двунаправленного списка

Функция распечатки списка:

```
// Распечатка двунаправленнного списка
void DListPrint (  DList L )
{
printf ("Содержимое списка DList: \n");
  Node * pE = L.Head.pNext;
```

```
        if ( pE == NULL) printf ("Список пуст! \n");
        while ( pE != NULL )
        {
                printf ("Элемент  = %d \n", pE->ListVal );
                pE = pE->pNext; // Очень важно - навигация по списку
    };
};
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

10.5. Добавление в голову двунаправленнного списка список

Функция добавления в голову:

```
// Добавление в голову двунаправленнного списка
void AddDList( DList *  pL  , Node * pNode ){
// В голову
        if ( pL->Head.pNext == NULL)
           { pL->Head.pNext = pNode;
            pL->Tail.pNext = pNode;
      pNode->pNext = NULL;
      pNode->pPrev = NULL;
                ( pL->Count ) ++ ;
           return;  };
// Для новой
pNode->pNext = pL->Head.pNext;
pNode->pPrev = NULL;
// Для старой первой
pL->Head.pNext->pPrev = pNode;
// Для головы
pL->Head.pNext = pNode;
//
( pL->Count ) ++ ;
};
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

10.6. Добавление в хвост двунаправленного список

Функция добавления в хвост:

```
// Добавление в хвост двунаправленнного списка
void AddTailDList( DList *  pL  , Node * pNode ){
        // Проверка пустого списка
        if ( pL->Tail.pNext == NULL)
     {
                // список пуст
                pL->Head.pNext = pNode;
           pL->Tail.pNext = pNode;
      pNode->pNext = NULL;
      pNode->pPrev = NULL;
                ( pL->Count ) ++ ;
```

```
            return;  };
          // Для новой в хвост
          pNode->pPrev = pL->Tail.pNext ;
              pNode->pNext = NULL ;
              // Для бывшей последней
          pL->Tail.pNext->pNext = pNode;
              // Для хвоста
              pL->Tail.pNext = pNode;
              // счетчик жлементов
              ( pL->Count ) ++ ;
    };
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

10.7. Добавить после заданного номера в двунаправленный список

Функция добавления по номеру:

```
        // Добавить после  заданного номера
        void AddNumDList( DList *  pL  , Node * pNode , int Num ){
          Node * pE = pL->Head.pNext; //  для навигации
          Node * pTemp = NULL; // Для запоминания
          // Пустой список
          if ( pE == NULL || Num < 1) { // printf ("Список пуст (AddNumDList)! \n");
          AddDList(  pL  ,  pNode ); //  в голову
          return;
          };
          // Проверка добавления в хвост
          if ( pL->Count <= Num  ) {
          AddTailDList(  pL  ,  pNode );
          return;
          };
          // Добавление в середину
        for (int i = 0 ; i < Num ; i++ )
        {
        pTemp = pE;
        pE = pE->pNext ;
        };
        // Добавление
        pNode->pNext = pE;
        pNode->pPrev = pTemp;
        pTemp->pNext  = pNode;
        pE->pPrev = pNode;
        ( pL->Count ) ++ ;
        return;
};
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

10.8. Удаление из головы двунаправленного списка

Функция удаления из головы:

```
        // Удаление из головы
        int DelDList( DList * pL  )
        {
```

```
                if ( pL->Head.pNext == NULL)  return NULL ;
                else
                { pL->Head.pNext =pL->Head.pNext ->pNext; // Прямая навигация
                if ( pL->Head.pNext == NULL)  pL->Tail.pNext = NULL;
                ( pL->Count ) -- ;
                return 1;};
    };
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

10.9. Очистка списка двунаправленного списка

Функция очистки списка:

```
        // Очистка
        void ClearDList( DList * pL  )
        {
         while ( DelDList ( pL ) != NULL );
};
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

10.10. Удаление из хвоста двунаправленного списка

Функция удаления из хвоста списка:

```
        // Удаление из хвоста DelTailDList
        int DelTailDList( DList * pL  )
        {
                if ( pL->Head.pNext == NULL)  return NULL ; // Список уже пуст
                else
                {
                pL->Tail.pNext =pL->Tail.pNext ->pPrev; // Обратная навигация
           pL->Tail.pNext ->pNext =  NULL;
                if ( pL->Tail.pNext== NULL)  pL->Head.pNext = NULL;
                ( pL->Count ) -- ;
                return 1;};
        };
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

10.11. Удаление по значению номера в ддвунаправленном списке

Функция удаления по номеру:

```
        // Удаление по значению номера в списке
        void DelNumDList( DList * pL ,  int  Num )
        {
          Node * pE = pL->Head.pNext; //  для навигации
          Node * pTemp = NULL; // Для запоминания
        // Удаление из головы
```

