

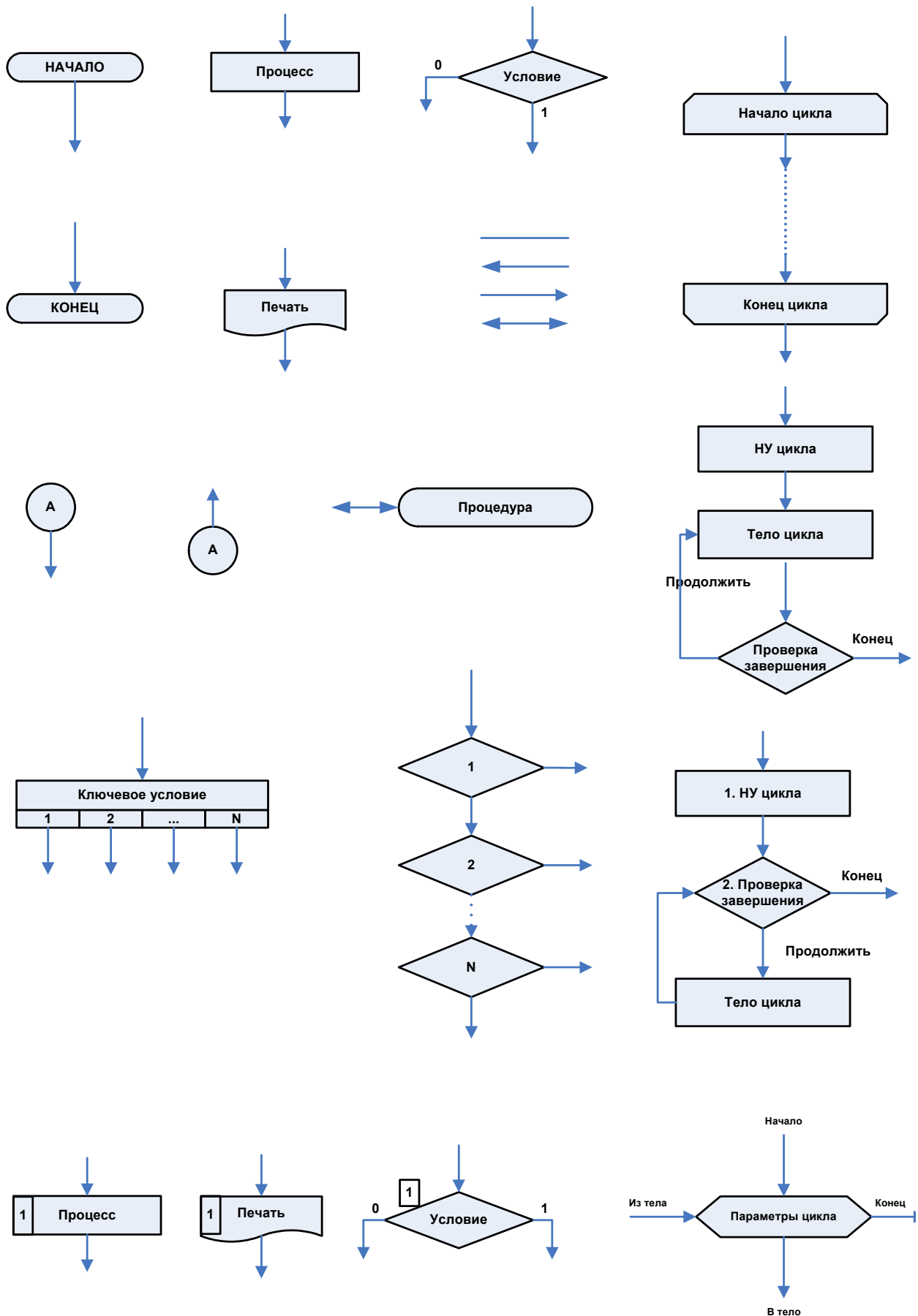
ОГЛАВЛЕНИЕ

ОГЛАВЛЕНИЕ	1
1. Лекция №1 Элементы блок-схем :	3
1.1. Элементы блок - схем.	3
1.2. Пример простейшей программы на языке СИ.	4
1.3. Простой пример алгоритма и блок-схемы.	4
1.4. Пример программы линейного вычисления с вводом выводом данных.	5
1.5. Проекты и их создание.	6
1.6. Русификация проекта консольного ввода и вывода.	8
1.7. Машина фон Неймана (и Тьюринга). Команды, инструкции и операторы.	8
1.8. Отладка программ.	10
2. Лекция № 2 - Линейные программы Основные понятия.	11
2.1. Линейные вычислительные процессы.	11
2.2. Пример простейшей программы на языке СИ.	11
2.3. Программные проекты в системах программирования на СИ.	11
2.4. Создание консольного программного проекта в СП на СИ.	12
2.5. Русификация проекта консольного ввода и вывода.	13
2.6. Программы и программирование	13
2.7. Переменные	15
2.8. Константы	16
2.9. Операторы и составные операторы.	16
2.10. Форматированный ввод-вывод.	17
2.11. Отладка программ.	19
2.12. Пример программы линейного вычисления с вводом выводом данных.	19
3. Лекция № 3 Разветвляющиеся вычислительные процессы.	22
3.1. Разветвляющиеся вычислительные процессы.	22
3.2. Безусловный переход (goto)	22
3.3. Ветвление и операторы ветвления (if - else).	23
3.4. Переключатели (switch - case)	25
3.5. Циклы (for, while, do)	26
3.6. Отладка программ.	29
3.7. Модули	29
3.8. Понятие о блок-схемах	31
3.9. Библиотеки функций	31
3.10. Примеры программ с ветвлением и циклами.	32
4. Лекция № 4 – Массивы. Основные понятия	35

4.1. Основные понятия	35
4.2. Примеры по теме ЛР № 3	43
5. Лекция № 5 – Строки. Основные понятия	46
5.1. Основные понятия	46
5.2. Преобразование данных в строку и обратно	53
5.3. Сортировка строк.....	53
5.4. Динамические строки	55
5.5. Библиотеки функций и классы для строк	56
5.6. Примеры программы с использованием строк	57
6. Лекция № 6 – Функции. Основные понятия	62
6.1. Основные понятия	62
6.2. Примеры программы с использованием функций.....	78
7. Лекция № 7 – Структуры. Основные понятия	83
7.1. Основные понятия	83
7.2. Примеры программы с использованием структур.....	99
8. Лекция № 8 – Файлы. Основные понятия	110
8.1. Основные понятия	110
8.2. Примеры программы с использованием файлового ввода и вывода.....	132
9. Лекция № 9 – Списки. Основные понятия	147
9.1. Основные понятия	147
9.2. Примеры программы с использованием файлового ввода и вывода.....	173
10. Литература.....	190

1. Лекция №1 Элементы блок-схем :

1.1. Элементы блок - схем.



1.2. Пример простейшей программы на языке СИ.

Пример простейшей программы на языке СИ приведен ниже. В строках – комментариях, помеченных слешами («//») даны пояснения для каждой строки программы. Для выполнения этой программы ее необходимо ставить в главный модуль проекта.

```
// Заголовочный файл библиотеки ввода и вывода.(это комментарий)
#include <stdio.h>

// Название главной програвмы на СИ
void main(void)
{
    // Вызов функции вывода данных на экран
    printf("HELLO!!!\n");
    // Вызов функции ввода символа с клавиатуры
    getchar();
    //
}
```

Результат работы этой программы:

HELLO!!!

Данная программа выводит на экран командной строки текст - "**HELLO!!!**", переводит строку и ожидает нажатия любой клавиши на клавиатуре. После этого она завершает работу.

1.3. Простой пример алгоритма и блок-схемы.

Ниже дан пример простейшей программы на языке C (СИ):

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

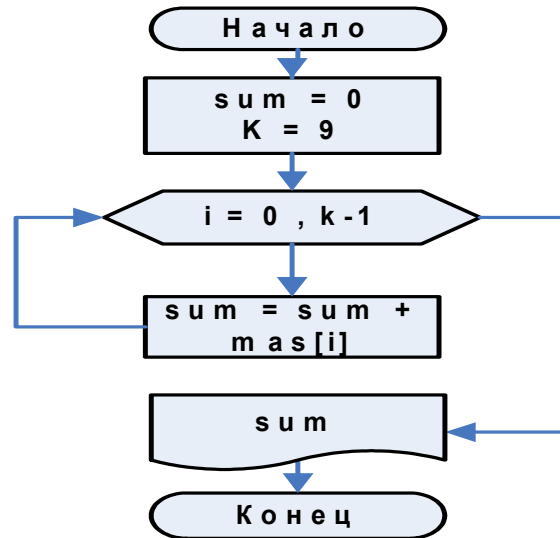
void main(void)
{
    system(" chcp 1251 > nul");
    // НУ цикла
    int sum = 0 ;
    int mas[9] = {1,2,3,4,5,6,7,8,9} ;
    int k = 9;
    // Цикл суммирования
    for (int i = 0 ; i < k ; i++ )
        sum = sum + mas[i]; // тело цикла
    printf("Сумма sum = %3d \n" , sum );
    //
    system(" PAUSE");
}
```

};

Результат работы этой программы:

Сумма `sum = 45`

Для продолжения нажмите любую клавишу . . .



1.4. Пример программы линейного вычисления с вводом выводом данных.

Пусть необходимо создать программу для вычисления значения переменной F (переменной с плавающей точкой) на основе следующей формулы:

$$F = 2ab \sin(l + 0.5\pi)$$

Параметры вычисления **a** и **b** являются целыми переменными (тип - **int**), а переменная **l** является переменной с плавающей точкой (вещественной переменной). Параметры вычисления должны быть введены с клавиатуры (с консоли), а результат должен быть выведен в окно командной строки. Тогда можно создать программу, приведенную ниже и выполнить необходимые вычисления. Пояснения в тексте программы даны в виде комментариев(`//`).

Текст программы (один исходный модуль):

```

#define _USE_MATH_DEFINES
// Подключение библиотеки математических функций
#include <math.h>
#include <stdio.h>
#include <process.h>
// описание константы для числа пи
#define PI 3.14f
// Начало программы – точка входа в программу

```

```

void main(void)
{
    // Руссификация ввода и вывода в консольном окне
    system(" chcp 1251 > nul");
    // Описание исходных данных и вычисляемых переменных
    int a , b; // переменные целого типа
    float F , l ; // переменные вещественного типа
    // Ввод данных a
    printf("Введите a: ");
    scanf_s("%d", &a);
    // Ввод данных b
    printf("Введите b: ");
    scanf_s("%d", &b);
    // Ввод данных l
    printf("Введите l: ");
    scanf_s("%f", &l);
    // Вычисление по формуле
    F = 2*a*b*sin(l+0.5f*PI);
    // Вывод результата работы программы
    printf("\nF = %7.2f для a = %d b = %d l = %5.2f \n" , F , a , b , l);
    // Ожидание завершения работы программы
    system(" PAUSE");
};

```

Результаты работы такой программы после ввода необходимых переменных выглядят так (проверьте ее работу на аналогичных данных):

```

Введите a: 2
Введите b: 3
Введите l: 10
F =  -10.07 для a = 2  b = 3  l = 10.00
Для продолжения нажмите любую клавишу . . .

```

1.5. Проекты и их создание.

Для профессионального программирования в системах программирования создается проект, содержащий исходные модули (текстовые файлы) программы. Модули могут быть двух видов: программные (файлы имеют расширения *.cpp или *.c) и заголовочные (файлы имеют расширения *.hpp или *.h). При компиляции и компоновке проекта используются единые настройки для всех модулей (1). Компилируются только те модули, которые с предыдущей компиляции изменялись (2). Это позволяет экономить

время на новое построение проекта (сборку -build). Важным свойством проектов является то, что настройки, сделанные один раз для этого проекта сохраняются для следующих запусков проекта(3). Цифрами в конце предложений помечены главные свойства и преимущества программных проектов. Кроме того, отмечу, что проекты могут быть разных типов: консольные проекты, проекты для Windows – приложений, WEB – проекты для Интернет, и многие другие. Мы будем использовать консольные проекты для изучения основ программирования.

Для создания консольного проекта необходимо:

- Запустить систему программирования VS 2005/8/10/12;
- В меню **“File”** выбрать пункт **“New”** и в подменю выбрать позицию **“Project...”**;
- В списке **“Project types”** выбрать **“Visual C++/Win32”**, а в списке **“Templates”** выбрать **“Win32 Console Application”**;
- В поле **“Name”** ввести: LAB1_XDD (где X – номер группы, а DD – номер варианта по журналу группы текущего семестра. Например, для студента группы ИУ5-31 с вариантом 5 – введем – LAB1_15). Далее нажать **“OK”**;
- В новом окне мастера проектов нажать **“Next”**. Проверить настройки проекта: **“Application Type”** должно быть – **“Console Application”**, **“Additional option”** -> **“Empty Project”** должен быть включено. Остальные галочки должны быть выключены.
- Далее необходимо нажать кнопку **“Finish”**. Новый проект будет создан.
- Далее нужно добавить исходный модуль в проект: модуль LAB1_XDD.CPP (у нас это модуль LAB1_15.CPP) добавляется в раздел **“Source Files”**. Нажмем правую кнопку на этом тексте, а затем: **“Add”** -> **“New Item ...”** -> **“Code”** -> **“C++ File”** -> Ввод поля **“Name”**;
- Далее нужно добавить заголовочный модуль в проект: LAB1_XDD.H (у нас в примере LAB1_15.H) - в раздел **“Header Files”**. Нажмем правую кнопку на этом тексте, а затем: **“Add”** -> **“New Item ...”** -> **“Code”** -> **“Header File”** -> Ввод поля **“Name”**;
- В файл LAB1_15.CPP нужно занести информацию расположенную ниже:

```
#include "lab1_15.h" // Поправить индекс группы и вариант
#include <process.h>
#include <stdio.h> // Подключение библиотеки ввода вывода
void main(void)
{
// ...

}
```

- Файл LAB1_15.H можно оставить пустым.

- Для контроля правильности создания пустого проекта, нажмем клавишу “F7” для проверки возможности создания программы (**build**) и “F5” для проверки ее выполнения (run/debug). Все перечисленные действия и операции должны быть выполнены без ошибок и предупреждений со стороны системы программирования СИ.

1.6. Русификация проекта консольного ввода и вывода.

Для корректного отображения текстов на русском языке и его ввода в окне командной строки (после первого запуска программы) нужно сделать настройки шрифта этого окна. Переключаем шрифт в тип - Lucida Console. Выбираем настройки (после вывода консольного окна на экран, правой кнопкой вызываем системное меню): СВОЙСВА->ШРИФТ -> Lucida Console). После переключения шрифта, на запрос в отдельном окошке нужно выбрать режим – “Для всех окон с данным именем!”. Для правильной русификации окна консоли, кроме этого, в самом начале главной программы нужно переключить кодовую страницу для вывода:

```
system(" chcp 1251 > nul");
```

Для приостановки завершения программы в консольном окне в конце ее работы можно вызвать паузу следующим образом (например, в конце текста программы):

```
system(" PAUSE");
```

На экране появиться следующая строка (смотри ниже) и программа будет ожидать нажатия клавиши:

Для продолжения нажмите любую клавишу . . .

Примечание. Обратите внимание на то, что при другом способе локализации (setlocale(0,"rus")); не все в программе работает правильно. Вывод на консоль и ввод с консоли выполняется правильно, но после этого введенные в консольном окне данные (например, строка) имеют другую кодировку и выводятся неверно! Можете сами это проверить. Поэтому предпочтительно использовать предложенный выше способ с переключением кодовой страницы.

Примечание. Если вы затрудняетесь выполнить заданный пункт ЛР, обратитесь к разделу “Основные понятия”, где приведены примеры для иллюстрации данного пункта.

Примечание. Все описания функций и общих данных нужно выполнять в заголовочном файле LAB1_XDD.H (у нас в примере LAB1_15.H). Программу нужно разместить в основном файле: LAB1_XDD.CPP (LAB1_15.CPP).

Далее будет представлено несколько разделов, понятия которых также будут разъяснены на лекции, но они необходимы для выполнения первой ЛР.

1.7. Машина фон Неймана (и Тьюринга). Команды, инструкции и операторы.

Принцип однородности памяти

Команды и данные хранятся в одной и той же памяти и внешне в памяти неразличимы. Распознать их можно только по способу использования; то есть одно и то же значение в ячейке памяти может использоваться и как данные, и как команда, и как

адрес в зависимости лишь от способа обращения к нему. Это позволяет производить над командами те же операции, что и над числами, и, соответственно, открывает ряд возможностей. Так, циклически изменяя адресную часть команды, можно обеспечить обращение к последовательным элементам массива данных. Такой прием носит название модификации команд и с позиций современного программирования не приветствуется. Более полезным является другое следствие принципа однородности, когда команды одной программы могут быть получены как результат исполнения другой программы. Эта возможность лежит в основе трансляции — перевода текста программы с языка высокого уровня на язык конкретной вычислительной машины.

Принцип адресности

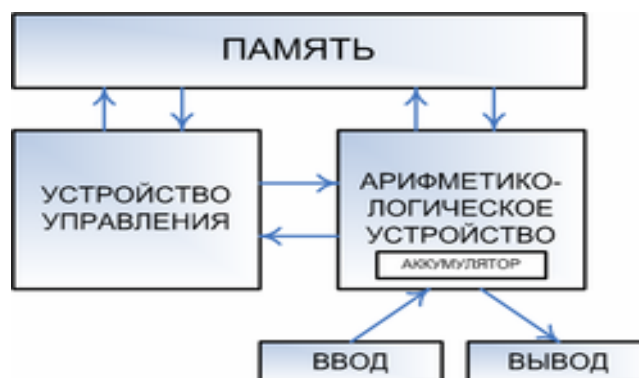
Структурно основная память состоит из пронумерованных ячеек, причем процессору в произвольный момент доступна любая ячейка. Двоичные коды команд и данных разделяются на единицы информации, называемые словами, и хранятся в ячейках памяти, а для доступа к ним используются номера соответствующих ячеек — адреса.

Принцип программного управления

Все вычисления, предусмотренные алгоритмом решения задачи, должны быть представлены в виде программы, состоящей из последовательности управляющих слов — команд. Каждая команда предписывает некоторую операцию из набора операций, реализуемых вычислительной машиной. Команды программы хранятся в последовательных ячейках памяти вычислительной машины и выполняются в естественной последовательности, то есть в порядке их положения в программе. При необходимости, с помощью специальных команд, эта последовательность может быть изменена. Решение об изменении порядка выполнения команд программы принимается либо на основании анализа результатов предшествующих вычислений по условиям либо безусловно.

Принцип двоичного кодирования

Согласно этому принципу, вся информация, как данные, так и команды, кодируются двоичными цифрами 0 и 1. Каждый тип информации представляется двоичной последовательностью и имеет свой формат. Последовательность битов в формате, имеющая определенный смысл, называется полем. В числовой информации обычно выделяют поле знака и поле значащих разрядов. В формате команды можно выделить два поля: поле кода операции и поле адресов.



1.8. Отладка программ.

При разработке программ важную роль играет отладчик, который встроен в систему программирования. В режиме отладки можно проверить работоспособность программы и выполнить поиск ошибок самого разного характера. Отладчик позволяет проследить ход (по шагам) выполнения программы и одновременно получить текущие значения всех переменных и объектов программы, что позволяет установить моменты времени (и операторы), в которые происходит ошибка и предпринять меры ее устранения. В целом, отладчик позволяет выполнить следующие действия:

- Запустить программу в режиме отладки без трассировки по шагам (**F5**);
- Выполнить программу по шагам (**F10**);
- Выполнить программу по шагам с обращениями к вложенным функциям(**F11**);
- Установить точку останова (**BreakPoint – F9**);
- Выполнить программу до первой точки останова (**F5**);
- Просмотреть любые данные в режиме отладки в специальном окне (**locals**);
- Просмотреть любые данные в режиме отладки при помощи мышки;
- Изменить любые данные в режиме отладки в специальном окне (**locals** и **Watch**);
- Установить просмотр переменных в специальном окне (**Watch**);
- Просмотреть последовательность и вложенность вызова функций.

При выполнении всех лабораторных курса студенты должны активно использовать отладчик VS, знать его возможности и отвечать на контрольные вопросы, связанные с отладкой и тестированием программ.

Лекция 2 – Линейные программы

2. Лекция № 2 - Линейные программы Основные понятия**2.1. Линейные вычислительные процессы.**

Линейные программы состоят из последовательности операторов ввода-вывода и операторов присваивания, которые выполняются в том порядке, в каком они записаны. Текст программы на СИ может быть разделен на несколько исходных файлов (в проекте), каждый из которых представляет собой текстовый файл, содержащий либо всю программу, либо ее часть. При компиляции исходной программы каждый из ее составляющих файлов компилируется отдельно, а затем связывается компоновщиком с другими файлами. Отдельные файлы можно объединить в один посредством директивы препроцессора **include**. Иногда удобно в одном файле размещать переменные, а в других файлах использовать эти переменные путем их объявления. Каждая программа содержит главную функцию **main**. Если программа аргументов командной строки не содержит, то функцию **main** можно объявить без параметров.

2.2. Пример простейшей программы на языке СИ.

Пример простейшей программы на языке СИ приведен ниже. В строках – комментариях, помеченных слешами («//») даны пояснения для каждой строки программы. Для выполнения этой программы ее необходимо ставить в главный модуль проекта.

```
// Заголовочный файл библиотеки ввода и вывода.(это комментарий)
#include <stdio.h>
// Название главной программы на СИ
void main(void)
{
    // Вызов функции вывода данных на экран
    printf("HELLO!!!\n");
    // Вызов функции ввода символа с клавиатуры
    getchar();
    //
}
```

Данная программа выводит на экран командной строки текст - "HELLO!!!", переводит строку и ожидает нажатия любой клавиши на клавиатуре. После этого она завершает работу.

2.3. Программные проекты в системах программирования на СИ.

Для профессионального программирования в системах программирования создается проект, содержащий исходные модули (текстовые файлы) программы. Модули могут быть двух видов: программные (файлы имеют расширения ***.cpp** или ***.c**) и заголовочные (файлы имеют расширения ***.hpp** или ***.h**). При компиляции и компоновке проекта используются единые настройки для всех модулей (1). Компилируются только те модули, которые с предыдущей компиляции изменялись (2). Это позволяет экономить время на новое построение проекта (сборку -build). Важным свойством проектов является то, что настройки, сделанные один раз для этого проекта сохраняются для следующих запусков проекта(3). Цифрами в конце предложений помечены главные свойства и преимущества программных проектов. Кроме того, отмечу, что проекты могут быть разных типов: консольные проекты, проекты для **Windows** – приложений, **WEB** – проекты Интернет, и многие другие. Мы будем использовать консольные проекты для изучения основ программирования на языке СИ.

2.4. Создание консольного программного проекта в СП на СИ.

Для создания консольного проекта необходимо:

– Запустить систему программирования VS 2005/8/10/12;

В меню **“File”** выбрать пункт **“New”** и в подменю выбрать позицию **“Project...”**;

В списке **“Project types”** выбрать **“Visual C++/Win32”**, а в списке **“Templates”** выбрать **“Win32 Console Application”**;

В поле **“Name”** ввести: LAB1_XDD (где X – номер группы, а DD – номер варианта по журналу группы текущего семестра. Например, для студента группы ИУ5-31 с вариантом 5 – введем – LAB1_15). Далее нажать **“OK”**;

В новом окне мастера проектов нажать **“Next”**. Проверить настройки проекта: **“Application Type”** должно быть – **“Console Application”**, **“Additional option”** -> **“Empty Project”** должен быть включено. Остальные галочки должны быть выключены.

Далее необходимо нажать кнопку **“Finish”**. Новый проект будет создан.

Далее нужно добавить исходный модуль в проект: модуль LAB1_XDD.CPP (у нас это модуль LAB1_15.CPP) добавляется в раздел **“Source Files”**. Нажмем правую кнопку на этом тексте, а затем: **“Add”** -> **“New Item ...”** -> **“Code”** -> **“C++ File”** -> Ввод поля **“Name”**;

Далее нужно добавить заголовочный модуль в проект: LAB1_XDD.H (у нас в примере LAB1_15.H) - в раздел **“Header Files”**. Нажмем правую кнопку на этом тексте, а затем: **“Add”** -> **“New Item ...”** -> **“Code”** -> **“Header File”** -> Ввод поля **“Name”**;

– В файл LAB1_15.CPP нужно занести информацию расположенную ниже:

```
#include "lab1_15.h" // Поправить индекс группы и вариант
#include <process.h>
#include <stdio.h> // Подключение библиотеки ввода вывода
void main(void)
{
    // ...
}
```

- Файл LAB1_15.H можно оставить пустым.
- Для контроля правильности создания пустого проекта, нажмем клавишу “F7” для проверки возможности создания программы (**build**) и “F5” для проверки ее выполнения (run/**debug**). Все перечисленные действия и операции должны быть выполнены без ошибок и предупреждений со стороны системы программирования СИ.

2.5. Русификация проекта консольного ввода и вывода.

Для корректного отображения текстов на русском языке и его ввода в окне командной строки (после первого запуска программы) нужно сделать настройки шрифта этого окна. Переключаем шрифт в тип - Lucida Console. Выбираем настройки (после вывода консольного окна на экран, правой кнопкой вызываем системное меню): СВОЙСВА->ШРИФТ -> Lucida Console). После переключения шрифта, на запрос в отдельном окошке нужно выбрать режим – “Для всех окон с данным именем!”. Для правильной русификации окна консоли, кроме этого, в самом начале главной программы нужно переключить кодовую страницу для вывода:

```
system(" chcp 1251 > nul");
```

Для приостановки завершения программы в консольном окне в конце ее работы можно вызвать паузу следующим образом (например, в конце текста программы):

```
system(" PAUSE");
```

На экране появиться следующая строка (смотри ниже) и программа будет ожидать нажатия клавиши:

Для продолжения нажмите любую клавишу . . .

Примечание. Обратите внимание на то, что при другом способе локализации (setlocale(0,"rus");) не все в программе работает правильно. Вывод на консоль и ввод с консоли выполняется правильно, но после этого введенные в консольном окне данные (например, строка) имеют другую кодировку и выводятся неверно! Можете сами это проверить. Поэтому предпочтительно использовать предложенный выше способ с переключением кодовой страницы.

Примечание. Если вы затрудняетесь выполнить заданный пункт ЛР, обратитесь к разделу “Основные понятия”, где приведены примеры для иллюстрации данного пункта.

Примечание. Все описания функций и общих данных нужно выполнять в заголовочном файле LAB1_XDD.H (у нас в примере LAB1_15.H). Программу нужно разместить в основном файле: LAB1_XDD.CPP (LAB1_15.CPP).

Далее будет представлено несколько разделов, понятия которых также будут разъяснены на лекции, но они необходимы для выполнения первой ЛР.

2.6. Программы и программирование

Программа – это упорядоченная совокупность операторов языка (S), которые определяют действия (шаги) для реализации поставленной задачи на основе разработанного алгоритма. Программа в самом общем виде предназначена для

преобразования информации (данных), поэтому при программировании значительную роль играют данные и их структуры.

Схематично любая программа может быть представлена так:

< S1 , S2 , S3 , ..., Si ,... , Sk>

Здесь **Si** - это либо директива описания переменных, либо оператор вычисления значений, либо оператор управления, либо оператор вызова функций. Так как операторы и директивы записаны в определенном порядке, то они выполняются, последовательно. Специальные операторы (ветвления, циклов, вызова функций) могут изменить последовательность выполнения операторов. Правила записи операторов определяются конкретным языком программирования (точнее его синтаксисом), которые формально и точно определяют способы записи операторов и описаний данных. В нашем курсе мы используем язык C/C++, хотя многие понятия являются общими и для других языков программирования.

Программирование – это процесс создания программ для компьютера. Для этой работы применяются специальные программные комплексы – системы программирования. Системы программирования включают в себя много разных сервисных программ (например, компиляторов, отладчиков и т.д.), которые упрощают работу и делают ее более производительной. Кроме этого в системы программирования включаются специальные библиотеки, значительно упрощающие процесс написания сложных программ и их отладку. Наличие в библиотеках различных программ, готовых к использованию, и которые можно подключить во вновь разрабатываемые программы, делает процесс программирования более эффективным. Библиотечные программы называют подпрограммами или функциями.

Процесс программирования включает следующие этапы:

- Осмысливание задачи, которую нужно решить путем создания программы;
- Разработка алгоритма решения задачи;
- Выбор подходящего языка программирования для реализации программы;
- Разработка формализованных описаний алгоритма и логического проекта (блок-схемы и диаграммы классов, диаграммы объектов);

Написание программы на языке программирования и безошибочный ввод ее в компьютер;

Отладка программы, включающая все шаги (компиляции, редактирования связей, создание исполнимого модуля, пошагового исполнения программы с контролем промежуточных и окончательных результатов).

Оформление документации на программный продукт и предоставление ее заказчику.

Эти этапы могут быть простыми для простых задач и трудоемкими для сложных проектов, они могут занимать продолжительные периоды времени.

2.7. Переменные

При написании (создании) программ, как было сказано выше, большое значение имеют данные, которые могут для разных целей представляться в разной форме. Форма (или формат) представления данных называется **типом** данных. Для работы с данными в программе им присваиваются специальные имена, которые иногда называются идентификаторами данных. Данные в программе могут быть постоянными и изменяемыми. Постоянные данные называются **константами** (см. ниже). Изменяемые данные называются **переменными**. Таким образом, переменной называется элемент программы, который имеет уникальное имя в программе и специальный тип. Программа работает с именами переменных, которые программист может задавать сам. Отметим, что переменным соответствуют области оперативной памяти компьютера, в которых запоминаются их текущие значения во время выполнения программы. Тип переменным необходим для указания операторам программы того, какие действия над конкретными переменными можно выполнять и сколько оперативной памяти нужно выделить для них размещения. В языке C++ используется простая форма для описания переменных:

<тип переменной > <имя/название/идентификатор переменной >;

В язык заложен набор стандартных типов переменных (int, long, char и т.д.) и специальные механизмы описания новых типов. Для описания новых типов переменных используются классы. Стандартные библиотеки языка также предлагают большой набор новых типов и операций над ними. С классами мы познакомимся в других ЛР. Переменные в программе описываются по следующему правилу:

<тип переменной > <имя переменной> = < значение для инициализации>;

Примеры описания простых переменных стандартных типов C++ показаны ниже:

```
// Простые переменные разных типов
int j = 5; // переменная целого типа
unsigned char c = 'A'; // переменная символьного типа
long l; // переменная целого типа длинная
float f = 5.5f; // переменная вещественного типа
double d = 10.00; // переменная вещественного типа длинная
bool b = true; // переменная логического типа (в базовом СИ нет такого типа)
string s; // переменная типа строка из библиотеки STL (в базовом СИ нет)
```

Подчеркну еще раз, что для переменных выделяется специальная область в оперативной памяти (ОП), в которой можно записывать и перезаписывать значения данных. Напомню, что оперативная память это специальное устройство компьютера, которое непосредственно связано с микропроцессором и служит для хранения программ и данных. Для доступа к переменным и командам используются адреса в оперативной памяти. В C++ можно явно использовать в программах адреса переменных. Для этого применяются специальные типы переменных: указатели и ссылки.

2.8. Константы

Кроме переменных, в операторах и операциях можно использовать константы. Константы не могут изменяться и в большинстве случаев не хранятся в оперативной памяти. Они заменяются в исходном тексте программы. Константы записываются на основе правил, определенных в языке и бывают двух основных типов: числовые и символьные. Для группового описания констант используются перечисления (**enum**). Примеры использования констант показаны ниже.

```
int i = 5; // константа целого типа при инициализации переменной,
double d = 5.2; // вещественная константа, используемая для инициализации
d = d*5 + 11.7; // константы в выражении
string Str1 = "Пример строки 1 "; // строковая константа в " ... "
```

В языке C++ можно использовать также переменные константного типа. Для этого используется специальное ключевое слово **const**. Для таких переменных также выделяется оперативная память, но в программе их изменять нельзя. Константные переменные должны быть обязательно инициализированы. Примеры:

```
const int i = 3; // константа целого типа,
const double d = 3.3; // вещественная константа, используемая
// для инициализации
const float f = 5.5F; // вещественная константа float, нужна
// спецификация ("F") константы
```

2.9. Операторы и составные операторы

Для работы с константами и переменными используются операторы программы (S_i). Операторы подразделяются на две группы: операторы изменения переменных и операторы управления. Основным оператором изменения переменных, является оператор присваивания, который имеет вид:

<левая часть выражения присваивания> = <правая часть выражения присваивания>;

В левой части оператора присваивания (в общем случае выражения) чаще записывается конкретная переменная, а в правой части выражение аналогичного типа. Например:

```
a = b + c; // a, b, c - переменные одного типа
Str3 = Str1 + Str2; // переменные Str1, Str2, Str3 имеют тип string (для C++)
```

Операторы управления рассмотрим ниже. В языках структурного программирования используется понятие составного оператора. Составной оператор – это любая совокупность операторов заключенная в специальные операторные скобки. В C++ операторными скобками являются фигурные скобки (“{”, “}”). В других языках программирования могут быть и другие операторные скобки (например, **begin** и **end**). В

принципе, существуют языки, в которых не применяются операторные скобки и для объединения операторов используется специальное структурное текстовое представление программы (например, **OUTLINE** как в **WORD**). Составной оператор можно записать так:

{ S1 , S2 , S3 , ..., Si,... , Sk } ;

Где **Si** - любые операторы и директивы описания языка C++.

Примеры составных операторов:

```
{ i = 5 ; c = (a>b) ? a : b; fun(a , 15); };
{
    if ( a > b)
        { int i =5; a = 0; b = 15 + i;}
    else
        {
            a = 15; b = 0; };
};
```

В данном примере использованы также операторы ветвления (if – else и условный оператор () ? :).

В языке программирования C/C++ состав стандартных операторов значительно ограничен. Он легко запоминается. Перечень этих операторов дан ниже (в СИ):

- Оператор присваивания “-”.
- Операторы ветвления (**if – else**).
- Операторы цикла (**for, do, while**).
- Оператор переключатель (**switch**).
- Оператор вызова функций и процедур .
- Оператор возврата из процедур (**return**).
- Вспомогательные операторы управления (**break, continue, case**)

В данной работе мы будем изучать оператор присваивания, который используется для вычисления значений переменных программы.

2.10. Форматированный ввод-вывод.

Функция **printf** (вывод по формату), является основной функцией вывода на консоль в языке СИ. Первый аргумент функции – строка форматирования, она задается в двойных кавычках. Эта строка выводит текст и определяет формат вывода переменных. Пример вывода текста без переменных:

```
printf("Пример сообщения"); // выведет на печать текст:
```

Пример сообщения

А функция выводит текст и значение одной переменной целого типа (**i**):

```
printf("Длина равна = %4d см", i); // при i = 1 выведет:
Длина равна =    1см
```

Таким образом, запись **"%4d"** означает: печатать заданную вторым аргументом аеугwbb величину в десятичной (**d**) форме в следующие 4 позиции. Кроме десятичного спецификатора формата вывода могут быть применяться и другие:

- о -преобразование в восьмеричный формат;
- х-преобразование в шестнадцатеричный формат;
- п-используется для запоминания текущей позиции в строке вывода;
- s-печать в строковом формате;
- с-печать один символ.

Пусть р-указатель на строку "Первое значение", а переменная х- содержит вещественное число равное 5.8134, тогда оператор:

```
printf("%s%6.3f", p , x );
```

выведет на печать:

Первое значение 5.813.

Под число с плавающей точкой отведено **6** позиций, три из которых выделено под дробную часть. Это справедливо для всех типов данных. Спецификатор **f** используется для вывода данных типа **float** (число с плавающей точкой). Для вывода числа типа **double** нужно указать формат для вывода - **%lf**. Форматная строка в функции **printf** может содержать специальные управляющие символы:

- \n- перевод на новую строку;
- \f-новая страница;
- \t-табуляция;
- \b-стереть предыдущий символ и т.д.

В языке СИ для ввода данных используются различные функции, в том числе **getchar()** и **scanf()**:

```
char c;
c=getchar();
printf("Символ = %hc !!!\n", c );
```

Здесь **c** - переменная типа **char**, ей будет присвоено значение, введенное с клавиатуры. Ввод с помощью функции **scanf**. В этом случае вводимые переменные задаются с помощью указателей. Пусть переменные вводятся переменные: **a**, **letter** и **n**:

```
int a,n;
char letter;
scanf("%4d%c%2d",&a, &letter,&n);
```

Если мы вводим с клавиатуры в окне командной строки следующие данные:

3162 y 47,

тогда функция **scanf** присвоит переменным значения соответствующие значения: a=3162, letter='y', n=47. (Знак & перед переменной, определяет использование указателя-адреса переменной, но об этом речь пойдет в других ЛР). Все аргументы представляют собой указатели. При вводе функция **scanf** рассматривает пробелы как разделители вводимых данных. Красным цветом выделена символьная переменная (**letter**) и ее значения.

Примечание: Для систематических сведений о работе функций можно воспользоваться также: литературой по курсу, лекциями по курсу, MSDN и другими источниками информации.

2.11. Отладка программ.

При разработке программ важную роль играет отладчик, который встроен в систему программирования. В режиме отладки можно проверить работоспособность программы и выполнить поиск ошибок самого разного характера. Отладчик позволяет проследить ход (по шагам) выполнения программы и одновременно получить текущие значения всех переменных и объектов программы, что позволяет установить моменты времени (и операторы), в которые происходит ошибка и предпринять меры ее устранения. В целом, отладчик позволяет выполнить следующие действия:

- Запустить программу в режиме отладки без трассировки по шагам (**F5**);
- Выполнить программу по шагам (**F10**);
- Выполнить программу по шагам с обращениями к вложенным функциям(**F11**);
- Установить точку останова (**BreakPoint – F9**);
- Выполнить программу до первой точки останова (**F5**);
- Просмотреть любые данные в режиме отладки в специальном окне (**locals**);
- Просмотреть любые данные в режиме отладки при помощи мышки;
- Изменить любые данные в режиме отладки в специальном окне (**locals** и **Watch**);
- Установить просмотр переменных в специальном окне (**Watch**);
- Просмотреть последовательность и вложенность вызова функций.

При выполнении всех лабораторных курса студенты должны активно использовать отладчик **VS**, знать его возможности и отвечать на контрольные вопросы, связанные с отладкой и тестированием программ.

Примечание. Подробное описание материала и понятий вы можете найти в литературе [1 - 6] или справочной системе MS VS. Кроме того, не пропускайте лекции по курсу. Не рекомендую безоговорочно верить материалам из сети Интернет (например, в Википедии), так как там в некоторых статьях есть ошибки!

2.12. Пример программы линейного вычисления с вводом выводом данных.

Пусть необходимо создать программу для вычисления значения переменной F (переменной с плавающей точкой) на основе следующей формулы:

$$F = 2ab \sin(l + 0.5\pi)$$

Параметры вычисления **a** и **b** являются целыми переменными (тип - **int**), а переменная **l** является переменной с плавающей точкой (вещественной переменной). Параметры вычисления должны быть введены с клавиатуры (с консоли), а результат должен быть выведен в окно командной строки. Тогда можно создать программу, приведенную ниже и выполнить необходимые вычисления. Пояснения в тексте программы даны в виде комментариев(**//**).

```
#define _USE_MATH_DEFINES
// Подключение библиотеки математических функций
#include <math.h>
#include <stdio.h>
#include <process.h>
// описание константы для числа пи
#define PI 3.14f
// Начало программы – точка входа в программу
void main(void)
{
// Руссификация ввода и вывода в консольном окне
system(" chcp 1251 > nul");
// Описание исходных данных и вычисляемых переменных
int a , b; // переменные целого типа
float F , l ; // переменные вещественного типа
// Ввод данных a
printf("Введите a: ");
scanf_s("%d", &a);
// Ввод данных b
printf("Введите b: ");
scanf_s("%d", &b);
// Ввод данных l
printf("Введите l: ");
scanf_s("%f", &l);

// Вычисление по формуле
F = 2*a*b*sin(l+0.5f*PI);
// Вывод результата работы программы
printf("\nF = %7.2f для a = %d b = %d l = %5.2f \n" , F , a , b , l);
// Ожидание завершения работы программы
system(" PAUSE");
};
```

Результаты работы такой программы после ввода необходимых переменных выглядят так (проверьте ее работу на аналогичных данных):

Введите a: 2

Введите b: 3

Введите l: 10

F = -10.07 для a = 2 b = 3 l = 10.00

Для продолжения нажмите любую клавишу . . .

Нужно создать пустой проект в MS VS, как описано выше (раздел 3.4 и 3.5), скопировать через буфер обмена в него текст данного примера, отладить его (Раздел 3.11) и выполнить.

Лекция 3 – Разветвляющиеся вычислительные процессы

3. Лекция № 3 Разветвляющиеся вычислительные процессы.

В теоретической части описания лабораторной работы вводятся основные понятия и рассматриваются принципы для работы с циклическими и условными операторами на языке программирования СИ.

3.1. Разветвляющиеся вычислительные процессы.

Традиционно операторы выполняются последовательно (машина Тьюринга). Однако реально построить более или менее сложную программу, имеющую линейную последовательность выполняемых операторов невозможно. Для этого, помимо операторов выполняющих непосредственные вычисления (напомним – операторы присваивания значений) в языках программирования предусматриваются операторы управления или, более точно, операторы, управляющие последовательностью выполнения других операторов программы.

Возможные варианты изменения последовательности выполнения операторов следующие, они обусловлены реальными потребностями программирования, а именно:

- Необходимость безусловного перехода к выполнению другого оператора, например завершающего программу (безусловный переход на метку).
- Необходимость выбора альтернативных путей выполнения программы (условный оператор и оператор переключатель выполнения групп операторов).
- Необходимость многократного повторения последовательности операторов (операторы циклического повторения).
- Вызов повторяющейся последовательности операторов, настраиваемых на заранее выделенные параметры (вызов процедур и функций).

Рассмотрим применительно к языку СИ предусмотренные в нем операторы управления.

3.2. Безусловный переход (goto)

Для безусловного перехода управления к другому оператору программы, не являющимся следующим по порядку, используется оператор `goto`. Место программы, в которое осуществляется передача управления, задается специальной меткой. Метка задается именем (идентификатором) и размещается перед оператором, к которому мы хотим перейти. Признаком метки является двоеточие, которое без пробела размещается за меткой. Формально это выглядит так:

```

goto <метка>;

.....

<метка>: <оператор>;

```

Ниже приведен фрагмент программы, в котором используются метки (Lab0, Lab1) и операторы безусловной передачи управления **goto**. Оператор передачи управления на метку Lab0 закомментирован, так как при его использовании возникает бесконечный цикл – выполнение программы никогда не остановится.

```

// переход и метка

printf("Начало программы!!\n");

Lab0:

goto Lab1;

printf("Никогда не печатается!!!\n");

Lab1:

// goto Lab0; // Бесконечный цикл – закомментировано!

printf("Переходим сюда!!\n");

```

Операторы безусловной передачи управления используются в программах на СИ только в исключительном случае. Во-первых, доказано, что любой алгоритм можно реализовать без использования этого оператора, а, во-вторых, его применение может приводить к сложным для поиска ошибкам. В наших примерах, для организации циклов мы продемонстрируем использование этого оператора.

3.3. Ветвление и операторы ветвления (if - else)

Часто в процессе программных вычислений, в зависимости от логических условий, определяемых переменными программы, необходимо выполнять либо одни, либо другие действия. Характерным примером может служить программа, для вычисления корней квадратного уравнения. В зависимости от значения дискриминанта мы получаем либо действительные, либо комплексные значения корней уравнения (пример такой программы мы рассмотрим ниже).

Условный оператор (if - else) позволяет сделать проверку и обеспечивает ветвление в программе. В условном операторе проверяется логическое условие и, в зависимости от результата выполняется либо одна группа операторов (составной оператор в общем случае), либо другая группа операторов (составных операторов). Формализовано это может быть записано так:

```

if (<логическое условие>)

{ <составной оператор, выполняемый при истинности условия>}

[ else [ if ]

```

```

{ < составной оператор, выполняемый при ложности условия > } ]
;

```

Вторая часть условного оператора необязательна, в этом случае в программе выполняется составной оператор при истинности условия, в противном случае никаких действий просто не производится. Пример условного оператора для сравнения двух переменных, в котором по результатам сравнения выводится текст о том, какая из переменных больше:

```

// условный переход (максимум из двух переменных)
int a = 5;
int b = 3;
if ( a > b) // Простое условие
    printf(" a>b !\n");
else
    printf (" a<=b !\n");

```

Условные операторы могут быть вложены, внутри одного оператора, в теле составного оператора размещаются другие. Это показано на примере, в котором, после else, вставляется новый условный оператор:

```

if ( a > b)
    printf(" a>b !\n");
else
{
    if ( a < b) // вложенный условный оператор
        printf (" a<b !\n");
    else
        printf (" a=b !\n");
};

```

Другой пример условного оператора:

```

if ( i > 5 && Flag) // проверка условия
{ iMas[i] = 0; Flag = false;} // Старшим элементам массива присвоено значение 0
else
{ iMas[i] = iMasB[i]; Flag = true;}; // Младшим элементам массива - iMasB[i]

```


3.4. Переключатели (switch - case)

Переключатели (**switch - case**) позволяют сделать выбор из множества альтернатив. Здесь для краткости мы их не рассматриваем. С ними более подробно можно познакомиться в рекомендованной литературе. Кратко поясним следующее: в заголовке оператора (**switch**) проверяется целочисленное выражение (у нас просто - **num**); в зависимости от числа выполняется сравнение с константой указанной после частью выбора (**case**); если значения совпадают, то выполняется группа операторов после **case**. Далее последовательно выполняются все операторы, которые расположены в данном переключателе (вне зависимости от сравнений **case**), если такое выполнение не прерывается с помощью оператора **break**. Если такой оператор (**break**) встретился в тексте, то далее выполняется оператор, который следует за переключателем. Если совпадений с константами **case** вообще не было, то при наличии выполняется группа операторов, которая следует за ключевым словом **default**, и проверки прекращаются. Ниже приводится переключатель, в котором значение переменной **num** проверяется последовательно с константами 1, 2, 3.

```
// Переключатель
int num = 2;
switch ( num)
{
    case 1:
        printf("Выбор 1!\n");
        break;
    case 2:
        printf("Выбор 2!\n");
        case 3:
            printf("Выбор 3!\n");
            // break
            break;
    default:
        printf("Выбор по умолчанию!\n");
};
```

Результат работы данного текста такой (проанализируйте, почему так):

Выбор 2!

Выбор 3!

В методическом пособии [6] в разделе 8 посмотрите, пожалуйста, как оформляется в блок-схемах оператор переключатель.

3.5. Циклы (for, while, do)

Циклы – это фрагменты программы, которые для достижения результата нужно повторять многократно. Циклы содержат три основных элемента:

- Начальные условия цикла, выполняемые однократно для начала цикла
- Тело цикла – повторяющиеся операторы;
- Условие, проверяющее завершение или продолжение циклических повторений.

В зависимости от того как в программе записаны эти элементы, в СИ предлагаются операторы цикла различного вида: повторить заданное число раз (**for**), повторить пока истинно условие (**while**) и выполнить, проверив условие продолжения(**do - while**). Во всех случаях начальные условия цикла могут быть заданы операторами предварительно. Условие продолжения записывается в самих операторах цикла. Тело цикла задается в виде составного оператора, который может включать и другие операторы цикла и директивы описания переменных. Такие циклы называются вложенными. Приведем примеры различных операторов цикла.

В операторе цикла **for** для управления циклом в его заголовке задаются три составляющие, разделенные знаком “;” – точка с запятой: задание начального значение переменной управления циклом (может даже включаться ее описание); проверка условия продолжения цикла и группа операторов, выполняемых после завершения каждого шага цикла перед проверкой условия. Приведенный ниже цикл (его тело – оператор печати) выполняется 5 раз для i от 1 до 5 включительно с шагом 1 (i++). Далее выполняется следующий оператор, расположенный после тела цикла.

```
// Цикл for
for (int i = 1 ; i <= 5 ; i++ )
{
    printf("Шаг цикла(for) - %d\n" , i);
};
```

В операторе цикла **while** сначала проверяется условие продолжения цикла, указанное в заголовке. Если условие истинно, то тело цикла выполняется, а затем снова проверяется условие продолжения. Когда устанавливается факт ложности условия, повторы тела цикла прекращаются. Приведенный ниже цикл выполняется 5 раз для k от 0 до 4. (Заметьте, последнее значение k равно 5-ти). Далее выполняется следующий оператор, расположенный после тела цикла.

```
// Цикл while
int k = 0;
while ( k < 5)
{
    printf("Шаг цикла (while) - %d\n" , k);
```

```

k++;
};

```

В операторе цикла **do** сначала однократно (всегда) выполняется тело цикла, а затем проверяется условие продолжения цикла (**while**), указанное после тела. Если условие истинно, то тело цикла снова выполняется, а затем снова проверяется условие продолжения. Когда устанавливается факт ложности условия, повторы тела цикла прекращаются. Далее выполняется следующий оператор, расположенный после тела цикла. Приведенный ниже цикл выполняется 2 раза для *i* от 0 до 2, шагом 2. (Заметьте, последнее значение *i* равно 4-ти). Далее выполняется следующий оператор, расположенный после тела цикла.

```

// Циклы do
int i = 0;
do
{
    printf("Шаг цикла(do) - %d\n" , i);
    i++; i++;
} while ( i < 4);

```

Оператор **break** , также как и в переключателе прерывает выполнение цикла любого типа. После его выполнения циклические повторения прекращаются и выполняется оператор, следующий за оператором цикла.

```

// Цикл for с условной остановкой на втором шаге
for (int i =1 ; i <= 5 ; i++ )
{
    printf("Шаг цикла(break) - %d\n" , i);
    if ( i == 2) break;
};

```

Оператор **continue** прерывает выполнение только текущего шага цикла. Далее цикл продолжается так, как задано по начальному условию. В примере, расположенном ниже часть 2 цикла не выполняется при значении индекса *i* равного 2-м.

```

// Цикл for с условным пропуском части операторов на втором шаге
( continue)
for (int i =1 ; i <= 5 ; i++ )
{
    printf("Шаг цикла(break) - %d" , i);
    if ( i == 2) { printf("Пропуск %d!!\n", i); continue; };
    printf("Тело - часть 2!!\n");
}

```

```
};
```

Рассмотрим и другие примеры использования операторов циклов (**for**, **while** и **do**).

Для цикла **for** (суммирование массива):

```
Summ = 0; // начальное условие
for (int i = 0 ; i < MAX; i++ ) // начальное условие и проверка
    продолжения
{
    Summ = Summ + iMas[i]; // тело цикла
    ... // Другие операторы тела цикла
};
```

Для цикла **while**, тоже вычисление суммы массива:

```
Summ = 0; // начальные условия
int i =0; // начальные условия
while (i < MAX ) // проверка продолжения
{
    Summ = Summ + iMas[i]; // тело цикла
    i++;
    ... // Другие операторы тела цикла
};
```

Для цикла **do – while** (сумма в массиве):

```
Summ = 0; // начальные условия
int i =0; // начальные условия
do {
    Summ = Summ + iMas[i]; // тело цикла
    i++;
    ... // Другие операторы тела цикла
} while (i < MAX ); // проверка продолжения
```

Большинство ошибок в циклической программе связано с неверным заданием условий проверки продолжения и завершения циклов, а также задании начальных условий цикла. В этом случае часто программа может выполняться сколь угодно долго и говорят, что программа зациклилась (заметьте, что данное слово уже вошло также в обиход обычной речи!).

3.6. Отладка программ.

В этом разделе повторяем возможности отладчика, так как демонстрация программы преподавателю должна быть выполнена в режиме отладчика. При разработке программ важную роль играет отладчик, который встроен в систему программирования. В режиме отладки можно проверить работоспособность программы и выполнить поиск ошибок самого разного характера. Отладчик позволяет проследить ход (по шагам) выполнения программы и одновременно получить текущие значения всех переменных и объектов программы, что позволяет установить моменты времени (и операторы), в которые происходит ошибка и предпринять меры ее устранения. В целом, отладчик позволяет выполнить следующие действия:

- Перекомпилировать все и сделать новую сборку (**F7**);
- Запустить программу в режиме отладки без трассировки по шагам (**F5**);
- Выполнить программу по шагам (**F10**);
- Выполнить программу по шагам с обращениями к вложенным функциям(**F11**);
- Установить точку останова (**BreakPoint – F9**);
- Выполнить программу до первой точки останова (**F5**);
- Просмотреть любые данные в режиме отладки в специальном окне (**locals**);
- Просмотреть любые данные в режиме отладки при помощи мышки;
- Изменить любые данные в режиме отладки в специальном окне (**locals** и **Watch**);
- Установить просмотр переменных в специальном окне (**Watch**);
- Просмотреть последовательность и вложенность вызова функций.

При выполнении всех лабораторных курса студенты должны активно использовать отладчик VS, знать его возможности и отвечать на контрольные вопросы, связанные с отладкой и тестированием программ.

Примечание. Подробное описание материала и понятий вы можете найти в литературе [1 - 6] или справочной системе MS VS. Кроме того, не пропускайте лекции по курсу. Не рекомендую безоговорочно верить материалам из сети Интернет (например, в Википедии), так как там в некоторых статьях есть ошибки!

3.7. Модули

При проектировании и разработке программ применяется метод модульного программирования. Суть его заключается в том, что сложная программа разбивается на отдельные части (модули – не надо путать с учебными модулями). Каждый модуль разрабатывается отдельно и возможно разными программистами. Такой способ называют также декомпозицией. Совокупность модулей составляет проект программы. Модули бывают разных типов:

- Исходные модули, содержащие текст на языке программирования.
- Объектные модули получаются в результате компиляции программы.
- Исполнимые модули предназначены для выполнения программы.

Исходные модули могут быть разных типов. Основные исходные модули – это модули программ (*.c, *.cpp) и модули заголовочных файлов (*.h, *.hpp). Они включены в разные разделы дерева проекта программы.

Объектные модули формируются компилятором языка программирования (у нас C++) в том случае, если не было ошибок в программе. Объектные модули являются промежуточным звеном в создании программ, но не могут выполняться непосредственно. Подключаемые в программу библиотеки содержат объектные модули, поэтому не требуется их повторной компиляции. Объектные модули имеют расширение *.obj.

Исполнимые модули предназначены для непосредственного выполнения на компьютере или подключения в выполняемую программу. Основные исполнимые модули имеют расширение *.exe (или *.com), поэтому операционная система может контролировать их запуск. Модули динамических библиотек имеют расширение *.dll. Существуют и другие разновидности исполнимых модулей, зависящих от используемых технологий. Исполнимые модули формируются специальной программой системы программирования редактором связей (или компоновщиком). Редактирование связей заключается в проверке межмодульных связей по функциям и по данным и объединении их единый исполнимый модуль. Последовательный процесс обработки программы для получения исходного модуля представлен в методическом пособии в разделе 3[5].

Для объединения модулей функционально и по данным в C++ используются прототипы функций и описание внешних переменных. Покажем это на простом примере:

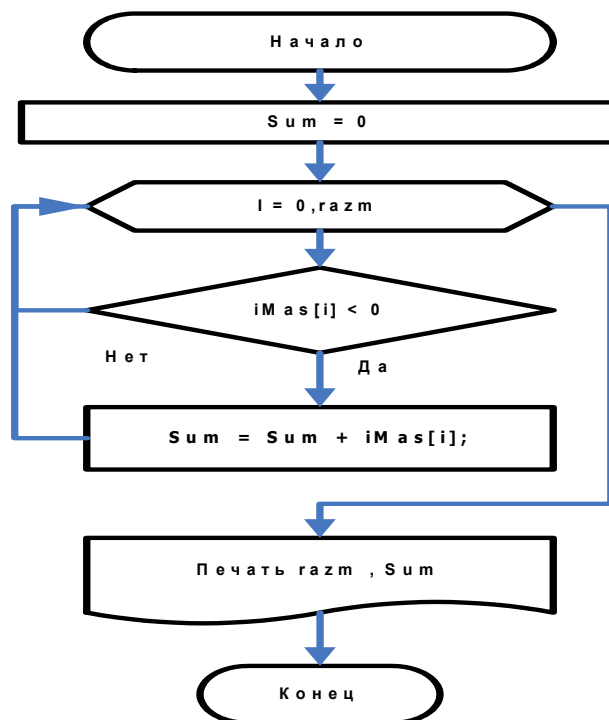
```
// Модуль 1
int i = 0; // Глобальная переменная
int Summ(int a, int b){}; // Описание функции в модуле 1
...

// Модуль 2
extern int i; // Описание внешних данных
int Summ(int , int ); // Прототип внешней функции
main(){
...
i = 5; // Использование внешних данных
S = Summ(2,2); // Вызов внешней функции
...
}
```

В модуле 2 используется переменная i, описанная как глобальная переменная в модуле 1. В модуле 2 используется функция Summ, описанная как в модуле 1. Для правильного объединения связей (работы редактора связей) используются описания внешних данных (**extern**) и задание прототипа функции Summ.

3.8. Понятие о блок-схемах

Для описания алгоритмов используются различные способы. Наиболее наглядным и простым является технология блок-схем. В лабораторных работах вам необходимо в совершенстве освоить эту технологию. В ЛР №2 фрагменты программ циклического вида и с ветвлением нужно оформить в виде блок-схемы. Блок-схемы позволяют не только логично и без ошибок построить программу, но и проверить ее работу в процессе отладки и проверки. Технология построения блок-схем подробно описана в методическом пособии к лабораторным работам по курсу[5] в разделе 8. Для графического построения блок-схем очень удобно использовать программный продукт MS Visio. Пример блок-схемы для вычисления суммы положительных элементов массива приведен ниже.



3.9. Библиотеки функций

В системах программирования предусматривается много библиотек для функций различного назначения (например, для работы со строками, выполнения ввода и вывода, работы с массивами и т.д.). Эти библиотеки подключаются с помощью заголовочных файлов или пространств описаний (имен - **namespace**). Кроме заголовочных файлов для использования библиотек подключаются специальные модули (иногда они подключаются автоматически), содержащие описания функций (*.lib или *.dll). Пример подключения библиотек ввода/вывода и библиотек для работы с математическими и системными функциями:

```
#include <math.h>
```

```
#include <process.h>
```

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

Стандартных библиотек очень много. Нужно хорошо знать их назначение и их состав для использования в программах. Чем лучше знания о библиотеках, тем быстрее и безошибочно можно создать сложную программу. В современных системах программирования доступны (большом количестве) библиотеки классов, которые описывают новые дополнительные типы данных.

3.10. Примеры программ с ветвлением и циклами.

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 ветвления (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 нужно сделать в отладчике.

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.

3.10.1. Пример поиска максимального значения из трех переменных (A , B, C)

Задавая вручную различные значения переменных **A, B, C** нужно проверить работы алгоритма по всем веткам условий (четыре варианта).

```
// условный переход
int A = 4 , B = 4 , C =1;

if ( A > B )
{
    if ( A > C )
        printf("Максимально (A) - %d\n" , A);
    else
        printf("Максимально (C) - %d\n" , C); }
else
{
    if ( B > C )
        printf("Максимально (B) - %d\n" , B);
    else
```



```
printf("Максимально (C) - %d\n" , C);
};
```

3.10.2. Решение квадратного уравнения

Задавая вручную различные значения переменных a, b, c нужно проверить работы алгоритма по всем веткам условий (два варианта). Уравнение в общем виде имеет вид. Его решение вы можете посмотреть в любом справочнике по математике. Общий вид квадратного уравнения:

$$ax^2+bx+c=0$$

Программа для вычисления корней уравнения (Напомню: корни могут быть действительными и комплексными).

```
float a,b,c;
float c,d,e,f,x1,x2;
scanf ("%5f%5f%5f",&a,&b,&c);
// Промежуточные вычисления
z=2.0f*a;
e=-b/z;
d=b*b-4*a*c; // детерминант
f=sqrt(fabs(d))/z;
// Проверка корней
if(d>=0)
{
x1=e+f;
x2=e-f;
printf("Корни уравнения действительные\n");
printf("\n x1= %4.1f",x1);
printf("\n x2= %4.1f\n",x2);
};
else
{
printf("Корни уравнения комплексные\n");
printf("\Вещественная часть корня: e= %4.1f",e);
```

```
printf("\nМнимая часть корня: f= %4.1f\n",f); };
```

3.10.3. Использование условного оператора для организации цикла

Для организации цикла может быть использован и условный оператор. Как это делается, показано в следующем фрагменте текста для вычислений суммы арифметической прогрессии.

```
// Организация цикла с помощью if
int j = 1; // индекс для цикла
int Sum = 0; // Переменная для вычисления суммы
int step = 5; // Шаг прогрессии
M1:
if ( j < 10)
{
    Sum +=j*step; // Накапливание суммы
    j++;
    goto M1;
};
// Вывод результата
printf ("\nЗначение Sum = %d для j =%d\n" , Sum, j);
```

3.10.4. Использование переключателя и операторов цикла

Проверить выполнение примеров из разделов данных методических указаний для условных и безусловных операторов (**goto**, **if**, **if-else**), переключателей и операторов цикла (**for**, **while** и **do**) с вариантами **break** и **continue**.

Примеры смотрите в разделах расположенных выше. Желательно при изучении материала создать новый консольный проект и копировать в него программы из текста этого документа.

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

4. Лекция № 4 – Массивы. Основные понятия

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

4.1. Основные понятия

4.1.1. Массивы

Переменная, которая может сохранять только одно значение, называется простой переменной. Для разработки сложных программ часто недостаточно одних простых переменных, так как в противном случае простых переменных было бы очень много с разными именами, и программист может легко запутаться в таких наименованиях. Поэтому в языках программирования введено понятие массивов (упорядоченных наборов) однородных (одного типа) переменных, доступ к которым выполняется с указанием номера отдельного элемента (индекса). Такую переменную называют также переменной с индексами или элементом массива. При описании помимо имени и типа для массивов необходимо указать его размер, который называется размерностью массива. В общем случае массив описывается так:

<тип массива > <имя массива> [<размерность массива>];

Например:

```
int iMas [10]; // описан массив iMas целого типа, содержащий 10 элементов ( номера 0:9)
char sMas [MAX]; // массив sMas символьного типа, содержащий MAX элементов
```

Размерность массива задается целой константой, переменной константного типа или переменной этапа компиляции, и при таком описании размерность массивов не может изменяться во время работы программы. Номера массива начинаются с нуля (0), поэтому последний элемент массива имеет номер (точнее индекс) на единицу меньший, чем размерность массива (в нашем случае 9). При обращении к элементам массива можно указать, как константу, так и переменную целого типа, так и выражение целого типа.

Например:

```
iMas [5] = 3; // элементу массива iMas с номером 5 (6-му) присваивается 3
sMas [i + 5] = 'A'; // элементу массива sMas с номером i (i+4) присваивается символ 'A'
```

Массивы могут иметь более одного измерения, при этом указываются значения размерности по каждому измерению отдельно в квадратных скобках:

<тип массива > <имя массива> [<размерность массива1>][<размерность массива2>] ...;

Например, двумерный массив может быть описан так:

```
int iMas [5][5]; // двумерный массив iMas целого типа, содержащий 5*5 элементов
```

При использовании переменных массивов с размерностью более 1-й должен быть указан номер (индекс) по каждому измерению (переменная с индексами – определяет в каждый момент времени один элемент массива):

```
iMas [i][j] = 10; // iMas с номерами i и j по каждому измерению присваивается 10
```

Массивы нужны для того, чтобы сделать программу более короткой, для обработки больших объемов данных и для обеспечения динамической настройки программ. Число измерений массивов в С и С++ не ограничивается.

Размерность массива в каждом измерении может быть задана (как сказано выше): целочисленной константой, переменной этапа компиляции (#define), константной переменной целого типа и целой переменной (для динамических массивов). При инициализации массивов фиксированным значением констант инициализации, размерность может быть не указана. Рассмотрим еще примеры для разных способов задания размерности массива:

```
#define N 10 // Переменная этапа компиляции
...
int Mas[N]; // Описание массива размерностью 10

const int NMax = 5; // константная переменная
int MasS[NMax]; // Описание массива размерностью 5

int MasIni[] = {1,2,3,4}; // Описание массива размерностью 4
```

Инициализация двумерных массивов:

```
int C[3][6] = { { 1,2,3,4,5,6}, { 1,2,3,4,5,6},{ 1,2,3,4,5,6} };
```

В программе можно динамически определить число элементов в массиве:

```
int iMas1[] = {3,3,2,4,5,6,0,1,9};
Razm = sizeof(iMas1)/sizeof(int);
```

Указатель – это переменная, которая содержит адрес памяти, где расположена другая переменная (см. раздел ниже) или массив переменных (Этот фрагмент раздела можно пропустить, познакомится подробнее с указателями и вернуться к нему позже). По умолчанию само имя массива трактуется как указатель на его начало. Проиллюстрируем это примером.

```
int * PtrMas; // Указатель на целую переменную
int MasP[]={ 1,2,3}; // Описание массива
```

```
PtrMas = MasP; // Указателю присваивается указатель на начало массива
int m = PtrMas[1]; // k = 2!
```

Также забегаая вперед, проиллюстрируем использование указателей для работы с динамическими массивами. Проиллюстрируем это сразу на примере. В указателе **PtrMas** запоминается адрес области динамической памяти, выделенной оператором **new**. Далее динамический массив заполняется в цикле. Затем с использованием операции разыменования (*) мы получим новое значение из массива (в переменную l). Оно равно 3.

```
PtrMas = new int [5];
for ( int k = 0 ; k < 5 ; k++)
    PtrMas[k] = k + 1; // Динамическое заполнение массива числами от 1 до 5
int l = 2 ;
*(PtrMas + l) = l + 1; //Использование указателя для занесение про [2] значения 3
(2+1)
l = PtrMas[l] ; // l = 3
```

4.1.2. Указатели

Кроме использования массивов, есть и другой способ, чтобы сделать программу динамически настраиваемой во время выполнения. Это переменные специального типа - указатели. Такие переменные содержат в качестве значения не число, а адрес другой переменной (в оперативной памяти). При описании таких переменных нужно указать специальный знак – звездочка (“*”). Кроме того, должно быть указан тип переменных, на которые данный указатель может ссылаться. Можно объявить и массив указателей и выполнить предварительную инициализацию указателя. Описание указателя:

```
<тип указателя> * <имя указателя> [= <значение для инициализации>];
```

Например:

```
// Указатели
int /*i , */ j , k ;
int *pInt; // Указатель на переменную типа int
int **ppInt; // Указатель на указатель на переменную типа int
int *pMas[10]; // Массив указателей
int aI = 5; // Простая переменная
int *pInit = &aI; // инициализация указателя
```

Для работы с указателями используются две специальные операции: операция именования (“&”) и операция разыменования (“*”). Операция именования используется для вычисления значения указателя – адреса переменной или выражения. Операция разыменования используется для получения значения переменной, адрес которой задан данным указателем. Примеры:

```
/// Задание значений и адресов
```

```

j = 15;
pInt = &j; // именованье - в указатель записывается адрес
i = *pInt; // разыменованье – берем значение переменной по указателю (j)
ppInt = &pInt;
k = **ppInt; // двойное разыменованье указателя

```

Для упрощения работы с указателями, а также для удобной перегрузки операций в классах в C++ введено понятие ссылки. Ссылка также задает адреса других переменных и объектов, но явного использования операций именования и разыменования для ссылок не требуется. Более детально со ссылками вы познакомитесь в курсе Объектно - ориентированного программирования (см. литературу[6] и MSDN).

4.1.3. Динамическая память

Часто все переменные и массивы располагаются в оперативной памяти заранее, до начала выполнения программы. Все рассмотренные ранее примеры включали такие переменные и массивы. Размер выделенной памяти под массив, то есть его размерность в этом случае изменить нельзя. Поэтому приходится заранее рассчитывать максимально возможную размерность массива, которая приемлема для всех случаев. Это приводит к перерасходу памяти для отдельных случаев. Для экономии памяти и для построения более универсальных программ используют динамически выделяемые переменные. С помощью специальных операций (**new** и **delete**) можно выделять память во время выполнения программы. Такие данные называются динамическими. Ранее в языке СИ оперативная память выделялась с помощью специальных функций (alloc, malloc и т.д.). Динамические переменные располагаются в оперативной памяти в специальной области. Для работы с такими данными используют указатели и ссылки. Пример выделения динамической памяти для указателей и ссылок:

```

int * piDyn = new int; // Выделяется область для целой переменной
int *piDMas = new int[10]; // Выделяется область для массива целых 10
переменных
int *piDMas = new int[i]; // Выделяется область для массива целых i переменных

```

Работа с указателями на динамические переменные и массивы выглядит так:

```

*piDyn = 5;
piDMas[0] = 5;

```

По завершению блока, в котором выделены динамические переменные их надо удалить специальным оператором **delete**:

```

delete piDyn;
delete []piDMas;

```

Кроме этого для работы с динамической памятью могут быть использованы функции работы с динамической памятью (библиотека **malloc.h**): выделение (**calloc** , **malloc**) и освобождения (**free**), оперативной памяти Пример:

```
#include <malloc.h>

...

int *pMasInt;

...

pMasInt = (int *) malloc ( 10 ); // выделить 10 байт

...

free( pMasInt ); // Освободить динамическую память

...

pMasInt = (int *) calloc ( 10 , sizeof (int)); // выделить для массива 10 блоков по
размеру int
pMasInt[3] = 10; // Работа с динамическим массивом
int iTest = pMasInt[3] ;

...

free( pMasInt ); // Освободить динамическую память
```

Другие возможности для работы с библиотекой вы найдете в литературе и документации по СИ.

4.1.4. Отладка программ.

В этом разделе повторяем возможности отладчика, так как демонстрация программы преподавателю должна быть выполнена в режиме отладчика. При разработке программ важную роль играет отладчик, который встроен в систему программирования. В режиме отладки можно проверить работоспособность программы и выполнить поиск ошибок самого разного характера. Отладчик позволяет проследить ход (по шагам) выполнения программы и одновременно получить текущие значения всех переменных и объектов программы, что позволяет установить моменты времени (и операторы), в которые происходит ошибка и предпринять меры ее устранения. В целом, отладчик позволяет выполнить следующие действия:

- Перекомпилировать все и сделать новую сборку (**F7**);
- Запустить программу в режиме отладки без трассировки по шагам (**F5**);
- Выполнить программу по шагам (**F10**);
- Выполнить программу по шагам с обращениями к вложенным функциям(**F11**);
- Установить точку останова (**BreakPoint** – **F9**);
- Выполнить программу до первой точки останова (**F5**);
- Просмотреть любые данные в режиме отладки в специальном окне (**locals**);
- Просмотреть любые данные в режиме отладки при помощи мышки;
- Изменить любые данные в режиме отладки в специальном окне (**locals** и **Watch**);

- Установить просмотр переменных в специальном окне (**Watch**);
- Просмотреть последовательность и вложенность вызова функций.

При выполнении всех лабораторных курса студенты должны активно использовать отладчик VS, знать его возможности и отвечать на контрольные вопросы, связанные с отладкой и тестированием программ.

Примечание. Подробное описание материала и понятий вы можете найти в литературе [1 - 6] или справочной системе MS VS. Кроме того, не пропускайте лекции по курсу. Не рекомендую безоговорочно верить материалам из сети Интернет (например, в Википедии), так как там в некоторых статьях есть ошибки!

4.1.5. Модули

При проектировании и разработке программ применяется метод модульного программирования. Суть его заключается в том, что сложная программа разбивается на отдельные части (модули – не надо путать с учебными модулями). Каждый модуль разрабатывается отдельно и возможно разными программистами. Такой способ называют также декомпозицией. Совокупность модулей составляет проект программы. Модули бывают разных типов:

- Исходные модули, содержащие текст на языке программирования.
- Объектные модули получаются в результате компиляции программы.
- Исполнимые модули предназначены для выполнения программы.
- Исходные модули могут быть разных типов.

Основные исходные модули – это модули программ (*.c, *.cpp) и модули заголовочных файлов (*.h , *.hpp). Они включены в разные разделы дерева проекта программы.

Объектные модули формируются компилятором языка программирования (у нас C++) в том случае, если не было ошибок в программе. Объектные модули являются промежуточным звеном в создании программ, но не могут выполняться непосредственно. Подключаемые в программу библиотеки содержат объектные модули, поэтому не требуется их повторной компиляции. Объектные модули имеют расширение *.obj.

Исполнимые модули предназначены для непосредственного выполнения на компьютере или подключения в выполняемую программу. Основные исполнимые модули имеют расширение *.exe (или *.com), поэтому операционная система может контролировать их запуск. Модули динамических библиотек имеют расширение *.dll. Существуют и другие разновидности исполнимых модулей, зависящих от используемых технологий. Исполнимые модули формируются специальной программой системы программирования редактором связей (или компоновщиком). Редактирование связей заключается в проверке межмодульных связей по функциям и по данным и объединении их единый исполнимый модуль. Последовательный процесс обработки программы для получения исходного модуля представлен в методическом пособии в разделе 3[5].

Для объединения модулей функционально и по данным в C++ используются прототипы функций и описание внешних переменных. Покажем это на простом примере:

```
// Модуль 1
```

```
int i = 0; // Глобальная переменная
```



```

int Summ(int a, int b){}; // Описание функции в модуле 1
...
// Модуль 2
extern int i; // Описание внешних данных
int Summ(int , int ); // Прототип внешней функции
main(){
...
i = 5; // Использование внешних данных
S = Summ(2,2); // Вызов внешней функции
...
}

```

В модуле 2 используется переменная *i*, описанная как глобальная переменная в модуле 1. В модуле 2 используется функция **Summ**, описанная как в модуле 1. Для правильного объединения связей (работы редактора связей) используются описания внешних данных (**extern**) и задание прототипа функции **Summ**.

4.1.6. Библиотеки функций

В системах программирования предусматривается много библиотек для функций различного назначения (например, для работы со строками, выполнения ввода и вывода, работы с массивами и т.д.). Эти библиотеки подключаются с помощью заголовочных файлов или пространств описаний (имен - **namespace**). Кроме заголовочных файлов для использования библиотек подключаются специальные модули (иногда они подключаются автоматически), содержащие описания функций (*.lib или *.dll). Пример подключения библиотек ввода/вывода и библиотек для работы с математическими и системными функциями:

```

#include <math.h>
#include <process.h>
#include <stdio.h>

```

Стандартных библиотек очень много. Нужно хорошо знать их назначение и их состав для использования в программах. Чем лучше знания о библиотеках, тем быстрее и безошибочно можно создать сложную программу. В современных системах программирования доступны (большом количестве) библиотеки классов, которые описывают новые дополнительные типы данных.

4.1.7. Библиотеки функций и шаблонов классов: RTL, STL, MFC, ATL

В СИ++ введены специальные классы для работы с массивами. Для их использования необходимо освоить основы Объектно-ориентированного

программирования, что вам предстоит в дальнейших дисциплинах. Для общих сведений приводим здесь материал о назначении библиотек MS VS.

В системы программирования включаются различные библиотеки функций и классов. В MS VS включено несколько групп таких библиотек, которые постоянно развиваются и добавлялись по мере развития самой системы программирования. К сожалению, для разных языков и систем программирования пока нет единого стандарта, поэтому разрабатывать программные системы, используя библиотеки разных разработчиков, затруднительно. Однако существуют технологии и платформы, позволяющие решить эту проблему.

В MS VS предусмотрены следующие группы библиотек:

- C RTL - C Run-Time Libraries - стандартные библиотеки этапа выполнения.
- STL - Standard C++ Library – стандартные библиотеки C++.
- MFC - Microsoft Foundation Class Library – библиотеки классов MS.
- ATL - Active Template Library – библиотеки шаблонов.
- .NET Framework Class Library – библиотеки платформы .NET.

В C RTL включено много библиотек для работы со стандартными типами данных, файлами, строками и т.д. Эти библиотеки возникли относительно давно, но поддерживаются в настоящее время и представляют собой набор базовых средств программирования на языках C и C++.

В STL включено много библиотек, ориентированных на C++, и обеспечивающих работу с множеством классов, объектов и шаблонов. Эти библиотеки доступны всем пользователям и включают: библиотеки потокового ввода, вывода (cin, cout), работу с контейнерами объектов (vector, string, list, map, stack, set и многие другие.), работу с функциями из библиотек этапа выполнения (CRT).

В MFC включен широчайший набор классов и функций, позволяющих обеспечить программирование практически любых задач. Это библиотеки классов и функций: построения приложений, в том числе и в среде Windows, оконного интерфейса, управления файлами, работы с контейнерными классами, графического интерфейса, поддержки различных технологий, в том числе и современных (версии библиотек постоянно обновляются), программирования коммуникаций, обработки документов и многое другое. В частности, в других ЛР, мы познакомимся с контейнерами типа CArray, CList и др.

В ATL представлен набор библиотек, поддерживающих AX и COM технологий. Число доступных классов и шаблонов в этих библиотеках соизмеримы с набором библиотек MFC, они имеют дополнительные классы и классы, которые можно одновременно использовать совместно с MFC (“Shared by MFC and ATL”). В частности, в других ЛР, мы познакомимся с контейнерами типа CAtlArray, CAtlList и др.

.NET библиотеки представляют собой набор (иногда и более точно говорят платформу) библиотек для разработки программ – приложений самого универсального вида. В эту платформу включаются такие библиотеки, которые исключают необходимость использования устаревших библиотек (CTR и STL). Применение этой платформы позволяет сделать приложения более универсальными, охватывает большее число современных технологий и обеспечивает интеграцию с другими системами программирования, в том числе и на разных языках. В частности в этой платформе определены шаблоны классов Array, List и многие другие. .NET библиотеки широко используются для создания сайтов (ASP). Платформа .NET будет изучаться в отдельных дисциплинах.

Описания библиотек наиболее полно и с примерами в представлены [4]. Кроме того, описания библиотек CTR и STL вы найдете в [13]. Всю новую информацию об библиотеках можно также найти на сайтах MS. Хорошее знание существующих библиотек, их пополнения в новых версиях, несомненно, поможет вам стать профессионалом в программировании. В других ЛР цикла вы познакомитесь с составляющими других библиотек.

4.2. Примеры по теме ЛР № 3

Вторая часть задания, помимо первой связанной с изучением теоретического раздела, заключается в том, чтобы испытать в проекте СИ уже отлаженные программы и фрагменты программ. Нужно изучить технику этих программ и алгоритмы их работы. Это Вам понадобится для выполнения контрольных заданий. Возможно, что, осваивая теоретическую часть работы, вы уже на компьютере проверили выполнение фрагментов текста и применения массивов и указателей, тогда вам будет проще продемонстрировать их работу преподавателю. В дополнение к примерам, расположенным выше нужно испытать и изучить примеры расположенные ниже в этом разделе. Эти действия нужно обязательно сделать в отладчике.

Для выполнения этой части ЛР этого нужно создать пустой проект в MS VS (Test_LR3), как описано выше, скопировать через буфер обмена в него текст данных примеров, отладить его и выполнить.

4.2.1. Примеры описаний и инициализации массивов

Здесь нужно проверить все примеры описания и инициализации массивов, приведенные в теоретической части ЛР. Проверьте операторы динамического определения размерности массива.

4.2.2. Сумма элементов массива

Ниже приведена работающая программа, в которой вычисляется сумма элементов массива. Массив задан значениями инициализации. Число элементов фиксировано.

```
// Описание массивов
int Mas[5] = { 1,2,3,4,5}; // элементы с индексами 0 ... 4
// Простой цикл с массивом
int SumMas = 0;
for (int k = 0 ; k < 5 ; k++ )
    SumMas = SumMas + Mas[k] ;
printf("\nСумма массива SumMas = %d \n" , SumMas);
```

Попробуйте изменить значения в исходном массиве и выполнить вычисление заново.

4.2.3. Поиск максимума в массиве и его номера

Проверьте нижерасположенную программу поиска минимума в массиве.

```
// Дан массив iMas. Найти минимальный элемент и его порядковый номер.
int iMas[] = {1,22,3,-4,5,4,0};
int Max;
int Razm = sizeof(iMas)/sizeof(int);
// Распечатка массива
printf("\nЗадан массив iMas :\n");
for (int i =0 ; i < Razm; i++)
    printf( "iMas[%2d] = %d\n" , i , iMas[i]);
// Начальные условия цикла поиска максимума
int NumMax = 0;
Max = iMas[0];
// Поиск Max
for ( int i = 1; i < Razm; i++)
    if ( Max < iMas[i])
        { Max = iMas[i]; NumMax = i; };
// вывод результата
printf("\nВ массиве Max= %d его номер N = %d \n", Max ,NumMax);
```

4.2.4. Итерационный цикл расчета значения функции

Следующий пример – итерационные вычисления. Формула для вычисления значения функции – вычисление суммы членов ряда имеет вид:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots = \sum_{n=0}^{\infty} \frac{x^n}{n!}, x \in \mathbb{C}$$

Нетрудно составить циклическую программу для вычисления суммы этого ряда с точностью `eps`. Для математического обоснования алгоритма обратитесь к учебникам по математике.

```
////////// цикл итерационный для функции exp(y)
float sum = 1.0f; // Начальное значение для переменной суммирования
```

```

float x = 0.5f ;
float step = x; // Начальное значение для степени x
float fac = 1.0f; // Начальное значение для переменной текущего факториала
float Count = 1.0f;
float eps = 0.001; // точность, с которой будет рассчитана сумма ряда.

while ( step/fac > eps ) // проверка точности вычисления
{
    sum += step/fac; // суммирование членов ряда
    Count += 1.0f ; // n - порядковый номер члена ряда
    fac *= Count; // fac = fac * Count; - промежуточный факториал
    step = step*x; // в степени x
};
printf(" Ряд - exp = %f Функция из библиотеки = %f x = %f\n",sum , exp( x), (x ));

```

Измените вручную значение x (сейчас оно равно 0,5) прямо в программе и проведите повторные вычисления. В контрольном задании вы должны построить цикл для разных значений x и запомнить в массиве вычисленные значения функции. Красным цветом помечено выводимое значение для библиотечной функции, а синим значение степенного ряда, вычисленное в цикле..

Для более детального знакомства с алгоритмом приближенного вычисления значения алгебраической функции смотрите разделы математики – ряды Тейлора (конкретно ряды Маклорена) в wikipedia.org.

4.2.5. Указатели и динамические массивы

В тестовом проекте проверьте работу примеров, связанных с указателями и динамическими массивами, расположенных в теоретической части ЛР (см. выше). Проверку выполнения операций присваивания с помощью указателей необходимо произвести с помощью отладчика. В отладчике результаты необходимо продемонстрировать и преподавателю.

5. Лекция № 5 – Строки. Основные понятия

5.1. Основные понятия

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

5.1.1. Строки в СИ

Тип данных строка является одним из самых важных в информационных технологиях. В переменных и массивах данного типа запоминаются данные, характерные для использования в базах данных (названия, фамилии, имена, даты и т.д.). В принципе, любые данные могут храниться в символьном формате, а затем перед использованием преобразовываться в нужный формат с помощью специальных функций языка. К сожалению, этот тип отсутствует в списке стандартных типов языка СИ. В развитии СИ++ такой тип реализован на основе классов и библиотек. Несмотря на это в базовом языке СИ предусмотрены многочисленные возможности, библиотеки и средства для работы со строками. Строка в языке СИ представляется символьным массивом (**char**). Предусмотрены возможности: копирования и слияния строк, их ввода и вывода, выделение части строки, поиск в строке и многие другие возможности. Рассмотрим их.

5.1.2. Описание строк и инициализация строк

Описание строки совпадает с описанием символьного массива. При этом допустимы все изученные вами ранее возможности использования и инициализации массивов. Например:

```
char Str1[10]; // Описана строка максимальной длины 9 символов
char Str1[20]; // Описана строка максимальной длины 19 символов
Str1 [0] = 'П';
Str1 [1] = 'р';
Str1 [2] = 'и';
Str1 [3] = 'м';
Str1 [4] = 'е';
Str1 [5] = 'п';
Str1 [6] = '\0'; // Нулевой символ заносится в этом случае самостоятельно
```

Однако можно использовать конструкцию инициализации строки, использованную ниже. Отметим, что нулевой символ при такой инициализации заносится автоматически, нужно следить затем, чтобы длина строки инициализации не превышала размер символьного массива:

```
char Str2[20] = "Пример строки RTL!!";
```

```
char Str2[20] = {'П','р','и','м','е','р','!'}; // Можно и так для каждого эдемента – 1
СИМВОЛ
```

Когда исходный размер массива не так существен, то можно не указывать первоначальный максимальный размер в квадратных скобках. В этом случае размер массива будет вычислен автоматически с учетом нулевого символа. В этом случае можно записать так:

```
char Str3[] = "Пример строки с размером в длину строки инициализации!"; //Подсчитайте
сами (56)
```

Для заполнения строки можно использовать библиотечную функцию `strcpy`, которая входит в состав библиотеки RTL. Для этого нужно подключить заголовочный файл этой библиотеки:

```
#include <string.h>
...
char Str4[30];
strcpy(Str4, "Строка заносимая в программе!");
```

В данном случае нулевой символ в конце текста строки также будет занесен автоматически. Суммарная длина строки, вместе с нулевым символом, не должна превышать размер символьного массива. Для надежного копирования в библиотеки есть и другие функции: **`strcpy_s`** и **`strncpy`**.

5.1.3. Основные операции со строками

В принципе со строками можно работать как с массивом символов. Однако в этом случае программирование будет очень трудоемким (Например, копирование можно сделать в цикле!). Функции копирования мы уже рассмотрели (**`strcpy`**, **`strcpy_s`** и **`strncpy`**), однако в библиотеке строк (**`string.h`**), поддерживается много других возможностей. Очень важен контроль размерности строки при различных операциях со строками, так как, традиционно, эти ошибки приводят к затираниям в оперативной памяти программы, а, следовательно, к непредсказуемым результатам и новым ошибкам. Найти такие ошибки чрезвычайно трудно. Библиотечные функции включают контроль и позволяют избежать ошибок затирания. Например, функция копирования строк может быть выполнена так:

```
#include <string.h>
...
char Str5[30];
strncpy(Str5, Str4 , 29); // Копироваться более 29 символов не будет!
```

Предусмотрена возможность динамического определения фактической длины строки. Это распространенная функция **`strlen`**. Для определения размера массива используется другая возможность (`sizeof`), но число элементов в этом случае определяется с учетом типа массива. Примеры:

```
char Fam[14] ;
```

```
strcpy_s(Fam , "Петров");
printf ( "Строка Fam = %s имеет длину - %d \n", Fam , strlen (Fam));
strcpy_s(Fam , "Спиридонов");
printf ( "Строка Fam = %s имеет длину - %d \n", Fam , strlen (Fam));
printf ( "Максимальный размер Fam = %s равен - %d \n", Fam , sizeof (Fam)/sizeof
(char) );
```

Результат выполнения данного фрагмента программы следующий:

```
Строка Fam = Петров имеет длину - 6
Строка Fam = Спиридонов имеет длину - 10
Максимальный размер Fam = Спиридонов равен - 14
```

Последний оператор печати вычисляет максимальный размер конкретного символьного массива (**Fam**) с учетом нулевого символа (максимально возможная длина строки равна 13). Данные функции **strlen** и макросы (!) **sizeof (char)** можно использовать и в функциях копирования строк (см. выше) и в функциях слияния строк (**strcat** и **strncat**). Функция **strcat** выполняет слияние двух строк. Рассмотрим пример.

```
char Person[30];
char Name1[20] = "Сергей";
char Fam1[20] = "Большаков";
// Копирование и слияние без контроля
strcpy(Person,Fam1);
strcat(Person, " "); // Нужно для пробела между фамилией и именем
strcat(Person, Name1);
printf ("Фамилия и имя = %s !\n", Person);
```

Результат получим такой:

```
Большаков Сергей !
```

Если предусмотреть контроль копирования и слияния строк, то программа будет выглядеть так (используем функцию **strlen** и макрос **sizeof (char)**):

```
strncpy(Person,Fam1, sizeof (Person)/sizeof (char) - 1);
// Следующее нужно для пробела между фамилией и именем
strncat(Person, " ", sizeof (Person)/sizeof (char) - strlen (Person) - 1);
// Допустимо только - sizeof (Person)/sizeof (char) - strlen (Person) - 1
strncat(Person, Name1 , sizeof (Person)/sizeof (char) - strlen (Person) - 1);
printf ("Фамилия и имя = %s !\n", Person);
```

Результат будет аналогичным, но в данном тексте исключается затирание оперативной памяти в программе, а, следовательно, и ошибки с ним связанные.

5.1.4. Ввод и вывод строк

Для ввода вывода в языке СИ используются специальные библиотеки. Особенности ввода вывода будет посвящена отдельная лабораторная работа. Ввод данных в стандартном СИ предполагается с клавиатуры, а вывод на экран дисплея (консольный ввод/вывод). Возможен ввод/вывод с файловой системой, но мы будем этими технологиями заниматься в отдельной работе. Здесь же мы отметим, что предусмотрено три возможности выполнять ввод/вывод:

- Ввод/вывод верхнего уровня (Потоки, форматирование и буферизация);
- Ввод/вывод среднего уровня (Буферизация, минимальное форматирование);
- Ввод/вывод нижнего уровня (Буферизация и работа с байтами).

К функциям верхнего уровня относятся известные вам функции **printf** и **scanf**, которые для операций ввода вывода используют параметр форматирования “%s”. Кроме этого, для форматирования может указываться максимальная длина ввода вывода “%.10s” (10 символов), минимальная длина ввода вывода “%5s” (5 символов), совместное ограничение “%5.10s” и выравнивание влево “%-5.10s”. Примеры:

```
char Str10[ 20 ] = {"Пример"}; // строки!
printf("Строка -> '%s' \n", Str10);
printf("Строка -> '%10s' \n", Str10);
printf("Строка -> '%-10s' \n", Str10);
printf("Строка -> '%3.20s' \n", Str10);
printf("Строка -> '%-3.4s' \n", Str10);
```

Результат работы операторов вывода на экран будет следующий:

```
Строка -> 'Пример'
Строка -> '      Пример'
Строка -> 'Пример      '
Строка -> 'Пример'
Строка -> 'Прим'
```

Для функции **scanf** также указывается формат вида - “%s” (без ограничения длины вводимой строки) и с ограничением в заданное число символов - “%5s”.

```
scanf ("%s" , Str10);
printf("Строка (scanf) -> '%s' \n", Str10);
scanf ("%5s" , Str10);
printf("Строка (scanf) -> '%s' \n", Str10);
```

Получим результат:

```
1234567890
Строка (scanf) -> '1234567890'
1234567890
Строка (scanf) -> '12345'
```

При другом вводе (с пробелом!) результат будет другим, так как пробел ограничивает размер вводимой строки:

```
12345 678909
Строка (scanf) -> '12345'
Строка (scanf) -> '67890'
```

Для ввода с пробелами и другими особенностями для строк используют функции **gets** и **puts** и другие. Вам предлагается здесь познакомиться с этими функциями самостоятельно. Мы рассмотрим данные функции позже.

5.1.5. Манипуляция со строками и в строке

Для сравнения строк в СИ используются библиотечные функции: **strcmp**, **strncmp**, **_strnicmp** и другие. Пример и результаты приведены ниже.

```
// strcmp , strncmp - сравнение строк
char Name[14] = "Василий"; // Посмотреть в отладчике нуль-терм.
setlocale( LC_ALL, "" ); // нужно подключить locale.h
//
if ( strcmp(Name , "Василий") == 0) printf ("Строки равны (идентичны)! \n" );
if ( strcmp(Name , "Алексей") > 0) printf ("Строки не равны (1-я > 2- й)! \n" );
if ( strcmp(Name , "Федор") < 0) printf ("Строки не равны (1-я < 2- й)! \n" );
// число сравниваемых данных - 5
if ( strncmp(Name , "Василиса" , 5 ) == 0) printf ("Строки равны (идентичны)! \n" );
// нет ограничения
if ( strcmp(Name , "Василиса" ) == 0)
    printf ("Строки равны (идентичны)! \n" );
else
    printf ("Строки не равны ! \n" );
// _strnicmp – сравнение без учета регистра
if ( _strnicmp("Name" , "NAME" , 3 ) == 0) printf ("Строки равны (_strnicmp - идентичны)! \n" );
```

Результат получим такой:

```
Строки равны (идентичны) !
Строки не равны (1-я > 2- й) !
Строки не равны (1-я < 2- й) !
Строки равны (идентичны) !
Строки не равны !
Строки равны (_strnicmp - идентичны) !
```

Для поиска первого и последнего вхождения символа в строке используются функции: **strchr** и **strrchr**. Пример их применения дан ниже.

```
// strchr - первое вхождение "и" в слове Василий
printf ("Строка исходная = %s    Часть с найденным \"и\" = %s \n" , Name ,
strchr(Name , 'и') );
// strrchr - последнее вхождение "и" в слове Василий
printf ("Строка исходная = %s    Часть с найденным \"и\" = %s \n" , Name ,
strrchr(Name , 'и') );
```

Результат получим такой:

```
Строка исходная = Василий    Часть с найденным "и" = илий
Строка исходная = Василий    Часть с найденным "и" = ий
```

Контроль строки на содержания подмножества символов производится функцией **strspn**.

```
// strspn
char string1[] = "cabbage";
int result;
result = strspn( string1, "abc" );
printf( "Размер подстроки '%s', в которой только символы: a, b, or c "
      "= %d байт\n", string1, result );
```

Результат получим такой:

Размер подстроки 'cabbage', в которой только символы: a, b, or c = 5 байт

5.1.6. Преобразование к нижнему или верхнему регистру

Для преобразования строки к нижнему и верхнему регистру (строчные и прописные буквы) применяют функции **strlwr** и **strupr**, соответственно.

// ПРЕОБРАЗОВАНИЕ К НИЖНЕМУ И ВЕРХНЕМУ РЕГИСТРУ

```
char string5[10] = ;
strcpy( string5 , "Sample");
printf( " %s\n", _strupr ( string5 ) );
setlocale( LC_ALL, "" );
strcpy( string5 , "Пример!"); // русские не преобразует без setlocale( LC_ALL, "" )
printf( " %s\n", strupr ( string5 ) );
printf( " %s\n", strlwr ( string5 ) );
```

Результат получим такой:

SAMPLE

ПРИМЕР!

пример!

5.1.7. Дублирование динамических строк

Для дублирования строки с выделением памяти применяют функцию **strdup** (или **_strdup**). После ее использования ее освобождают функцией **free**. Отметим что эта операция отличается от операции копирования указателей.

```
char buffer[] = "Это текст буферной строки!";
char *newstring;
printf( "Исходная строка 1: %s\n", buffer );
newstring = strdup( buffer ); // Дублирование строки динамически выделяется память
newstring[2] = '*'; // Изменение 3-го символа в строке
printf( "Копия строки: %s\n", newstring );
printf( "Исходная строка 2: %s\n", buffer );
free( newstring ); // Изменение освобождение памяти под дубль строку
```

Результат получим такой:

Исходная строка 1: Это текст буферной строки!

Копия строки: Эт* текст буферной строки!

Исходная строка 2: Это текст буферной строки!!

5.1.8. Выделение подстрок на множестве разделителей

Удобной возможностью разбиения строки на части является применения функции **strtok**. На основе заданного множества разделителей (у нас в примере: пробел, запятая, табуляция и конец строки) последовательно в цикле выделяются подстроки. Такие действия, соответствуют грамматическому разбору строки и часто называются парзингом (parse). В примере подстроки выводятся на экран.

/// ВЫДЕЛЕНИЕ ПОДСТРОК НА МНОЖЕСТВЕ РАЗДЕЛИТЕЛЕЙ

```
char strText4[] = "Строка\tdля ,,разделения на tokens\n по множеству
разделителей\n";
char seps[] = " ,\t\n"; // Допустимые разделители
char *token;

printf( "Исходная строка:%s\n", strText4 ); // Разделители не видны
printf( "Подстроки (tokens):\n" );
// Получение первой подстроки:
token = strtok(strText4, seps ); //
// можно взять и функцию strtok_s
while( token != NULL ) // проверка наличия новых подстрок
{
    // Вывод
    printf( " %s\n", token );
    // Новая подстрока :
    token = strtok( NULL , seps ); //
}
//
```

Результат получим такой:

Исходная строка:Строка для ,,разделения на tokens
по множеству разделителей

Подстроки (tokens):

Строка

для

разделения

на

tokens

по

множеству

разделителей

5.2. Преобразование данных в строку и обратно

Очень важные действия в языках программирования связаны с взаимными преобразованиями данных. Например, число преобразуется в строку или наоборот строка с цифровым текстом преобразуется в число. Для таких преобразований в СИ есть функции, которые включены в библиотеку **stdlib.h**. Ниже показаны примеры такого использования.

// Преобразование из строки в число и чисел в строку

```
char Buf[15];
int Dec , Sign;
//
printf("Из Строки (printf) -> '%s' в целое - %d\n", "125" , atoi( "125" ) );
printf("Из Строки (printf) -> '%s' в вещественное - %8.3f или %e \n", "125.5" , atof(
"125.5" ) );
printf("Из целого (printf) -> '%d' в строку - %s в строку(itoa_s) - %s \n", 125 , itoa(
125, Buf , 10 ), _itoa_s( 25, Buf , 14 ,10 ) );
printf("Из целого с защитой Buf (printf) -> '%s' \n", Buf );
_gcvrt( -35.5, 12, Buf );
printf("Из вещественного Buf (_gcvrt) -> '%s' \n", Buf );
//
```

Результат получим такой:

```
Из Строки (printf) -> '125' в целое - 125
Из Строки (printf) -> '125.5' в вещественное - 125.500 или 1.299551e-
257
Из целого (printf) -> '125' в строку - 125 в строку(itoa_s) - (null)
Из целого с защитой Buf (printf) -> '125'
Из вещественного Buf (_gcvrt) -> '-35.5'
```

5.3. Сортировка строк

Сортировка массива строк массива строк по убыванию выполняется аналогично сортировке целого массива, только отличается фрагмент сравнения и обмена элементов в массиве строк. В примере предполагается, что все строки занимают размер не более 9-ти символов. Для обмена используется специальная функция **SwapStr**, которая предварительно описывается.

// Функция обмена строк вне main

```
void SwapStr (char * S1 , char * S2 )
```

```

{
    char TempStr[20];
    strcpy(TempStr , S1 );
    strcpy(S1 , S2);
    strcpy( S2 , TempStr);
};

//
#define RazmMas 5
...
// Массив строк инициализируется в программе фамилиями
char StrMas[RazmMas][10]={"Сидоров", "Алетров", "Иванов", "Жучков" , "Акулов"};
// печать массива строк до сортировки
printf ("До сортировки \n");
    for (int i =0 ; i < RazmMas ; i++ )
        printf (" %d. - %s \n" , i + 1 , &StrMas[i][0]);

// Сортировка
for (int k = 0 ; k < RazmMas - 1 ; k++)
    for ( int i =0 ; i < RazmMas - 1; i++ )
    {
        if ( strcmp(&StrMas[i][0] , &StrMas[i +1][0]) > 0 ) // Для убывания по алфавиту
// Обмен если условие сортировки не соблюдается
            SwapStr(&StrMas[i][0] , &StrMas[i +1][0]);
    };

// печать массива строк после сортировки
printf ("После сортировки \n");
    for (int i =0 ; i < RazmMas ; i++ )
        printf (" %d. - %s \n" , i + 1 , &StrMas[i][0]);

```

Результат получим такой:

До сортировки

- 1. - Сидоров**
- 2. - Алетров**
- 3. - Иванов**
- 4. - Жучков**
- 5. - Акулов**

После сортировки

- 1. - Акулов**
- 2. - Алетров**
- 3. - Жучков**
- 4. - Иванов**
- 5. - Сидоров**

5.4. Динамические строки

Часто все переменные и массивы располагаются в оперативной памяти заранее, до начала выполнения программы. Все рассмотренные ранее примеры включали такие переменные и массивы. Размер выделенной памяти под массив/строку, то есть его размерность в этом случае изменить в программе стандартным способом нельзя. Поэтому приходится заранее рассчитывать максимально возможную размерность массива, которая приемлема для всех случаев. Это приводит к перерасходу памяти для отдельных случаев. Для экономии памяти и для построения более универсальных программ используют динамически выделяемую память под массивы, строки и переменные. Такие данные называются динамическими. Важность динамического выделения памяти для строк трудно переоценить. Так как, например, для записи в новую строку текста превосходящего объема, нужно выделить новый больший объем ОП. Это выделение выполняется через предварительное освобождение старого фрагмента динамической памяти.

Для строк имя символьного массива рассматривается системой как указатель на **char (char *)**. Это позволяет его использовать отдельно в функциях и операциях как специальный объект (строка).

В языке СИ оперативная память выделялась с помощью специальных функций (alloc, calloc, malloc, free и т.д.). Они включены в библиотеку СИ – malloc.h. С помощью специальных операций в zpsrt C++ (**new** и **delete**) аналогично можно выделять память во время выполнения программы. Динамические переменные располагаются в оперативной памяти в специальной области. Для работы с такими данными используют указатели и ссылки. Примеры выделения динамической памяти для указателей и массивов строк даны ниже.:

```
// Динамическая память
char * pStr = (char *) malloc( 10);
strcpy( pStr , "Динамика!" );
printf ("Динамическая память - %s \n" , pStr);
// Освобождение памяти
free (pStr);
pStr = (char *) calloc (15 , sizeof(char));
strcpy( pStr , "Новая память!" );
printf ("Новая память - %s \n" , pStr);
free (pStr);
//
int Razm;

////////////////////
// Динамический массив строк
////////////////////
printf ("Введите размер массива строк: \n" );
```

```

scanf ("%d", &Razm);
char * pStrMas =(char *) calloc (Razm , sizeof(char) * 20); // 20 одна строка
printf ("\nВведите строки массива [%d]: \n" , Razm );
    for (int i =0 ; i < Razm ; i++ )
        // scanf ("%s", pStrMas + i * 20);
        scanf ("%s", &pStrMas[ i * 20]);
// Распечатка
    for (int i =0 ; i < Razm ; i++ )
        //printf ("Строки массива %d - %s\n", i , pStrMas + i * 20);
        printf ("Строки массива %d - %s\n", i , &pStrMas[ i * 20] );

//////////
// Слияние массива в отдельную строку
    int nMasSize = 200;
char * pBigString = (char *) malloc( nMasSize + 2 + Razm ); // Пробелы воск. знак
и ноль

    pBigString[0] = '\0';
    for (int i =0 ; i < Razm ; i++ )
        { strcat( pBigString , &pStrMas[ i * 20]);
          strcat( pBigString , " ");
        };
    strcat( pBigString , "!");
    printf ("Большая строка - %s \n" , pBigString);
    free (pBigString);

```

Результат получим такой:

Введите размер массива строк:

4

Введите строки массива [4]:

Студенты

могут

хорошо

учиться

Строки массива 0 - Студенты

Строки массива 1 - могут

Строки массива 2 - хорошо

Строки массива 3 - учиться

Большая строка - Студенты могут хорошо учиться !

5.5. Библиотеки функций и классы для строк

В системах программирования предусматривается много библиотек для функций различного назначения (например, для работы со строками, выполнения ввода и вывода, работы с массивами и т.д.). Эти библиотеки подключаются с помощью заголовочных файлов или пространств описаний (пространств имен в C++ - **namespace**). Кроме заголовочных файлов для использования библиотек подключаются специальные модули (иногда они подключаются автоматически), содержащие описания функций (*.lib или *.dll). Пример подключения библиотек ввода/вывода и библиотек для работы с математическими и системными функциями:

```
#include <math.h>    // Математическая библиотека функций
#include <process.h>  // Системная библиотека функций
#include <string.h>    // Библиотека функций для работы со строками
#include <stdlib.h>    // Стандартная библиотека разных функций
#include <locale.h>    // библиотека локализации программ
```

Стандартных библиотек и классов для описания строк очень много. Нужно хорошо знать их назначение и их состав для использования в программах. Чем лучше знания о библиотеках, тем быстрее и безошибочно можно создать сложную программу. В современных системах программирования доступны (большом количестве) библиотеки классов, которые описывают новые дополнительные типы данных.

5.6. Примеры программы с использованием строк

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 ветвления (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 нужно сделать в отладчике.

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.

5.6.1. 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 части, там представлены следующие примеры:

1. Описания строк и их инициализации, заполнения строки вручную,

2. Копирования и слияния строк, ввода и вывода строк,
3. Сравнения строк, контроль символов в строке,
4. Преобразование к нижнему или верхнему регистру,
5. Дублирования строк,
6. Выделения составляющих строки по разделителям,
7. Преобразование данных в строку и обратно,
8. Сортировки строкового массива,
9. Использования динамических строк.

Кроме этого ниже представлены примеры, которые могут быть полезными, в том числе и при выполнении контрольных заданий. Их тоже нужно изучить и проверить.

5.6.2. Пример на выделение подстроки

Пример выделения подстроки для строки strText5, начиная с символа с номером **begChar**. Число символов которые будут взяты равно – **sizeSubstr**. Эти значения задаются в программе. Для подстроки выделяется динамическая память, размером, на всякий случай, превышающая размер исходной строки на число символов в подстроке. Полученное значение распечатывается.

```
//Выделение подстроки
int begChar =4 ;
int sizeSubstr = 7;
char strText5[] = "10123456789 123456789_123456789";
char * pSubstr = (char *) malloc ( sizeof (strText5) + sizeSubstr +1);
strncpy( pSubstr , strText5 + begChar, sizeSubstr + 1 );
pSubstr[5] = '\0'; // Сознательное укорачивание подстроки до 5-ти символов
printf( "Строка: %s Подстрока = %s с =%d число=%d\n",strText5, pSubstr
,begChar,sizeSubstr );
```

Результат получим такой:

```
Строка: 10123456789 123456789_123456789 Подстрока = 34567 с =4 число=7
```

5.6.3. Обмен строк одинакового размера

Для обмена строк одинакового размера можно использовать следующий фрагмент. Максимальные размеры данных строк должны быть идентичны (или размеры символьных массивов или размеры выделенной динамической памяти).

```
char TempStr[40];
char S1[40] = "Первая строка";
char S2[40] = "Вторая строка";
printf( "Перед обменом: %s - %s\n", S1 , S2 );
strcpy(TempStr , S1 );
strcpy(S1 , S2);
```

```
strcpy( S2 , TempStr);
printf( "После обмена: %s - %s\n", S1 , S2 );
```

Результат получим такой:

Перед обменом: Первая строка - Вторая строка

После обмена: Вторая строка - Первая строка

5.6.4. Поиск символа в строке

Поиск символа, содержащегося в строке на заданном множестве ("**abc**") символов и определение его номера. Если символов нет, то номер – конец строки, если строка пуста, то нуль.

```
// ...
void test( const char * str, const char * strCharSet )
{
    int pos = strcspn( str, strCharSet );
    printf( "strcspn( \"%s\", \"%s\" ) = %d\n", str, strCharSet, pos );
};
// ...
char string7[] = "cabbage";
// ...
result = strspn( string7, "abc" );
printf( "The portion of '%s' containing only a, b, or c "
        "is %d bytes long у строки - %d\n", string7, result , strlen(string7) );
// strcspn
test( "xyzbxz", "abc" );
test( "xyzbxz", "xyz" );
test( "xyzbxz", "no match" );
test( "xyzbxz", "" );
test( "", "abc" );
test( "", "" );
```

Результат получим такой:

```
strcspn( "xyzbxz", "abc" ) = 3
strcspn( "xyzbxz", "xyz" ) = 0
strcspn( "xyzbxz", "no match" ) = 6
strcspn( "xyzbxz", "" ) = 6
strcspn( "", "abc" ) = 0
strcspn( "", "" ) = 0
```

5.6.5. Замена всех символов в строке

Для заполнения все строки одним символом используется функция `strset`

```
char string7[] = "cabbage";

printf( "Before: %s\n", string7 );
_strset( string7, '*' ); //
//
printf( "After: %s\n", string7 );
```

Результат получим такой:

```
Before: cabbage
After:  *****
```

5.6.6. Поиск вхождения подстроки в строке

Для поиска в строке (**string**) вхождения подстроки (**str**) используется функция **strstr**, пример ее использования и результаты показаны ниже. В строке мы ищем подстроку “вход”.

```
char str[] = "вход";
char string[] = "Счастливая собака остановилась у входа и залаяла";
char fmt1[] = "    1    2    3    4    5"; // Для разметки полей
char fmt2[] = "12345678901234567890123456789012345678901234567890" ; // Для
разметки
char *pdest;
int result1;

printf( "Строка, в которой ищем:\n%s\nПодстроку: %s \n", string, str );
printf( "%s\n%s\n\n", fmt1, fmt2 );
pdest = strstr( string, str );
result1 = (int)(pdest - string + 1);
if ( pdest != NULL )
    printf( "%s Подстрока найдена в позиции %d\n", str, result1 );
else
    printf( "%s Не найдена подстрока!\n", str );
```

Результат получим такой:

```
Строка, в которой ищем:
Счастливая собака остановилась у входа и залаяла
Подстроку: вход
      1      2      3      4      5
12345678901234567890123456789012345678901234567890
вход Подстрока найдена в позиции 34
```

5.6.7. Формирование действительного числа с точкой и знаком

Использование различных функций и приемов для работы со строками продемонстрируем на примере использования функции `_fcvt`, после использования которой полученную строку нужно преобразовать: поставить точку в нужное место и добавить знак.

```
printf("Из вещественного (printf) -> '%f' в строку - %s \n", 125.5 , _fcvt( 125.5, 4 , &Dec ,
&Sign ) );
printf(" Параметры(1): где точка - %d и знак %d \n", Dec , Sign);
printf("Из вещественного (printf) -> '%f' в строку - %s \n", -25.5 , _fcvt( -25.5, 4 , &Dec ,
&Sign ) );
printf(" Параметры(2): где точка - %d и знак %d \n", Dec , Sign);
// Составим число
char RealBuf[15];
strcpy_s (Buf , 14 , _fcvt( -25.5, 4 , &Dec , &Sign ) );
printf("Из вещественного с защитой Buf (printf) -> '%s' \n", Buf );
if ( Sign == 1 )
strcpy(RealBuf , "-" );
strncat(RealBuf ,Buf , Dec );
RealBuf[Dec + 1 ] = '\0';
printf("Из вещественного с защитой RealBuf (printf) -> '%s' \n", RealBuf );
strcat (RealBuf , ".");
printf("Из вещественного с защитой RealBuf (printf) -> '%s' \n", RealBuf );
strncat (RealBuf , Buf + Dec , sizeof (Buf + Dec) );
printf("Из вещественного с защитой RealBuf (printf) -> '%s' \n", RealBuf );
```

Результат получим такой:

```
Из вещественного (printf) -> '125.500000' в строку - 1255000
Параметры(1): где точка - 3 и знак 0
Из вещественного (printf) -> '-25.500000' в строку - 255000
Параметры(2): где точка - 2 и знак 1
Из вещественного с защитой Buf (printf) -> '255000'
Из вещественного с защитой RealBuf (printf) -> '-25'
Из вещественного с защитой RealBuf (printf) -> '-25.'
Из вещественного с защитой RealBuf (printf) -> '-25.5000'
```

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

6. Лекция № 6 – Функции. Основные понятия

6.1. Основные понятия

В теоретической части методических указаний лабораторной работы вводятся основные понятия и рассматриваются принципы для работы с функциями на языке программирования СИ.

6.1.1. Функции – основа процедурного программирования.

Возможно, термин функция введенный в язык программирования СИ несколько сойдет с толку отдельным студентам, которые успешно изучали и изучают математические дисциплины в школе и университете. В программировании под функцией обычно понимается другое – это отдельно записанный фрагмент текста программы, который можно вызывать многократно в разных местах программы, задавая при этих вызовах разные параметры. Кроме того, для каждой функции, в зависимости от ее назначения мы можем присвоить ей уникальное имя (или название), соответствующее ее фактическому назначению. Например: **PrintArray** (печать массива) или **SortArray** (сортировка массива), что во-первых легче запоминается, а во-вторых, делает программу более наглядной и легко читаемой (обозримой). Последние примеры соответствуют понятию абстрагирования (абстракции действий, операторов - функций или процедур). Это позволяет, в свою очередь, абстрагироваться (отвлекаться – не учитывать) от деталей внутреннего устройства самой функции. Приемы абстрагирования экономят расходы человеческой памяти.

Абстракция функций или процедур, в свою очередь, является основой концепции программирования, которая называется также концепцией процедурно – ориентированного программирования (ПОП). ПОП рассматривает программу в виде взаимосвязанной совокупности функций – процедур. Исторически сложилось так, что ПОП была ранее разработана и в некоторой степени повторяла работу вычислителя (вычислительной машины) – компьютера, основанного на модели Машины Тьюринга (логическая модель) и построенного на базе модели фон Неймана (техническая модель)[5]. Отметим также, что процедурную концепцию иногда называют структурным подходом к программированию, что, по сути, не совсем верно. В данной лабораторной работе мы рассмотрим вопросы, связанные с: описанием и использованием функций, передачей в них параметров, вызовом функций и многие другие аспекты, непосредственно связанные с их использованием.

В концепции процедурно ориентированного программирования главной конструкцией (модулем, строительным блоком) является функция/процедура. В ней

разрабатываются и изучаются способы построения процедур, разделение (декомпозиция) сложной задачи на процедуры, способы связи процедур и передачи параметров между ними, совместного функционирования процедур, отладки программных систем и многое другое. Некоторые элементы концепции ПОП мы рассмотрим в этой ЛР.

6.1.2. Понятие функции

Программа на языке программирования может быть представлена упорядоченной совокупностью последовательно расположенных операторов (S_i):

$$< S_1, S_2, S_3, S_4, \dots, S_i, S_{i+1}, S_{i+2} \dots S_{k-2}, S_{k-1}, S_k >$$

В предыдущих лабораторных работах мы рассмотрели вопросы циклической и разветвляющейся организации выполнения программы. Для этого используются специальные операторы цикла и ветвления. Однако их применение не дает хорошей и наглядной возможности решить проблему повторяющихся последовательностей операторов, особенно, расположенных в разных частях последовательности операторов в программе. Ниже такие возможные повторы выделены красным шрифтом (S_{i+1}, S_{i+2}):

$$< S_1, S_2, S_3, S_4, \dots, S_i, S_{i+1}, S_{i+2} \dots S_{k-2}, S_{k-1}, S_k >$$

Эти группы могут содержать много операторов, и, кроме того, выполняться в разных текущих условиях с разными переменными программы и с разными исходными данными. Возникает естественное желание (кстати, как в математике используется новое обозначение формулы) дать им отдельное имя, однократно разместить их в отдельном месте программы ($S_{\phi 1} = S_{i+1}, S_{i+2}$) и выполнять их по мере необходимости с разными параметрами. Если иметь специальный оператор вызова группы этих операторов, с возможностью передачи им своих параметров ($S_{\text{вф}1}$ - вызов функции 1 - $S_{\phi 1}$), то тогда нашу программу можно представить следующим образом:

$$< S_1, S_{\text{вф}1}, S_4, \dots, S_i, S_{\text{вф}1} \dots S_{\text{вф}1}, S_k, \dots > < S_{\phi 1} \dots >.$$

Если представить сказанное выше в виде текста на языке СИ, то это может выглядеть таким образом:

```
...
    int a = 5 , b =5 , c;
...
    c = Summa (a , b); // c = 10
...
    int x = 2 , y =3 ;
...
    c = Summa (x , y); // c = 5
...
```

Здесь мы придумали простую функцию **Summa**, в которой производятся вычисления, причем используются разные параметры для вычислений (a и b) и (x и y). Описание такой функции рассмотрим ниже.

Размер программы, если число операторов в функции велико, при этом значительно сокращается, она становится более наглядной, но самое главное отлаживать теперь необходимо не все идентичные группы операторов (S_{i+1} , S_{i+2}), а только одну группу ($S_{\Phi 1}$), которая называется в языке СИ **функцией** (или процедурой). В разных языках существуют и другие названия таких групп операторов: процедуры, подпрограммы, subroutine, методы и т.д. Названия сути не меняет.

Разбиение программ на функции при программировании дает следующие преимущества, которые, отметим, появились на стадиях эволюционного развития языков программирования:

- Функцию можно вызывать из разных мест программы, что позволяет избежать повторного программирования.
- Одну и ту же функцию можно использовать в разных программах (библиотеки функций, как стандартных, так и собственных).
- Использование функций повышают уровень структурированности программы и облегчают её понимание и проектирование.
- Локализация имен переменных и параметров внутри функции, что позволяет значительно сократить их разнообразие.
- Использование функций облегчает чтение программы и значительно ускоряет поиск ошибок и их исправление.

Следствием процедурного подхода является начало выполнения одной главной программы – функции с названием **main**. Эта функция является главной и всегда определяет “точку входа” в программу – первый выполняемый оператор (S_1). Функция **main** является особой, она также имеет тип, и формальные параметры, но об этом речь пойдет ниже.

6.1.3. Понятия, связанные с функциями в программировании

С функциями в языках программирования связаны следующие важные понятия:

- Определение функции или описание функции (синонимы)
- Прототип функции
- Тип возврата функции и способы возврата данных из функции
- Формальные параметры функции
- Вызов функции
- Фактические параметры функции
- Тело функции

Эти понятия мы более подробно рассмотрим ниже, а здесь начнем с простейшего работающего примера с использованием функции. Пусть в отдельном модуле проекта (**second.cpp**, в проекте есть еще один исходный модуль - **first.cpp**), и в нем дано описание функции суммирования двух переменных - **Summa**:

```
// Описание-определение функции (модуль second.cpp)
int Summa (int a , int b) // формальные параметры функции a и b
{
    // тело функции – всего один оператор return
    return (a + b); // возвращаемое значение функции типа int
```



```
};
```

Оператор **return** в функции (оператор возврата значения) задает возвращаемое функцией значение. В главной программе **main** выполнено (в исходном модуле - **first.cpp**) обращение (вызов) к этой функции. Оно показано непосредственно в качестве параметра функции **printf** при печати результата:

```
// Основная программа (first.cpp) с вызовом функции
#include <stdio.h>
#include <process.h>
// Прототип функции
int Summa (int a , int b);
// Главная функция main
int main()
{
    // Руссификация проекта
    system ( " chcp 1251 > nul " );
    // Вызов функции в параметрах printf
    printf ( "Сумма = %d \n" , Summa(12,13)); // фактические параметры константы
    //
    system(" PAUSE");
    //
}
```

Результат работы такой программы:

Сумма = 25

Для продолжения нажмите любую клавишу . . .

6.1.4. Описания и определения функций

Термины описание функции и определение функции являются синонимами. Мы будем использовать оба этих термина. Описание функции дается однократно в каждой программе и должно быть доступно (уже известно – расположено выше по тексту) перед вызовом этой функции. При описании функции задаются:

- Название функции, уникальное имя в пределах всей программы.
- Тип возврата функции.
- Список формальных параметров функции с указанием их типов
- Тело функции – составной оператор: описания и операторы функции, заключенные в операторные скобки ({}).

Название функции это уникальное имя в всей программе, которое однозначно определяет действия, выполняемые данной функцией. Могут быть хорошие и плохие названия (для запоминания, понимания и т.д.). Например, хорошие названия:

PrintArray, FindKey, PoiskMaximum и др.

Плохие названия функций имеют вид:

A13, Fun1, Proc2 и т.д.

Стандартом хорошего названия являются правила так называемой “Венгерской нотации”, основной смысл которых является запись названия таким образом, что раскрывается смысл переменной или функции. Например, название функции: **Sort_And_Print_Array**. Более подробно об этих правилах смотрите в литературе [10]. Формальное описание функции выглядит так:

<Описание функции> := [<Спецификация типа возврата функции>] <Название функции> ([<Список Формальных параметров>]) {<тело функции>;}

Спецификация типа возврата функции – это любой допустимый тип в программе: стандартный (**int**, **float** и т.д.), системный (системные структуры и **typedef** переменные) и пользовательский (пользовательские структуры в СИ и классы в C++). Допускаются типы с указателями и ссылками. В отдельных случаях можно отказаться от задания типа возврата, тогда используется спецификатор – **void**. В этом случае функцию нельзя использовать в выражениях, операторах присваивания и фактических параметров при вызове функций . Если спецификатор возврата отсутствует, то подразумевается по умолчанию возврат типа **int**.

Тело функции – это любой составной оператор, заключенный в фигурные скобки. Завершение выполнения функции выполняется двумя вариантами:

- При достижении последней в теле функции закрывающей фигурной скобки (“}”).
- Выполнении в программе специального оператора **return**.

Оператор **return** может быть задан в двух видах:

```
return; // Когда тип возврата void
return < Выражение, тип которого совпадает с типом возврата функции>;
```

Список формальных параметров функции – это перечень описаний переменных со специальными именами, которые используются только в этой функции. Разделителем между описаниями является запятая. Имена формальных параметров должны быть уникальными в теле самой функции и хорошо подобраны по смыслу, хотя они могут совпадать с именами главной программы и других функций. Число формальных параметров не ограничивается, но не должно быть большим, для наглядности. Пример описания функции с формальными параметрами (параметры выделены красным цветом):

```
int MaxMas ( int * iMas , int Razm, int * Max)
{
    int TempMax;
    TempMax = iMas[0];
    for ( int i = 1; i < Razm; i++)
        if ( TempMax < iMas[i])
            TempMax = iMas[i];
```

```

    *Max = TempMax;
    return *Max; // Когда
};

```

Функция поиска максимального значения в массиве, заданном указателем (**iMas**), размерностью (**Razm**), тип возврата **int**, возвращаемое максимальное значение указатель (**Max**).

6.1.5. Прототипы функций

Если функция должна быть вызвана в программе одного модуля (например, см. выше `first.cpp`), а ее описание дано в другом исходном модуле (см. `second.cpp`), необходимо задать прототип функции – краткое описание заголовка функции без ее тела. Даже при описании функции в одном исходном модуле требуется прототип, если вызов функции планируется до ее описания (оно может располагаться ниже в исходном модуле). Прототип позволяет компилятору проконтролировать правильность задания параметров при вызове функции. Прототип задается так:

<Прототип функции> := [<Спецификация типа возврата функции>] <Название функции> ([<Список типов параметров [с тегами]>]);

Тело при задании прототипа функции отсутствует, вместо списка параметров задается список типов, в которых можно условно указать любые имена (они иногда называются тегами). Эти имена являются своего рода подсказками и не обязательно должны совпадать с именами формальных параметров, задаваемых в описании функции. Прототипы функции, приведенной выше, могут быть заданы так (все варианты правильные):

```

int MaxMas ( int * iMas , int Razm, int * Max); // Вариант 1
int MaxMas ( int *, int, int *); // Вариант 2
int MaxMas ( int * piMas , int iRazm, int * piMax); // Вариант 3
int MaxMas ( int * piMas , int iRazm = 10, int * piMax); // Вариант 4, (10) 2-й
параметр по-умолчанию

```

Отметим на будущее, что прототип определяет так называемую сигнатуру описания функции. Если сигнатуры функций различны, например, отличается число параметров (или их типы), то функции в СИ++ могут иметь одинаковые имена. Такая технология программирования называется перегрузкой функций.

6.1.6. Вызовы функций и возврат значений функции

Вызов функции - это передача управления с возвратом в тело заданной функцией с предварительной настройкой на те параметры, которые указаны при таком вызове. Эти параметры в этом случае называются фактическими. В качестве параметров могут быть задаваться выражения необходимого типа. Если в функции параметр может быть изменен, то он может задаваться указателем. В случае выражения значением его тоже должен быть указатель.

Формально вызов функции можно записать так:

<Вызов функции> <Название функции> ([<Список выражений для каждого типа параметров функции>]);

Или упрощая для понимания можно записать так:

<Вызов функции> <Название функции> ([<Список фактических параметров>]);

Примеры вызова функции **Summa**(описание смотри выше):

```
int Sum;
int a = 5 , b = 5;
Sum = Summa(2,3); // Вызов с константами
Sum = Summa(a, b); // Вызов с переменными
Sum = Summa(a, a + b); // Вызов с выражением
Sum = Summa(Summa(a,b) ,Summa(2,4)); // Вызов с вызовом другой функции
```

Пример вызова функции **MaxMas**:

```
// Описание массивов
int iMas[5] = {1,2,3,4,5}; // 0 - 4
int iMas1[] = {1,2,3,1,1,1,1,1}; // 0 - ?
int MaxM , c;
// Вызов Функции для массивов
c = MaxMas (iMas , sizeof(iMas)/sizeof(int) ,&MaxM);
printf ("Максимум в массиве iMas = %d \n " , c );
c = MaxMas (iMas1 , sizeof(iMas1)/sizeof(int) ,&MaxM);
printf ("Максимум в массиве = %d \n" , c );
//
```

Передача параметров в СИ в функцию выполняется по значению фактического параметра, а это приводит к тому, что в явном виде изменить параметр в функции невозможно. Например, после вызова функции (из исходного модуля проекта - **second.cpp**):

```
// Попытка возврата суммы через параметр sum
int Summ2 (int a , int b, int sum)
{
    sum = (a + b);
    return sum; // возвращаемое значение функции
};
```

...

В главной программе (**first.cpp**):

```
...
int Summ2 (int a , int b, int sum);
...
```

```
//
int SUM = 10;
Sum = Summa2(2, 3, SUM ); // Вызов с константами
// Получим значение SUM = 10 (Осталось старым) , а Sum = 5 (изменилось);
```

Для обеспечения правильного возврата через параметр в функцию нужно передать параметр-указатель (из **second.cpp**):

```
int Summ3 (int a , int b, int * psum) // Параметр указатель
{
    *psum = (a + b);
    return *psum; // возвращаемое значение функции
};
//
...
```

Тогда в главной программе (**first.cpp**) имеем:

```
...
int Summ3 (int a , int b, int * psum);
//...
Sum= Summ3 (2 , 3, &SUM // Адрес массива); // Вызов с указателем
//Получим Sum= 5 и SUM = 5
```

В дополнение к двум рассмотренным вариантам возврата значений из функций (через возврат функции одного параметра - **return** и через параметр указатель), принципиально, можно вернуть значение после ее работы и через глобальную переменную (у нас переменная **GSum**), хотя этот стиль программирования очень плохой, с позиций надежности программы. Все равно покажем пример (из **second.cpp**):

```
// Возврат суммы через глобальный параметр GSum
extern int GSum; // Для доступа к глобальному параметру из (first.cpp)
int Summ2 (int a , int b, int sum)
{
    sum = (a + b);
    GSum = sum; // возвращаемое значение через глобальную переменную
    return sum; // возвращаемое значение функции
};
...
```

В главной программе (**first.cpp**):

```
...
int Summ2 (int a , int b, int sum);
int GSum;
...
```

```
Sum = Summa2(2, 3, SUM ); // Вызов с константами
// Значение SUM = 10 , а Sum = 5 GSum = 5
```

6.1.7. Переменные в функциях

В пределах составного оператора (“тела функции”) доступны для операций, выполняемых внутри функции следующие данные:

- Формальные параметры, передаваемые при вызове функции.
- Локальные переменные, которые описаны в теле функции.
- Константы разного типа.
- Глобальные параметры данного исходного модуля (где описана функция) и в других модулях, которые доступны с помощью спецификатора **extern**.

Из одной функции могут быть вызваны другие функции и т.д., что позволяет спроектировать иерархическую систему взаимных вызовов функций (архитектуру проекта).

Если в функции должны быть заданы такие формальные параметры, которые не должны быть в ней изменены. Для этого используется спецификатор **const** для каждого константного формального параметра.

```
// Попытка изменения константного параметра "a" - const
int Summ0 (const int a , int b, int * sum)
{
    a = 5; // НА ДАННОМ ОПЕРАТОРЕ КОМПИЛЯТОР ВЫДАЕТ ОШИБКУ!!!
    sum = (a + b);
    return *sum;
};
```

Формальный параметр “**a**” не может быть изменен (присвоен) в функции. Модификатор **const** может быть использован для характеристики всей функции, это означает, что данная функция не может изменять любые параметры.

6.1.8. Параметр массив в функции

Массив может быть передан в функцию следующими способами:

- Фиксированное число элементов в массиве
- Через указатель на массив и его размер
- Задание нулевого элемента в конце массива (ограниченное применение)

В некоторых случаях размер массива может быть фиксированным, тогда можно воспользоваться описаниями и вызовами, представленными ниже:

```
// Заранее фиксировано число элементов в массиве - 5
int Summ51 (int mas[5] , int * psum)
{
    int sum = 0 ;
```

```

for (int i = 0 ; i < 5 ; i++ )
    sum = sum + mas[i];
*psum = sum ;
return *psum; // возвращаемое значение функции
};

```

...

В основной программе:

```

int iMas[5] = {1,2,3,4,5}; // 0 - 4
printf ("Сумма в массиве iMas = %d \n" , Summ51 ( iMas, &Sum) );

```

..

Можно в предыдущем случае использовать для размера массива и **#define** переменные (переменные этапа компиляции).

При передаче размера массива в качестве параметров описание функции может иметь следующий вид:

```

// Через указатель на массив и его размер (Razm - формальный параметр)
int Summ5 (int * mas ,int Razm , int * psum)
{
    // тело функции
    int sum = 0 ;
    for (int i = 0 ; i < Razm ; i++ )
        sum = sum + mas[i];
    *psum = sum ;
    return *psum; // возвращаемое значение функции
};

```

...

В основной программе, размер массива вычисляется динамически (подчеркнуто):

```

int iMas[5] = {1,2,3,4,5}; // 0 - 4
printf ("Сумма в массиве iMas = %d \n" , Summ5 ( iMas, sizeof(iMas)/sizeof(int)
,&Sum) );

```

...

Если не предполагается в массиве хранить нулевые элементы, то в конце, массива в качестве ограничителя размера, можно поместить нуль и организовать цикл обработки до первого нуля. Роль нуля может играть и “-1”. Функция имеет вид представленный ниже, отметим, что в этом случае размер не передается.

```

// Нулевой элемент - конец массива
long Summ6 (int * iMas, long * sum)
{
    int i = 0;

```

```

while (iMas[i] != 0)
{
    *sum = *sum + iMas[i];
    i++;
};
return *sum;
}
...

```

В основной программе для работы алгоритма должно быть:

```

int iMas[] = {1,2,3,4,5, 0}; // 0 - 4
long SumLong = 0;

printf ("Сумма в массиве iMas = %d \n" , Summ6( iMas, &SumLong) );

```

Передача размерности может быть выполнена через глобальную переменную, переменную этапа компиляции. Необходимо знать, что в СИ границы индексов не контролируются. Если массив имеет несколько измерений (например, двумерный массив), то размер каждого измерения должен передаваться отдельно.

6.1.9. Размещение функций

Описание функции конкретного проекта могут быть размещены в следующих составляющих многомодульной программы:

- В этом же программном модуле в его начале (до **main**), в этом случае прототипа функции задавать не надо.
- В этом же программном модуле в его конце (после **main**), в этом случае прототип функции задавать обязательно.
- В другом исходном модуле, прототип должен быть задан обязательно в либо начале главного модуля или либо в подключаемом к нему заголовочном файле.
- В подключаемом заголовочном файле, прототипа в этом случае задавать не нужно.

При невнимательном описании возможно сообщение компилятора переопределения имен (**multiply defined**). Качество проекта во многом зависит от того, как грамотно расположены функции в разных модулях и распределены для программирования по различным разработчикам, составляющим команду программистов данного проекта.

6.1.10. Рекурсивные функции

В СИ допускается использовать рекурсивные функции, которые могут вызывать сами себя. Пример рекурсивной функции можно показать при вычислении факториала натурального числа. Описание функции вычисления факториала (в математике - **n!**):

```

int fact( int n)
{
    int rez;

```



```

if ( n == 0 )
    return rez = 1;
else
    return rez = n * fact ( n - 1 );
};

```

Вызов функции вычисления факториала (**fact**) из основной программы:

```

//
printf ("fact 5 = %d\n" , fact(5) );
..

```

Полученный результат:

```
fact 5 = 120
```

В литературе можно познакомиться и с другими вариантами рекурсивных функций. Они широко используются в алгоритмах комбинаторных вычислениях.

6.1.11. Макросы и переменные этапа компиляции

В базовом языке СИ (и, конечно, в C++) предусмотрены возможности задания макросов (макрокоманд) или переменных/выражений этапа компиляции (иногда их называют препроцессорными переменными). Для их задания используется директива препроцессора **#define**. Переменная или макрос этапа компиляции задается так:

```
#define <имя этапа компиляции> <пробел “ ”> <выражение этапа компиляции>
```

или

```
#define <имя макроса>(<параметр>, ..., <параметр>) <текст на языке, содержащий параметры>
```

Например, для размерности массива мы можем задать переменную NMAX:

```

#define NMAX 10 // Описание переменной этапа компиляции
...
int iMas[NMAX]; // Использование этой переменной для задания размерности массива
...
for ( int k = 0; k < NMAX ; k++ )... // Задание числа повторений цикла

```

С помощью специальных директив препроцессора (**#ifndef** и **#ifdef**) можно проверить определена ли переменная этапа компиляции к данному моменту обработки текста, и вставить в программу новый фрагмент текста:

```

#ifndef MyLibrary
#include <my_lib.h> // Подключение заголовочного файла

```

`#endif`

Примечание. Подстановка не производится в комментариях программы и текстовых константах или литералах.

При использовании макросов (макрокоманд) можно задавать параметры, на которые будет настраиваться текст макроопределения. При описании макросов задаются формальные макропараметры, которые должны быть текстовыми. Для обращения к макросам используется макровывозы, которых может быть много. Такие параметры называются фактическими макропараметрами. Определение макроса выполняется на основе следующего формального правила:

`#define <имя макроса>(<параметр>, ..., <параметр>) <текст на языке, содержащий параметры>`

Имена формальных параметров должны быть уникальными в пределах описания. Между именем макроса и открывающей скобкой не должно быть пробелов. Если макрос продолжается на следующую строку текста, то используется обратная наклонная черта (“\”). Макровывоз может быть размещен в тексте программы после определения макроса и содержит конкретные параметры. Примеры макрокоманд и макровывозов:

```
// Описанные макросы с параметрами
#define max(a,b) ((a>b)?a:b) // Макрос вычисления максимума их двух переменных
#define Swap(type,a,b) {type t;t=a;a=b;b=t;} // Макрос кода программы
...
// Макросы
int imax = max(3,5); // Макровывоз max с константами
printf ("Максимум из двух = %d \n",imax);
// Аналогично imax = ((3>5)?3:5);
a=10 ; b = 20;
imax = max(a, b); // Макровывоз max с переменными
printf ("Максимум из двух = %d \n",imax);
// Аналогично imax = ((a>b)?a:b);
int x = 5 , y = 10 ;
printf ("До Swap  x , y  %d %d \n",x , y);
Swap(int,x,y);
printf ("После Swap x , y  %d %d \n",x , y);
// Аналогично {int t;t=x;a=y;b=t;};
// Макрос с типом переменной
double d1 = 5.5 , d2 = 10.5 ;
printf ("До Swap  d1 , d2  %f %f \n",d1 , d2);
Swap(double,d1,d2);
printf ("После Swap d1 , d2  %f %f \n",d1 , d2);
// Аналогично { double t;t=d1;a=d2;b=t;};
```

Результат будет таким:

```

Максимум из двух = 5
Максимум из двух = 20
До Swap   x , y   5 10
После Swap x , y  10 5
До Swap   d1 , d2  5.500000 10.500000
После Swap d1 , d2 10.500000 5.500000

```

Примечание. Использовать и разрабатывать макросы необходимо очень внимательно, так как при макроподстановке возможны различные ошибки: типов переменных, ошибки повторных описаний переменных и т.д. Такие ошибки трудно обнаружить, так как на этапе компиляции невозможно использовать отладчик. В нашем примере, если неверно указать тип переменных (вместо double задать int), выполнится неявное округление переменной и результат получится неверным. Проверьте это на практике. Если макрос разбивается на несколько строк, то между ними ставиться слеш - “/”.

6.1.12. Параметры главной функции

Главная функция программы (**main**) может использоваться с параметрами, формат которых следующий:

`void main (int argc, [char * [] argv, [char * [] env]]`), где

argc – задает число параметров командной строки, если равно 1 то параметров нет.

Argv – массив указателей на строки представляющие параметры командной строки.

Env - массив указателей на строки для переменных окружения.

Небольшая программа позволяет вывести значения параметров командной строки и переменных окружения.

```

void main( int argc, char * argv[] , char * env[] )
{
    ...

    // Число параметров командной строки
    printf ("Число параметров командной строки = %d\n" , argc );

    // Распечатка списка параметров
    printf ("Параметры командной строки:\n" );
    if ( argc >0)
    {
        for (int i = 0 ; i < argc ; i++)
            printf ("Номер - %d Значение =%s \n" , i+1 , argv[i] );
    };

    // Распечатка переменных окружения (set – переменные для текущего процесса)

```

```
printf ("Переменные окружения:\n" );
i = 0;
while ( env[i] !=NULL )
{
printf ("Номер - %d  Значение =%s \n" , i+1 , env[i] );
i++;
};
```

Результат распечатаем не полностью, так как большой объем переменных окружения:

```
Число параметров командной строки = 3
Параметры командной строки:
Номер - 1  Значение =i:\2014_2015\kaf\оп\лр\prog\lr5_op\debug\LR5_OP.exe
Номер - 2  Значение =aaa
Номер - 3  Значение =ddd
Переменные окружения:
Номер - 1  Значение =ALLUSERSPROFILE=C:\Documents and Settings\All Users
Номер - 2  Значение =APPDATA=C:\Documents and Settings\serge\Application
Data
... (N.B. - параметров значительно больше!)
```

Первый параметр командной строки (**argv[i]**) всегда задает имя выполняемой программы, а два других мы ввели в параметры проекта: Project-> Debugging -> Comand Arguments -> aaa ddd.

6.1.13. Inline функции

В языке СИ предусмотрена возможность предписания компилятору не вызова функции (передачи управления к операторам функции), а непосредственной вставки операторов функции в текст основной программы. В ряде случаев это приводит к экономии памяти и времени выполнения программы. Такие функции называются встраиваемыми и имеют спецификатор **inline**. Пример встраиваемой функции и ее использования приведен ниже:

```
//описание inline функция
inline int even (int x)
{
return ! (x%2); // возврат по модулю 2 четное 1 (истина) нечетное 0 (ложь)
};
...
```

Пример вызова в основной программе:

```
//вызов inline функции
i =10;
```

```

if (even (i)) // встраивается операция взятия по модулю с отрицание
// это эквивалентно выражению - if (!(i%2))
printf ("Число %d является четным\n", i );
else
printf ("Число %d является нечетным\n", i );
i = 5 ;
if (even (i))
printf ("Число %d является четным\n", i );
else
printf ("Число %d является нечетным\n", i );

```

В результате получим:

```

Число 10 является четным
Число 5 является нечетным

```

6.1.14. Указатели на функции

В языке C++ предусмотрена возможности описания указателей на функции. Покажем пример их применения. Сначала опишем простые функции для демонстрации использования указателей на функции.

```

// Функции для указателей
//
int fun1 (int i) { return i=5;};
//
int fun2 (int i) { return i=10;};
//...

```

В основной программе, указатель на функцию (**pFun**) вычисляется динамически:

```

int i , j , k =5;
int (* pFun) (int); // указатель на функцию с параметром int
pFun = &fun1;
i = (pFun)(k); // выражение одинаковое для вызова функции через указатель
printf ("pFun = &fun1 => %d\n" , i );
pFun = &fun2; //
j = (pFun)(k); // выражение одинаковое для вызова функции через указатель
printf ("pFun = &fun2 => %d\n" , j );
j = pFun(k); // можно и так
printf ("j = pFun(k) => %d\n" , j );

```

В результате получим:

```

pFun = &fun1 => 5
pFun = &fun2 => 10

```

```
j = pFun(k) => 10
```

6.1.15. Библиотеки стандартных функций

В системах программирования предусматривается много стандартных библиотек для функций различного назначения (например, для работы со строками, выполнения ввода и вывода, работы с массивами и т.д.). Эти библиотеки подключаются с помощью заголовочных файлов или пространств описаний (пространств имен в C++ - **namespace**). Кроме заголовочных файлов для использования библиотек подключаются специальные модули (иногда они подключаются автоматически), содержащие описания функций (*.lib или *.dll). Пример подключения библиотек ввода/вывода и библиотек для работы с математическими, системными функциями и других:

```
#include <math.h>    // Математическая библиотека функций
#include <process.h>  // Системная библиотека функций
#include <string.h>   // библиотека функций для работы со строками
#include <stdlib.h>   // Стандартная библиотека разных функций
#include <locale.h>   // библиотека локализации программ
```

Стандартных библиотек и классов для описания строк очень много. Нужно хорошо знать их назначение и их состав для использования в программах. Чем лучше знания о библиотеках, тем быстрее и безошибочно можно создать сложную программу. В современных системах программирования доступны (большом количестве) библиотеки классов, которые описывают новые дополнительные типы данных. Для детального знакомства с библиотеками нужно использовать: литературу, справочники и MSDN [3] в локальном варианте (нужно установить вместе с VS) или в Интернет.

6.2. Примеры программы с использованием функций

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 ветвления (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 нужно сделать в отладчике.

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.

6.2.1. Примеры, описанные в теоретической части ЛР

Нужно внимательно изучить и проверить работу всех примеров из теоретической части ЛР. Эти примеры расположены выше (раздел 3). Все эти примеры можно скопировать в свой тестовый консольный 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 части, там представлены следующие примеры:

1. Простая функция суммирования 2-х целых (**Summa**).
2. Функция максимума в целом массиве (**MaxMas**).
3. Функция с попыткой возврата значений (**Summ2**).
4. Функция с возвратом указателя (**Summ3**).
5. Функция с константным параметром (**Summ0**).
6. Функции с передачей массива в качестве параметра (**Summ51** , **Summ5**).
7. Функция с массивом с нулевым элементом (**Summ6**).
8. Рекурсивная функция факториала (**fact**).
9. Макросы (**max** и **Swap**).
10. Пример с распечаткой параметров командной строки и окружения.
11. Пример с **inline** функцией.
12. Пример с вызовом функции через указатели.

Кроме этого ниже представлены примеры, которые могут быть полезными, в том числе и при выполнении контрольных заданий. Их тоже можно изучить и проверить, хотя это задание необязательное для выполнения.

6.2.2. Пример с функцией SWAP для целых

Описание функции SWAP:

```
void SWAP(int * a, int * b)
{
    int Temp;
    Temp = *a;
    *a = *b;
    *b = Temp;
}
```

Вызов функции SWAP:

```
//SWAP функция
void SWAP( int *, int *); // Прототип
int k1 = 1 , k2 = 2;
printf ("До SWAP функция k1 , k2  %d %d \n",k1 , k2);
SWAP(&k1 , &k2);
printf ("После SWAP функция k1 , k2  %d %d \n",k1 , k2);
```

Результат работы фрагмента программы:

```
До SWAP функция k1 , k2  1 2
```

После SWAP функция k1 , k2 2 1

6.2.3. Пузырьковая сортировка целого массива (без функции)

Использование функции SWAP из предыдущего примера:

```
// Пузырьковая сортировка
int iMas[6] = {1,2,3,4,5,0}; // 0 - 5
int Razm = sizeof(iMas)/sizeof(int);
// Вывод начальный
for (int i = 0 ; i < Razm ; i++ )
    printf ( "%2d \n" , iMas[ i ] );
    printf ( "\n" );
int Flag = 0;
// Сортировка
for (int k= 0 ; k<Razm - 1 ;k++ )
{
    Flag = 0;
    for (int i = 0 ; i < Razm - k - 1 ; i++)
    {
        if ( iMas[ i ] > iMas[ i+1] ) // убывание
        // if ( iMas[ i ] < iMas[ i+1] ) //возрастание
        { SWAP( &iMas[ i ], &iMas[ i+1] ); Flag = 1;};
    };
    if (Flag == 0) break;
};
// Вывод после сортировки
for (int i = 0 ; i < Razm ; i++ )
    printf ( "%2d \n" , iMas[ i ] );
    printf ( "\n" );
//////////
```

Результат сортировки по возрастанию:

1
2
3
4
5
0

0

1
2
3
4
5

6.2.4. Сортировка минимакс

Также использование функции **SWAP** из предыдущего примера (не путайте с макросом Swap!):

```
int iMas[6] = {1,2,3,4,5,0}; // 0 - 5
int Razm = sizeof(iMas)/sizeof(int);
// Вывод начальный
for (int i = 0 ; i < Razm ; i++)
    printf ( "%2d \n" , iMas[ i ]);
    printf ( "\n" );
Flag = 0;
// Сортировка
for (int k= 0 ; k<Razm - 1 ;k++ )
{
    Flag = 0;
    for (int i = 0 ; i < Razm - k - 1 ; i++)
    {
        // if ( iMas[ i ] > iMas[ i+1] ) // возрастание
        if ( iMas[ i ] < iMas[ i+1] ) // убывание
        { SWAP( &iMas[ i ], &iMas[ i+1] ); Flag = 1;}
    };
    if (Flag == 0) break;
};
// Вывод после сортировки
for (int i = 0 ; i < Razm ; i++)
    printf ( "%2d \n" , iMas[ i ]);
    printf ( "\n" );
```

Результат сортировки по убыванию:

0
1
2
3
4

5

5

4

3

2

1

0

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

Лекция 7 – Структуры

7. Лекция № 7 – Структуры. Основные понятия

7.1. Основные понятия

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

7.1.1. Проблемы хранения и обработки данных

Самыми важными функциями современных компьютеров являются функции хранения и обработки информации. Количество хранимой в компьютерах информации возрастает многократно, поэтому то, как эти данные хранятся, существенно влияет на эффективность процессов использования этой информации.

Одним из решений наглядного и обозримого способа хранения информации, является ее структуризация и абстрагирование. Структуризация подразумевает, что данные, связанные между собой хранятся вместе, а абстрагирование позволяет отвлечься от детализации информации на определенных этапах ее обработки.

Массивы информации могли бы быть таким хранилищем связанной информации, если бы не требования ее однородности (Напомним, массив – множество однотипных переменных). В реальных задачах требования однотипности не соблюдаются. Так для описания человека, как минимум, нужны и символьные данные (ФИО) и числовые данные (даты и время) и вещественные данные (зарплата, стипендия). Для реализации технологии совместного хранения разнотипных данных в языки программирования добавлено понятие структура данных. Технология структур данных позволяет в программах реализовать более высокий уровень абстракции данных и сделать программы более наглядными.

7.1.2. Структура - элемент хранения разнородных данных - **struct**

Массивы объединяют группы переменных одного типа. В программах часто требуется группировать вместе разнотипные переменные. Для этого предусмотрены специальные описания – структуры данных (struct). Структура данных – это совокупность разнородных данных для описания отдельного информационного множества данных. Фактически структуры описывают новые типы переменных (подобно классам, но об этом в других ЛР). Нужно различать понятия:

- Описание шаблона структуры.
- Описание переменных структурного типа.

– Использование переменных структурного типа.

Описание шаблона структуры включает описание отдельных переменных (часто их называют полями структуры) с указанием их типов и заданием уникальных имен в пределах одной структуры. Формализовано это выглядит так:

```
struct <имя структуры> {
    <описание поля 1 структуры>;
    ...
    <описание поля N структуры>;
};
```

Пример описания структуры:

```
// Описание структуры Student
struct Student {
    char Name[14]; // Фамилия студента
    int kurs; // Курс обучения
    bool pol; // Пол студента: true - women, false - men
    float Stipen; // Размер стипендии
}; //
```

Допустимо описание шаблона структуры с одновременным описанием структурных переменных этого типа:

```
struct <имя структуры> {
    <описание поля 1 структуры>;
    ...
    <описание поля N структуры>;
} [<список описаний конкретных структурных переменных>;];
```

Пример одновременного описания шаблона структуры и структурных переменных:

```
// Описание структуры Student с одновременным описанием структурной переменной этого
// типа
// (Student1) и массива этого типа (Group31[30])
struct Student {
    char Name[14]; // Фамилия студента
    int kurs; // Курс обучения
    bool pol; // Пол студента: true - women, false - men
    float Stipen; // Размер стипендии
} Student1, Group31[30] ; // Необязательное описание переменных и массивов этой
// структуры
```

Допустимо отдельное описание структурных переменных, после того как описан шаблон структуры. Это выглядит так:

```
struct Student Student1, Group31[30] ;
```

Или так, таким образом, без ключевого слова **struct**, но только с названием новой структуры (кстати, ранее не допускалось):

```
Student Student1, Group31[30] ;
```

Ставить точку с запятой после описания структуры нужно обязательно. Другие примеры описания структур даны ниже. Их назначение понятно из комментариев, которые даны в тексте программы:

```
struct Complex { // Структура - комплексная переменная
double re; // действительная часть
double im; // мнимая часть
};
//
struct Date { // Структура - дата
int day; // День
int month; // месяц
int year; // Год
};
//
struct Person { // Структура - Персона
char name[50];
Date birthdate; // структурная переменная дата рожденич
double salary; // Оклад
};
```

Правила описания отдельных полей структуры совпадают с правилами описания обычных переменных. Поля могут быть любого типа, в том числе и также структурными переменными (за исключением типа самой описываемой структуры). Однако допустимо объявлять указатели на данный структурный тип в виде отдельного поля структурной переменной. Описание структурных переменных производится с указанием имени структуры как нового типа данных и может быть выполнено в программе многократно. Шаблон структуры должен быть доступен перед первым его использованием для описания переменных. Пример отдельного описания структурных переменных и их массивов:

```
struct Student S333; // Описание без инициализации полей
Student S1 = {"Петров", 1, false, 1500.0f }; // Описание с инициализацией полей
Student Group[30]; // Описание массива структур
Complex z;
Date d;
Person p;
```

7.1.3. Инициализация структурных переменных

Инициализация структурных переменных (начальные значения полей структуры), как и для массивов, может быть выполнена при описании конкретной структурной переменной. Строковые значения указываются при этом в кавычках, а инициализация массивов в фигурных скобках. Описание структурных переменных с их инициализацией:

```
Complex z1 = { 1.6, 0.5 };
Date d1 = { 1, 4, 2001 };
Person p1 = { "Сидоров", {10, 3, 1978}, 1500.48 };

// Описание структуры лицо
struct Face {
    int MasF [5]; // М ассив целых
    Date birthdate; // Структура даты
    char name[50]; // Строка
};

//Описание структурной переменной с инициализацией массива, строки и
другой структуры
Face f = { {1,2,3,4,5} , { 1, 9, 2014 } , "Представительное лицо"};

//
```

В последнем случае мы имеем пример инициализации вложенной структурной переменной: в структуре **Face** объявлена структура **Date**. Для ее инициализации добавляются фигурные скобки.

Допустимо для инициализации использовать переменные, но их значения к моменту описания новой структуры должны быть вычислены:

```
int test = 10; // Простая переменная
Person STest= { "Большаков" , test , 60000.0f} ; // Переменная задана при
инициализации
```

Допустимо для инициализации использовать и структурные переменные если они предварительно проинициализированы:

```
Person S31;
Person STest3 =STest; // Правильно
Person STest4 =S31; // Ошибка - S31 не инициализирована
```

7.1.4. Работа с полями структуры через структуру и через указатель на структуру

При описании структурных переменных ключевое слово **struct** в новой нотации перед описанием C++ необязательно, поэтому не будем его писать. Для работы с переменными такого типа недостаточно указывать только имя структурной переменной, нужно указать также и имя конкретного поля. Такой элемент программы называется квалифицированной ссылкой (имя структуры и поля разделяет точка – “.”). Такая квалифицированная ссылка рассматривается системой программирования как обычная переменная. Ее можно использовать в любых операторах и выражениях программы, без каких либо ограничений. Единственным требованием в этом случае – необходимость

предварительного описания или обеспечения доступа к этой структурной переменной на момент ее использования.

Формализовано ссылка на поле структурной переменной выглядит так:

<имя структуры>.<имя поля структуры>

Например, для структурных переменных:

```
struct Student S333; // Описание без инициализации полей
Student S1 = {"Петров", 1, false, 1500.0f }; // Описание с инициализацией полей
Student Group[30]; // Описание массива структур
```

Примеры использования конкретных полей (**kurs**) конкретных структурных переменных (**S1, Group5[]**) в операторах присваивания:

```
S1.kurs = 2;
Group5[0].kurs = 3; // Для 0-го элемента массива Group5
```

Можно использовать указатели на структурную переменную:

```
Student * pStud = &S1; // Описание указателя и его инициализация
```

В этом случае доступ к полю структуры задается не точкой, а специальной операцией из двух символов (“->”- похоже на сам указатель):

```
pStud->kurs = 5 // обращение к полю структурной переменной
```

Для работы в функциях со структурами в качестве параметра передается указатель на нее, что позволяет существенно сократить список передаваемых параметров.

7.1.5. Многоуровневая квалификация полей в структурах

Для вложенных структур (в одной структуре используется другая и т.д.) применяется много уровневая квалификация. Это необходимо при использовании других структурных переменных в качестве полей основной структуры (в **Face** описано поле типа **Date** – см. ниже). Доступ к полю в этом случае выполняется через двойную квалификацию:

```
struct Date { // Структура даты
    int day;    // День
    int month;  // Месяц
    int year;   // Год
};
// Описание структуры лицо
struct Face {
    int MasF [5]; // Массив целых
    Date birthdate; // Структура типа даты
    char name[50]; // Строка
```

```

};
...
// Описание структурной переменной
Face Face2;
...
// Использование поля и вложенного поля day
Face2.birthdate.day = 30; // Двойная квалификация
...

```

В программах не исключается возможность вложений структурных переменных с большим числом уровней. Например (**Person - Student - Date**):

```

struct Date { // дата
int day; // День
...
};
//
struct Student{
...
Date birthdate; // Подструктура типа даты
...
};
//
struct Person{
...
Student Stud; // Подструктура типа даты
...
};
// Описание структурной переменной
Person Person2;
...
Person2.Stud.birthdate.day = 30; // Тройная квалификация

```

7.1.6. Указатели на структуры и на динамические структуры

Мы уже говорили, что на структуру можно задать указатель. Пусть есть такое описание указателя:

```

struct Date { // Структура даты
int day;      // День
int month;    // Месяц
int year;     // Год
};

```



```

struct Person { // Персона
char name[50];
Date birthdate; // структурная переменная дата рождения
double salary; // Оклад
};

...
// Описание структуры и ее инициализация
Person p2 = { "Сидоров", {10, 3, 1978}, 1500.48 };
...
// Описание указателя и его инициализация
Person * pPerson = &p2;
...
// Изменить оклад
p2.salary = 100.50; // Без указателя
pPerson -> salary = 10.50; // С указателем
(*pPerson).salary = 12.50; // Можно и так (разименование указателя),
скобки обязательны

```

Если память под структуру выделена динамически (функцией **malloc** или операцией **new**), то также должен использоваться указатель на структуру (**pPerson**):

```

...
// Выделить динамическую память
pPerson = (Person *) malloc (sizeof(Person));
pPerson -> salary = 5.5;
strcpy(pPerson -> name, "Петров"); // Копирование строк
pPerson -> birthdate.year = 1995; // Комбинированная квалификация (и
стрелка и точка)
// Освободить динамическую память
free(pPerson);
...

```

Если в структурах описаны указатели на структуры (Например, **pStudent**), то квалификация указателями выполняется двойными стрелками (здесь внутри одной структуры задан указатель на вложенную структуру **Student**):

```
ptrPerson -> pStudent -> salary = 15.00;
```

или такого вида (где имеет место двойная квалифицированная ссылка):

```
p2.birthdate. year = 2014;
```

Напомню, что в структуре допускается использовать указатель на “себя” – структуру такого же типа, это часто применяется для создания списковых структур данных.

7.1.7. Передача структур в функцию

Передать структуру в функцию, в качестве фактического параметра можно двумя способами: по значению, тогда изменить структуру в функции нельзя, или передать указатель на структуру. Рассмотрим здесь первый случай. Пусть разработана функция для печати отдельного поля структуры (**name**) и изменения другого поля (**salary**):

```
// Описание структуры
struct Person { // Структура - Персона
    char name[50];
    Date birthdate; // структурная переменная дата рожденич
    double salary; // Оклад
};

// Описание функции печати поля структуры (name)
void PrintPersonName( Person per)
{
    printf ("Печать в функции per.name = %s\n" , per.name);
    per.salary = 5.0; // зменение поля структуры
    // Для проверки изменения поля структуры в функции
    printf( "Печать внутри функции per.salary = %f \n" ,per .salary );
};

...
```

Зададим прототип функции печати проля структуры в главной программе

```
void PrintPersonName( Person per);

...
```

В главной программе зададим описание структуры и выполним вызов функции:

```
// Описание структуры
Person p2 = { "Сидоров", {10, 3, 1978}, 15.00 };

//Вызов функции параметром структура
printf( "До функции p2.salary = %f \n" ,p2 .salary );
PrintPersonName( p2 ); // p2 – имя структуры в функцию передача по значению
printf( "После функции p2.salary = %f \n" ,p2 .salary ); //Поле в функции не
изменилось
```

Получим результат:

```
До функции p2.salary = 15.000000
Печать в функции per.name = Сидоров
Печать внутри функции per.salary = 5.000000
После функции p2.salary = 15.000000
```

Значение полей структуры (в частности **p2.salary**) не изменяются при передаче всей структуры. Подумайте почему.

7.1.8. Передача указателя на структуру в функцию

При передаче в функцию всей структуры в качестве параметра, значения полей изменить нельзя, так как передача в СИ выполняется по значению (через стек передается копия структуры). Если мы хотим изменять значения полей в структурной переменной внутри функции, то в нее необходимо передать указатель на эту структуру. Функция передачи указателя на структуру, имеет вид (опишем ее для разнообразия в другом файле проекта - **second.cpp**):

```
//
void ChangePersonSalary( Person * p , double newSalary)
{
    p -> salary = newSalary; // Изменение поля по указателю на структуру p поля salary
};
...
```

Прототип этой функции нужно задать в начале главной программы:

```
void ChangePersonSalary ( Person per , double Salary);
...
// Описание структурной переменной Person
Person p2 = { "Сидоров", {10, 3, 1978}, 15.00 };
...
// Печать поля до вызова функции изменения поля
printf( "До функции p2.salary = %f \n" ,p2 .salary );
// Передача указателя в функцию
ChangePersonSalary( &p2 , 30.0); // Передача указателя = &p2
// Печать поля после вызова функции изменения поля
printf( "После функции ChangePersonSalary p2.salary = %f \n" ,p2 .salary );
```

Получим результат:

```
До функции p2.salary = 15.000000
После функции ChangePersonSalary p2 .salary = 30.000000
```

7.1.9. Массивы структур

Так как структура определяет новый тип переменной в СИ, то разрешается описывать массивы структур этих переменных. Например, ниже дано описание шаблона структуры и массива таких структур:

```
// Описание шаблона структуры типа Prepod
struct Prepod {
    ...
    float Oklad; // Оклад
```

```

...
};
// Описание массива структур типа Prepod
Prepod KafIU[30];
// Работа с элементами массива структур (индексными переменными)
KafIU[0].Oklad = 10.0;
KafIU[10].Oklad = 10.0;
// Пример цикла занесения оклада для всех структурных переменных массива
for (int i=0 ; i < 30 ; i++)
{
    KafIU[i].Oklad = 10.0;
};

```

С массивом, естественно, можно работать и через указатель (**ptrMas**), при этом инициализация указателя должна быть проведена с начальным адресом массива структур (**&KafIU[0]**) или просто именем этого массива (**KafIU**), так как само имя массива задает адрес начала массива (указатель на массив структур). Примеры:

```

// или с указателем
Prepod *ptrMas = &KafIU[0];
// или
Prepod *ptrMas = KafIU;

```

Не трудно проверить, что значения вычисленных указателей **ptrMas** в первом и втором случае одинаково.

Допустимы разные способы доступа через указатели с индексацией в массиве структур (Например, так **ptrMas[2].Oklad**). Или с вычислением нового адресного выражения для указателя (**ptrMas + 2**), причем плюс 2 на самом деле эквивалентно прибавлению двойного размера одиночной структуры, то есть величины равной - **sizeof(Prepod)**. Можно также выполнить разыменование указателя структуры (*) и работать с ней как с обычной структурной переменной. Посмотрите и проверьте примеры, приведенные ниже:

```

ptrMas->Oklad = 45.0;           // Использование указателя для структуры
ptrMas[2].Oklad = 15.0;        // ИНДЕКСАЦИЯ: для второго элемента массива
структур
(*ptrMas).Oklad = 55.0;        // РАЗИМЕНОВАНИЕ указателя
( ptrMas + 2 ) -> Oklad = 25.0; // СЛОЖЕНИЕ УКАЗАТЕЛЕЙ И КОНСТАНТ
(*( ptrMas + 2)).Oklad = 35.0; // можно и так – разыменование адресного выражения
с указателем
ptrMas = ptrMas + 2;           // Явное изменение указателя (фактически плюс
sizeof(Prepod))
ptrMas->Oklad = 45.0;          // Использование вычисленного указателя для массива
структур

```

Во всех рассмотренных случаях будет изменена одно и тоже поле элемента структуры с номером **2** в массиве структур типа **Prepod**.

7.1.10. Вложенные структуры

Вложенными структурами называются такие структуры, в которых в качестве поля описана другая структурная переменная или указатель на структурную переменную. Причем поле структуры того же типа описывать запрещено, но описание указателя на саму себя допускается.

Примеры вложенных структур, где внутри одной структуры описана другая структура (тип **Prepod** - **DecPrep**) и указатель на другую структуру (тип **Person** - **pFace**) приведены ниже.

```
struct Date { // Структура Date
int day;    // День
int month;  // Месяц
int year;   // Год
};
// Структура Person с вложенной структурой Date
struct Person {
char name[50];
Date birthdate; // структурная переменная дата рождения типа Date
double salary; // Оклад
};
// Структура Prepod со вложенными указателями на структуру Person - pFace
struct Prepod {
char fam[50]; // Фамилия
Person * pFace; // Указатель на структурную переменную типа Person
double Oklad; // Оклад
};
// Структура со вложенными структурами Prepod
struct Decan {
char fam[50]; // Фамилия
Prepod DecPrep; // Структурная переменная Prepod вложенная (не указатель)
double Oklad; // Оклад
};
```

В программе для указателей поместим операторы вычисления (**salary**) стоимости:

```
Person p2 = { "Сидоров", {10, 3, 1978}, 20.00 };
// Динамическая структура – доступ через указатель
Prepod *ptrPerson = (Prepod * ) malloc ( sizeof (Prepod));
ptrPerson -> pFace = &p2;
ptrPerson -> pFace -> salary = 15.00;
// Доступ к полю динамической структуры возможен разными способами
printf( "p2 .salary = %f \n", p2 .salary );
printf( "ptrPerson ->pFace ->salary = %f \n", ptrPerson -> pFace ->salary ); // Два
указателя
```

В результате работы фрагмента программы получим на консоли:

```
p2 .salary = 15.000000
ptrPerson ->pStudent ->salary = 15.000000
```

Для вложенных структур (в структуру **Decan** вложена структура **Prepod**, см. Описания выше) можем записать и комбинированную ссылку, включая доступ и посредством указателя (**pFace ->salary**):

```
// Вложенные структуры
Decan DecIU;
DecIU.DecPrep.Oklad = 100.00;
// Доступ с указателем
DecIU.DecPrep. pFace = &p2;
DecIU.DecPrep. pFace ->salary = 10.0;
```

7.1.11. Размер и размещение структур в ОП

Поля структуры располагаются в оперативной памяти последовательно, в связи с описанием в программе. При размещении в памяти разные типы должны быть выровнены на границу адреса своего размера (**int** – 2 байта, **long** – 4 байта, **double** – 8 байт и т.д.). Из-за этого, даже при равных по количеству (общему объему) и типу полей размер структуры может отличаться. Это показано на примере двух структур одинаковых при первоначальном взгляде: **First** и **Second**.

```
struct First {
int i;
long j;
double k;
};
struct Second {
int i;
double k;
long j;
};
```

В программе определим актуальный размер структур и их полей:

```
// Размеры полей и величина размещения структур в ОП
printf( "Размер int = %d \n" ,sizeof (int) );
printf( "Размер long = %d \n" ,sizeof (long) );
printf( "Размер double = %d \n" ,sizeof (double) );
printf( "Размер структуры First = %d \n" ,sizeof (First) );
printf( "Размер структуры Second = %d \n" ,sizeof (Second) );
...
```

В результате работы программы получим:

```

Размер int    = 4
Размер long   = 4
Размер double = 8
Размер структуры First  = 16
Размер структуры Second = 24

```

При выравнивании на границу для другого порядка полей размер структуры увеличивается с 16 до 24 байт! Проверьте и другие порядки расположения полей в таких структурах.

7.1.12. Динамической структуры

Работа с динамическими структурами и их массивами выполняется посредством указателей соответствующих типов (указателей на структуры такого типа - **Decan * pDec**;). Выделение памяти будем производить производиться библиотечными функциями из **malloc.h**. В этой библиотеке доступны функции: **malloc**, **calloc**, **free**, **realloc** и др. На примере, размещенном ниже, показано применение этих функций для структур.

```

Decan * pDec = (Decan * ) malloc ( sizeof (Decan)); // Выделение динамической памяти
pDec ->DecPrep.Oklad = 100.00; // Работа с полямиуктур
strcpy( pDec ->fam , "Фамилия декана");
pDec ->Oklad = 50.00;
pDec = (Decan *) realloc( pDec , sizeof (Decan) * 2 ); // Изменение размера выделяемой
// памяти
//
pDec = pDec + 1;
strcpy( pDec ->fam , "Новая Фамилия декана");
pDec = pDec-1;//Восстановление указателя для освобождения памяти(можно и сначала
запомнить)
free ( pDec );

```

Обратите внимание на использование функции **realloc**, позволяющей изменить размер выделенной памяти в два раза, а также добавить к этому указателю единицы (**pDec = pDec + 1**). При операциях целого типа с указателями определенного вида одна единица соответствует размеру типа, для которого объявлен данный указатель (у нас - **sizeof(Decan)**).

7.1.13. Создание и удаление динамической структуры со строками

В динамических структурах часто возникает задача работы со строками (символьными массивами: имена, фамилии и другие тексты). Если заранее в структуре выделять максимальное число требуемых знаков в символьных массивах для строк (так мы поступали ранее, см. – **fam , name**), то, очевидно, будет значительный перерасход

оперативной памяти, особенно в тех случаях когда выполняется работа с большими массивами структурных переменных. Поэтому при инициализации таких структур намного экономнее выделить столько байт памяти, сколько необходимо в конкретном случае, то есть динамической памяти (ДП). Кроме того, подобная процедура необходима и при изменении значений строковых полей структурных переменных, для универсальности: первоначально выполняется освобождение памяти (**free**), а затем новый захват ДП (**malloc**) “по потребности”. Покажем это на примере. Пусть есть структура, в которой два указателя на строки (**pName** и **pAvtor**):

```
struct Book{
    char * pName; // Указатель на строку для Названия книги
    char * pAvtor; // // Указатель на строку для Автора книги
    int StrCount; // Число страниц в книге
};
```

При инициализации структурной переменной нужно захватить требуемую память, указатель запомнить в структуре и скопировать строку в полученную область ДП:

```
Book Book1; // Описание структурной переменной
char * pStr = (char *) malloc ( strlen ("Три мушкетера") + 1); // Число символов в строке +1 для \0
Book1.pName = pStr; // Запомним указатель
strcpy(pStr, "Три мушкетера"); // Копируем строку в поле ДП
pStr = (char *) malloc ( strlen ("Александр Дюма") + 1);
Book1.pAvtor = pStr;
strcpy(pStr, "Александр Дюма");
Book1.StrCount = 670;
```

При завершении программы динамическая память должна быть освобождена (функция освобождения - **free**):

```
// При завершении программы освободим память
free (Book1.pName);
free (Book1.pAvtor);
//
```

Если вся структура динамически порождается, этот текст программы будет выглядеть следующим образом:

```
// Динамическая структура
Book * pBook = (Book *) malloc ( sizeof(Book));
pStr = (char *) malloc ( strlen ("Две Дианы") + 1);
pBook->pName = pStr;
strcpy(pStr, "Две Дианы");
pStr = (char *) malloc ( strlen ("Александр Дюма") + 1);
pBook->pAvtor = pStr;
strcpy(pStr, "Александр Дюма");
pBook->StrCount = 470;
```



```
// При завершении программы освободим память под строки и саму структуру
free (pBook->pName);
free (pBook->pAvtor);
free (pBook );
/////////
```

Если нужно изменить значение содержимого в отдельной строке, то динамическую память предварительно освобождаем, а затем снова выделяем, так как размеры ДП строк могут не совпадать:

```
free (pBook->pName); // Заметим что было - "Две Дианы"
pStr = (char *) malloc ( strlen ("Дама с камелиями") + 1);
pBook->pName = pStr;
strcpy(pStr , " Дама с камелиями" );
```

Если память не освобождать, то очень скоро она переполнится (не хватит ресурса ОП).

7.1.14. Структуры с указателями на собственный тип

Выше было сказано, что структурах нельзя использовать поля типа структурных переменных такого же типа (получается бесконечное рекурсивное описание). Однако поля - указатели на эти структурные переменные допускаются. Часто такие указатели используются для описания данных типа список. Например, элемент двухсвязного списка может выглядеть так:

```
// структуры со ссылками на самих себя – Элемент двухсвязного списка
struct Node {
    Node * pNext;
    Node * pPrev;
    int ValList;
};
```

В данном случае **pNext** и **pPrev** являются указателями на структуру **Node** и используются для организации связей в списке (следующий – предыдущий).

7.1.15. Перечисления - enum

Перечисления **enum** (перенумерация) – это специальный вид структурных переменных, использующихся для универсализации программ и их большей наглядности. Например, можно перенумеровать дни недели, месяцы и т.д.

Перечисление, другими словами, - это набор именованных целых констант, используемый для большей наглядности программы и ее переносимости. Перечисление может быть использовано и для объявления переменных, которые принимают значения на заданном множестве значений. Перечисления - это по сути множество целых констант, используемых в программе на этапе компиляции. Формально перечисления (**enum**) могут быть описаны так:

enum [<имя>] {список перечисления} [список переменных];

Значение <имя перечисления> может быть опущено. Например, для дней недели можно задать список констант:

```
// Дни недели
enum { mon , tue , wed , thu , fri , sat , sun };
int d = mon; // При этом значения mon = 0 , tue = 1 и т.д.
```

Можно задать также имя перечисления для отдельного описания переменных этого перечисления:

```
// Дни недели с именем day
enum day { mon , tue , wed , thu , fri , sat , sun };
day dw = sun; // Получим dw = 6
```

Можно задать перечисления и с произвольными значениями констант, например:

```
enum { Con1 = 5 , Con2 = 8, Con3 = 15 };
printf("Перечисления с произвольными значением:   Con1=%d   Con2=%d
Con3=%d\n" , Con1 , Con2, Con3 );
```

Получим после выполнения:

Перечисления с произвольными значением: Con1=5 Con2=8 Con3=15

Константы можем проверять в условиях:

```
// Проверка значений перечислений в операторе ветвления
int m = Con1;
if( m ==Con1 )
    printf("значение: Con1 , m =%d\n" , m ); // Совпадение переменной и константы
else
    printf("значение: не Con1 \n" );
```

Получим после выполнения фрагмента:

значение: Con1 , m = 5

Можно задать перечисления с одним базовым значением:

```
enum { Const1=5 , Const2, Const3 };
printf("Перечисления с базовым значением: Const1=%d Const2=%d Const3=%d\n" ,
Const1 , Const2, Const3 );
```

Результат будет таким:

Перечисления с базовым значением: Const1=5 Const2=6 Const3=7

Более детально по технологиям работы с перечислениями вы можете познакомиться в литературе по языку СИ [1,2].

7.1.16. Союзы – union - объединения

В некоторых случаях структуру нужно использовать для данных, которые перекрываются. Для этого могут быть использованы объединения – **union**. Объединения являются полноправными структурами данных: имеют шаблон, имеют поля, могут

использоваться для описания структурных переменных. Поля в структура типа `union` перекрывают друг друга. Объединение (наложения) — это фактически выделенное место в памяти (они могут быть и динамическими), которое используется для хранения переменных, даже разных типов. Получается, что можно задавать разные имена, одним и тем же полям оперативной памяти. Покажем на примере. Пусть есть объединение содержащее три поля разной длины (Пусть это три разных типа: `int`, `float` и `char *`).

```
// Объединения union с названием - u_tag
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

После описания новой переменной (`U1`) и выполнения операций по занесению данных получим следующий результат:

```
u_tag U1; // u_tag – это новый тип данных!!!!
U1.ival = 5;
printf("Поле объединения в целом формате: u.ival=%d\n", U1.ival );
U1.fval = 0.5f;
printf("Поле объединения в действительном формате: u.ival=%5.2f\n", U1.fval );
printf("Поле объединения в целом формате: u.ival=%d\n", U1.ival ); // Попытка
печати как целое!!
U1.sval = new char[] = "Строка";
printf("Поле объединения в символьном формате: u.ival=%s\n", U1.sval );
```

Результат будет таким:

```
Поле объединения в целом формате:  u.ival=5
Поле объединения в действительном формате:  u.ival= 0.50
Поле объединения в целом формате:  u.ival=1056964608
Поле объединения в символьном формате:  u.ival=Строка
```

В третьей строке вывода для примера показана попытка печати данных как целого числа, хотя на это место было предварительно занесено вещественное число (`float`). Объединения могут использоваться при компактной передаче данных, при хранении или при плотной битовой упаковке данных (см. в литературе использование СИ для работы с битами и данных).

7.2. Примеры программы с использованием структур

Вторая часть задания в данной лабораторной работе, помимо первой, связанной с изучением теоретического раздела заключается в том, чтобы проверить и 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 ветвления (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 нужно проделать в отладчике.

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

7.2.1. 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 части (раздел 7.1), представлены следующие примеры:

- Описание простой структур **Student, Complex, Date, Person** (Разделы: 7.1.2, 7.1.3).
- Демонстрация примеров инициализации структур (Разделы: 7.1.2, 7.1.3).
- Работа со структурами через указатели и квалифицированные ссылки (Раздел: 7.1.4).
- Многоуровневая квалификация полей в структурах (Раздел: 7.1.5).
- Использование указателей для доступа к полям структур (Раздел: 7.1.5).
- Демонстрация передачи структур в функцию и указателей на структуры (Разделы: 7.1.7, 7.1.8).
- Использование массивов структур (Раздел: 7.1.9).
- Показать использование вложенных структур и доступа к полям (Раздел: 7.1.10).
- Показать определение размера структуры (Раздел: 7.1.11).
- Пример использования динамических структур и динамических строк в них (Раздел: 7.1.5, 7.1.12, 7.1.13, 7.1.14).
- Показать использование перечислений (Раздел: 7.1.15).
- Показать использование объединений (Раздел: 7.1.16).

Кроме этого ниже представлены примеры, которые могут быть полезными, в том числе и при выполнении контрольных заданий. Их тоже целесообразно изучить и проверить их работу в реальном режиме на компьютере.

7.2.2. Копирование и обмен статических структур

Если в структурных переменных все поля статические (фиксированный размер), то можно копировать структуру целиком. Отметим, что при копировании и обмене в функции параметры для структур должны задаваться указателями. Пусть есть структура следующего вида:

// Структура Person для примеров

```
struct Person {
char Name[20]; // Фамилия
int Kurs; // Курс
float Stipen; // Стипендия
```

```
};
```

...

/// Функции для структуры Person печать содержимого структурной переменной

void PersonPrint(Person P) //Для печати в функции указателя не нужно

```
{
printf( "Имя - %s Курс - %d Стипендия - %6.2f р. \n" ,P.Name, P.Kurs , P.Stipen );
};
```

// Функции копирования и обмена структур

// Копирование структур Person

void CopyPerson(Person * P1 , Person * P2) //Для изменения в функции нужны указатели

```
{
P1->Kurs = P2->Kurs;
P1->Stipen = P2->Stipen;
strcpy( P1->Name , P2->Name);
};
```

// Обмен статических структур

void SwapPerson(Person * P1 , Person * P2) //Для изменения в функции нужны указатели

```
{ // Структура статическая копируем память
Person Temp;
CopyPerson( &Temp , P1);
CopyPerson( P1 , P2);
CopyPerson( P2 , &Temp);
```

```
};
```

...

// Прототипы функций, представленных выше, в главной программе, если нужно

void PersonPrint(Person P);

void CopyPerson(Person * P1 , Person * P2);

```
void SwapPerson( Person * P1 , Person * P2);
```

```
...
```

```
// Вызов функций из главной программы
```

```
// Печать структуры
```

```
Person Stud ;//
```

```
// Заполнение структуры вручную
```

```
strcpy (Stud.Name , "Петров");
```

```
Stud.Kurs = 2;
```

```
Stud.Stipen = 2000.00f ;
```

```
// печать
```

```
PersonPrint ( Stud );
```

```
// Инициализация структуры
```

```
Person StudNew= {"Аксенова" , 10 , 50000.0f} ;//
```

```
PersonPrint ( StudNew );
```

Результат вызова функций имеет вид:

```
Имя - Петров    Курс - 2    Стипендия - 2000.00 р.
```

```
Имя - Аксенова  Курс - 10    Стипендия - 50000.00 р.
```

```
////////////////
```

```
// Копирование и обмен начальная инициализация
```

```
Person S11= {"Аксенова" , 10 , 50000.0f} ;//
```

```
Person S21= {"Большаков" , 11 , 60000.0f} ;
```

```
Person S31;
```

```
//
```

```
printf("Сору функцией: \n");
```

```
// Копирование
```

```
CopyPerson( &S31 , &S21);
```

```
PersonPrint ( S31 );
```

```
PersonPrint ( S21);//
```

Результат вызова функций имеет вид:

```
Сору функцией:
```

```
Имя - Большаков  Курс - 11    Стипендия - 60000.00 р.
```

```
Имя - Большаков  Курс - 11    Стипендия - 60000.00 р.
```

Для копирования структур без динамики можно использовать функции RTL (memcpy):

```
// Копирование структур средствами RTL
```

```
printf("Непосредственно в ОП - memcpy: \n");
```

```
Person Stud1 = { "Сидоров" , 1 , 100.00f};
```

```
Person Stud2 = { "" , 0 , 0.00f};
```

```
PersonPrint ( Stud1 );
```

```
PersonPrint ( Stud2 );
```

```
// Stud2 = Stud1; // Это возможности C++
memcpy ( &Stud2 , &Stud1 , sizeof (Person) ) ; // Так можно если нет указателей
PersonPrint ( Stud2 );
```

Результат вызова функций имеет вид:

Непосредственно в ОП - метсру:

Имя - Сидоров Курс - 1 Стипендия - 100.00 р.

Имя - Курс - 0 Стипендия - 0.00 р.

Имя - Сидоров Курс - 1 Стипендия - 100.00 р.

```
//
printf("Swap: \n");
PersonPrint ( S11 );
PersonPrint ( S21);
// Замена
SwapPerson( &S11 , &S21);
PersonPrint ( S11 );
PersonPrint ( S21);

//
```

Результат вызова функций имеет вид:

Swap:

Имя - Аксенова Курс - 10 Стипендия - 50000.00 р.

Имя - Большаков Курс - 11 Стипендия - 60000.00 р.

Имя - Большаков Курс - 11 Стипендия - 60000.00 р.

Имя - Аксенова Курс - 10 Стипендия - 50000.00 р.

Результаты выполненных фрагментов программ должны совпадать с результатами данных методических указаний.

7.2.3. Функция Swap для структур с динамическими строками

Рассмотрим простую структуру с динамическими строками(см. выше). При инициализации структуры нужно выделять динамическую память под имя (указатель **pName**) и фамилию (указатель **pFam**). В данном примере нас уже предусмотрена функция печати структуры (**StudentPrint**) и обмена структур с учетом динамики (**SwapStudent**).

```
//
struct Student { // Структура с динамическими строками
char * pName; // Динамические строка
char * pFam; // Динамические строк
int Kurs;
float Stipen;
};

// Функция печати
void StudentPrint( Student S)
```

```

{
    printf( "Фамилия - %s  Имя - %s  Курс - %d  Стипендия - %6.2f р.  \n"
,S.pFam,S.pName,
        S.Kurs , S.Stipen );
};

// Функция обмена с учетом изменения размера ДП
void SwapStudent( Student * pA , Student * pB )
{ // 1) Temp = a , 2) a = b , 3) b = Temp  => Обычный обмен!!!
    Student Temp; // Временная для хранения
    //1)
    Temp.Kurs = pA->Kurs;
    Temp.Stipen = pA->Stipen;
    Temp.pName = (char *) malloc ( strlen(pA->pName) + 1);
    Temp.pFam = (char *) malloc ( strlen(pA->pFam) + 1);
    strcpy ( Temp.pName , pA->pName);
    strcpy ( Temp.pFam , pA->pFam);
    free ( pA->pName); // Освобождение старой ДП для pA
    free ( pA->pFam); // Освобождение старой ДП для pA
    //2)
    pA->Kurs = pB->Kurs;
    pA->Stipen = pB->Stipen;
    pA->pName = (char *) malloc ( strlen(pB->pName) + 1);
    pA->pFam = (char *) malloc ( strlen(pB->pFam) + 1);
    strcpy ( pA->pName , pB->pName);
    strcpy ( pA->pFam , pB->pFam);
    free ( pB->pName); // Освобождение старой ДП для pB
    free ( pB->pFam); // Освобождение старой ДП для pB
    //3)
    pB->Kurs = Temp.Kurs;
    pB->Stipen = Temp.Stipen;
    pB->pName = (char *) malloc ( strlen(Temp.pName) + 1);
    pB->pFam = (char *) malloc ( strlen(Temp.pFam) + 1);
    strcpy ( pB->pName , Temp.pName);
    strcpy ( pB->pFam , Temp.pFam);
    free ( Temp.pName); // Освобождение старой ДП для Temp
    free ( Temp.pFam); // Освобождение старой ДП для Temp
};

```

Прототипы наших функций в главной программе:

```

void StudentPrint( Student S);

void SwapStudent( Student * pA , Student * pB );

```


Текст в главной программе: инициализация, распечатка, обмен и новая распечатка:

```
// Новая структура Student с динамическими строками (фамилия и имя)
Student Ivanov;
// Явное заполнение структуры Ivanov
Ivanov.Kurs = 5;
Ivanov.Stipen = 5000.00f ;
Ivanov.pFam = (char *) malloc (strlen ("Иванов") + 1) ; // Инициализация
Ivanov.pName = (char *) malloc ( strlen ("Иван") + 1 ) ;
strcpy ( Ivanov.pFam , "Иванов" );
strcpy ( Ivanov.pName , "Иван" );
StudentPrint( Ivanov);

// Неявное выделение динамической памяти при инициализации для структуры
Sidorov
Student Sidorov = { "Сидор" , "Сидоров" , 3 , 2000.0f};
StudentPrint( Sidorov);
// Нельзя изменять строки для структуры Sidorov
//strcpy ( Sidorov.pName , "Иван" ); // и это ошибка, так как pName указывает на
строку - константу
// Для возможности изменения нужно выделить ДП для того, чтобы работали
указатели!!
Sidorov.pName =(char *) malloc (strlen ("Петр") + 1);
strcpy ( Sidorov.pName , "Петр" );
Sidorov.pFam =(char *) malloc (strlen ("Петров") + 1);
strcpy ( Sidorov.pFam , "Петров" );
StudentPrint( Sidorov);

//SWAP для структуры Student
printf("ДО SWAP функцией: \n");
StudentPrint( Sidorov);
StudentPrint( Ivanov);
printf("SWAP функцией: \n");
SwapStudent( &Sidorov , &Ivanov ); // Обмен динамических структур – передаем
указатели
StudentPrint( Sidorov);
StudentPrint( Ivanov);
// Освободить динамическую память под строки
free (Ivanov.pFam);
free (Ivanov.pName);
free (Sidorov.pFam);
free (Sidorov.pName);

//
```

Результат вызова функций имеет вид:

```

Фамилия - Иванов  Имя - Иван  Курс - 5  Стипендия - 5000.00 р.
Фамилия - Сидоров  Имя - Сидор  Курс - 3  Стипендия - 2000.00 р.
Фамилия - Петров  Имя - Петр  Курс - 3  Стипендия - 2000.00 р.
ДО SWAP функцией:
Фамилия - Петров  Имя - Петр  Курс - 3  Стипендия - 2000.00 р.
Фамилия - Иванов  Имя - Иван  Курс - 5  Стипендия - 5000.00 р.
SWAP функцией:
Фамилия - Иванов  Имя - Иван  Курс - 5  Стипендия - 5000.00 р.
Фамилия - Петров  Имя - Петр  Курс - 3  Стипендия - 2000.00 р.

```

Синим цветом - работа функции после обмена структур.

7.2.4. Заполнение случайных - rand числовых полей массива структур

В следующем примере используются функции заполнения массива структурных переменных случайными значениями. Для этого в библиотеке СИ (**stdlib.h**) имеются специальные функции: **srand** (задание начального значения ряда случайных чисел) и **rand** (генерации вещественного случайного числа в диапазоне от 0 до **RAND_MAX**).

```

#include <stdlib.h>
...
//
struct Student { // Структура с динамическими строками
char * pName; // Динамические строка
char * pFam; // Динамические строк
int Kurs;
float Stipen;
};
...
// Заполнение статического массива структур случайными числами
const int Rzm = 6;
Student Potok[ Rzm ]; // Статический массив
// Для запуска новой последовательности случайных чисел
srand( (unsigned) time( NULL ) );
//srand( (unsigned) NULL ); // Если NULL то при новом запуске программы
последовательность //повторяется
//Цикл заполнения массива структур
for (int i = 0 ; i < Rzm ; i++ )
{
// Локальные временные массивы для генерации имен и номеров
char Buf[20];
char Num[5];
char Buf2[20];
char Num2[5];
//

```

```

strcpy(Buf , "Stud № - "); // Условная фамилия
strcpy(Buf2 , "Num - "); // условное имя
int k = ( i < Rzm/2 ) ? 1 : 2 ; // Курс для целого – условное выражение
// (1- Rzm/2) = 1, а (Rzm/2 - Rzm) = 2 (первая половина 1, а вторая 2 – без
случайностей)
// Стипендия задается 0 - 10000 Для вещественных (случайный диапазон)
float St = 1000.0f * 10.0f * rand() / RAND_MAX ;
// 0 - 99 – номер добавка для фамилии: "Stud № - " + этот случайный
номер
int n = (rand()*99)/ RAND_MAX ;
strcat(Buf , itoa (n + 1 ,Num, 10 ));
// 0 - 30 - номер добавка для имени: "Num - " + + этот случайный номер
n = (rand()*30)/ RAND_MAX ;
strcat(Buf2 , itoa (n + 1 ,Num, 10 ));
// Добавим пробелы
strcat(Buf , " ");
strcat(Buf2 , " ");

// Заполним отдельную структуру массива по индексу i (статика)
// Каждая структура статическая а строки динамические из буферов Buf и Buf2
Potok[i].Kurs = k;
Potok[i].Stipen =St;
// Выделяем память и копируем по указателю
Potok[i].pFam = (char *) malloc( strlen(Buf) +1 );
strcpy(Potok[i].pFam , Buf );
Potok[i].pName = (char *) malloc( strlen(Buf2) +1 );
strcpy(Potok[i].pName , Buf2 );
};

//

// Цикл печати массива структур через функцию
for (int i = 0 ; i < Rzm ; i++ )
    StudentPrint( Potok[i]);

//

```

Результат распечатки массива структур имеет вид, все поля кроме курса случайные:

Фамилия - Stud № - 11	Имя - Num - 3	Курс - 1	Стипендия - 1115.45 р.
Фамилия - Stud № - 82	Имя - Num - 15	Курс - 1	Стипендия - 6272.16 р.
Фамилия - Stud № - 9	Имя - Num - 6	Курс - 1	Стипендия - 4890.59 р.
Фамилия - Stud № - 86	Имя - Num - 10	Курс - 2	Стипендия - 1998.05 р.

Фамилия - Stud № - 52 Имя - Num - 19 Курс - 2 Стипендия - 4951.32 р.

Фамилия - Stud № - 80 Имя - Num - 8 Курс - 2 Стипендия - 2121.65 р.

При динамическом выделении памяти под массив (10 студентов):

```
const int Rzm = 10;
Student * pPotok = (Student *) calloc( Rzm , sizeof(Student) );
```

Работа в цикле через указатель будет выглядеть так:

```
// Заполним структуру динамического массива по индексу i
(pPotok + i ) ->Kurs = k;
(pPotok + i ) ->Stipen = St;
(pPotok + i )->pFam = (char *) malloc( strlen(Buf) +1 );
strcpy((pPotok + i )->pFam , Buf );
(pPotok + i )->pName = (char *) malloc( strlen(Buf2) +1 );
strcpy((pPotok + i )->pName , Buf2 );
```

7.2.5. Сортировка массива структур с помощью функции Swap

Сортировка массива **Potok** сгенерированного в предыдущем примере выполняется пузырьковым методом. При этом воспользуемся функцией **SwapStudent** , рассмотренной выше. Эта функции и ее применении описано выше. Текст программы сортировки приведен ниже:

```
// Сортировка массив структур Potok
for( int k =0 ; k< Rzm - 1 ; k++)
{
    for ( int i =0 ; i< Rzm - 1 ; i++ )
    {
        if ( strcmp(Potok[i].pFam , Potok[i +1].pFam ) < 0 ) // Убывание по фамилии
            SwapStudent( &Potok[i] , &Potok[i + 1] );
    };
};

// Печать после сортировки
printf("После сортировки 1: \n");
for( int k =0 ; k< Rzm ; k++)
    StudentPrint ( Potok[k]);

//
```

Результат распечатки массива структур после сортировки по фамилии имеет вид:

После сортировки 1:

Фамилия - Stud № - 67 Имя - Num - 21 Курс - 2 Стипендия - 4552.14 р.

Фамилия - Stud № - 48 Имя - Num - 19 Курс - 2 Стипендия - 2043.82 р.

Фамилия - Stud № - 43 6239.81 р.	Имя - Num - 29	Курс - 1	Стипендия -
Фамилия - Stud № - 33 8512.83 р.	Имя - Num - 17	Курс - 2	Стипендия -
Фамилия - Stud № - 28 2082.28 р.	Имя - Num - 28	Курс - 1	Стипендия -
Фамилия - Stud № - 16 9506.82 р.	Имя - Num - 22	Курс - 1	Стипендия -

Мы выбрали массив со случайной генерацией всех полей. При сортировке по фамилии текстовый номер в ней (он находится в конце строки) определяет убывание. Сортировка основана на использовании функции сравнения строк - **strcmp**.

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

8. Лекция № 8 – Файлы. Основные понятия

8.1. Основные понятия

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

8.1.1. Данные в программах

Основная цель любой программы, программного комплекса, программной системы - обработка данных. Во время выполнения программы (часто для этого этапа используется термин - **Run Time**) данные заносятся в оперативную память или сверхоперативную память (регистры процессора) и над этими данными операторы программы (команды микропроцессора) выполняют различные действия или операции. Когда программа завершилась, сохранить результат ее работы тоже получается в виде данных и их можно запомнить такими способами:

- Вывести результаты на экран дисплея и их посмотреть.
- Результат записать на “бумажке” (немного в шутку, но так тоже можно)
- Вывести результат на устройство печати (бумажная твердая копия).
- Записать результаты на внешний носитель (диски, флэшки и т.д.).

Первые три возможности не совсем удобны: с экрана информация стирается, бумага занимает много места, трудно хранить бумажные архивы или искать в них информацию, не удобно копировать полученную информацию и т.д. Результаты на “бумажке” быстро теряются.

Вывод информации на диск (дисками обобщенно будем называть любые внешние носители информации) требует выполнения определенных правил:

- нужно указать место, достаточное для запоминания совокупности информации,
- нужно присвоить имя этой информации (для ее поиска и чтения),
- нужно задать правила ее структурирования или организации, для того чтобы в дальнейшем можно было ее корректно прочесть с диска или изменить.

Кроме того, должна быть запомнена вспомогательная информация о связанной совокупности: даты создания и изменения, количество информации (размер), способы ее кодирования и т.д. Для корректной работы с информацией в ИТ системах введено фундаментальное понятие – файл. На этом понятии базируются все теории для построения программных систем обработки, хранения и преобразования информации, оно составляет основу всех ИТ - технологий.

8.1.2. Понятие файла, его определения и разновидности файлов

Существует несколько различных определений понятия файл. Это объясняется тем, что в разных условиях и разных аспектах использования (пользователь, программист, ИТ - технолог, техник по оборудованию и т.д.) для работы нужно учитывать разные свойства файлов.

Так для пользователя информационных систем важно знать название и расположение конкретного файла, возможно, его тип и размер. Заметим, что не нужно путать понятие файл с бытовым понятием файла – кармашка (из полиэтилена), хотя определенные моменты свойственные файлам и здесь тоже есть: совокупность информации, совместно хранимой.

Для программиста помимо названных выше свойств очень важна внутренняя организация файла, способы записи и чтения информации, наличие буферизации, форматирования файлов. Для технических специалистов и системщиков важной будет информация о расположении данных на физическом носителе, размеры блоков записи и кластеров, способы сборки мусора и т.д.

Файлы представлены в компьютере в электронном виде на внешних носителях (в виде нулей и единичек) и невидимы для человеческого глаза. Посмотреть и использовать содержимое конкретного файла можно только с помощью программ, способным распознать его структуру. В принципе можно работать и с файлами, работающими в оперативной памяти – виртуальными файлами.

Обобщая все необходимые для программиста и пользователя свойства файлов, мы можем, не претендуя на истину в последней инстанции, сформулировать такое определение файла:

Файл – это поименованная совокупность информации, определенного типа организации и расположенная в определенном месте памяти (внешней или внутренней) компьютерной системы, представленная в электронном виде.

В некоторых литературных источниках, например в толковом словаре [5], вы можете встретить такое определение:

Файл – это часть внешней памяти компьютера, имеющая идентификатор (имя) и содержащая данные.

На мой взгляд, даже для пользователя файлов, такое определение нельзя считать полным. Для управления файлами в компьютере его операционная система имеет специальную подсистему, которая занимается управлением файлами – файловая система. Об этом вы узнаете в курсе “Операционные системы”. Главная задача этой функции ОС управления файлами – эффективно размещать файлы на внешних носителях и обеспечивать к ним оперативный доступ для чтения и записи информации. Различают следующие файловые системы: FAT, FAT32, NTFS, HTFS и другие. Они используются в различных ОС и постоянно развиваются.

Кроме того, можно несколько “приземлить” определение файла (для его понимания и запоминания) следующим образом:

Файл – это поименованная совокупность единиц хранения информации (например, байт), расположенных в распознаваемом порядке (например, последовательно) и имеющий специальное обозначение конца совокупности данных.

Из такого взгляда на файлы можно дать еще более частное определение понятия файл:

Файл – это поименованная последовательность байтов, завершающаяся специальным символом (Конец файла – **EOF** – **End Of File**). Такие файлы могут быть текстовыми, когда на допустимые виды символов в файле наложено ограничение, и двоичными (бинарными), когда ограничений не предусмотрено, кроме символа конца файла.

Файлы можно представить также как совокупность строк, обычно так представляются текстовые файлы. В этом случае частное определение файла может иметь следующий вид.

Файл – это поименованная и упорядоченная совокупность строк, причем, в зависимости от назначения, строки могут быть как NTS (Null Terminated String), либо завешаться специальным символом конца строки - ('n' – '0A0DH').

Файл может рассматриваться как совокупность страниц (Page) или как совокупность бит информации. Файл может представлять закодированные определенным способом рисунки (совокупность разноцветных точек) или мелодии (звуковые файлы).

Файл может рассматриваться как упорядоченная совокупность однородных записей (одинаковых структурных типов), служащих для хранения и поиска информации. Такие файлы называются базами данных (БД) и имеют определенную структуру для поиска или чтения записей. В комплексной лабораторной работе по дисциплине “Основы программирования” (ЛР № 10) вы будете работать с такими файлами по своему варианту структурных переменных. Необходимо будет создать БД, добавить туда записи, отсортировать БД и так далее. Итогом работы будет завершенная программа, работающая с вашей БД.

Отдельное место во множестве разнообразных типов файлов занимают файлы, в которых записаны определенным образом программы