```
            if ( pE == NULL || Num < NULL ) { // printf ("Список пуст (DelNumDList)! \n");
            return;
              };
          // Голова
            if ( Num == 0  ) {
            DelDList(  pL  );
             return;
            };
          // Хвост
            if ( pL->Count < Num ||  pL->Count == Num  ) {
             return;
            };
            if (   pL->Count == Num + 1) {
            DelTailDList(  pL  );
             return;
       };
          // Поиск номера
          for (int i = 0 ; i < Num ; i++ )
          {
          pTemp = pE;
          pE = pE->pNext ;
          };
          // Удаление из списка найдено
          // удаление из середины
            pTemp->pNext = pE->pNext;
            pE->pNext->pPrev = pTemp;
            ( pL->Count ) -- ;
          return;
};
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

10.12. Получить из списка элемент по номеру

Функция получения из списка по номеру в списке:

```
          /// Получить из списка элемент по номеру без удаления
          int GetNumDList( DList * pL , Node ** pNode , int  Num   )
          {
          if ( Num < NULL || Num > pL->Count ) return 0; // Нет номера
          Node * pTemp = pL->Head.pNext;
          for ( int i = 0 ; i < Num; i++)
          {
          pTemp=pTemp->pNext;
          };
          *pNode = pTemp;
          return 1;
};
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

10.13. Обмен в списке 2-х элементов по номеру

Функция обмена 2-х элементов в спике:

```
/// обмен в списке 2-х элементов по номеру
void SwapDlist( DList * pL , int  NumA , int  NumB )
{
Node * pNodeA;//= new Node;
Node * pNodeB; //= new Node;
if (NumA == NumB  ) return;
if (GetNumDList( pL , &pNodeA , NumA   ) == 1 )
  if (GetNumDList( pL , &pNodeB , NumB   ) == 1 )
  {
        // Замена: сначала удалить по номеру, а потом добавить по номеру
          if (NumA > NumB  ) {
   DelNumDList( pL , NumB  );
   DelNumDList( pL , NumA - 1  );
        InitNode ( pNodeA , NULL,NULL, -1 );
   InitNode ( pNodeB , NULL,NULL, -1 );
   AddNumDList( pL  , pNodeA , NumB );
        AddNumDList( pL  , pNodeB , NumA ); }
          else {
   DelNumDList( pL , NumA  );
   DelNumDList( pL , NumB - 1 );
//    DListPrint (  *pL );
   AddNumDList( pL  , pNodeB , NumA );
//        DListPrint (  *pL );
  AddNumDList( pL  , pNodeA , NumB );
//   DListPrint (  *pL );
            }
  };
};
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

10.14. Сортировка двунаправленного списка

Фрагмент программы для сортировки списка:

…

```
printf ("До Сортировк: \n");
pNode = new Node;
InitNode ( pNode , NULL,NULL, 888 );
AddTailDList( &L1 , pNode );
DListPrint (  L1 );
//
printf ("После Сортировки: \n");
for ( int i = 0 ; i < L1.Count - 1 ; i++ )
  for ( int k = 0 ; k < L1.Count - 1 ; k++  )
  {
          Node * pNode1;
          Node * pNode2;
   GetNumDList( &L1 , &pNode1 , k   );
   GetNumDList( &L1 , &pNode2 , k +1  );
                if  (  pNode1->ListVal  <  pNode2->ListVal  )   // ">" - возрастание, "<" -
убывание
                SwapDlist( &L1 , k ,k+1 );
  };
DListPrint (  L1 );
```

```
};
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

10.15. Сумма элементов двунаправленного списка

Фрагмент программы для суммирования значения элементов списка:

```
        // Сумма данных в списке
    …
        int Sum;
        Sum = 0;
        Node * pTemp;
        pTemp = L1.Head.pNext;
        while ( pTemp != NULL)
        {
        Sum = Sum + pTemp->ListVal;
        pTemp = pTemp->pNext; // Очень важный оператор -- навигация по списку!!!
        };
    printf ("Результат суммирования в списке: Sum= %d \n\n", Sum);
};
```

Фрагмент программы для использования функции:

…

Результат работы программы:

…

(далее пока нет программ, можете примеры сделать сами)

10.16. Генерация случайных данных и заполнение двунаправленного списка

Функция …

Фрагмент программы для использования функции:

Результат работы программы:

10.17. Распечатка двунаправленного списка с хвоста

Функция …

Фрагмент программы для использования функции:

Результат работы программы:

10.18. Очистка динамического двунаправленного списка

Функция …

Фрагмент программы для использования функции:

Результат работы программы:

10.19. Минимум переменных списка, номер и адрес

Функция …

Фрагмент программы для использования функции:

Результат работы программы:

10.20. Поиск переменных списка по ключу

Функция …

Фрагмент программы для использования функции:

Результат работы программы: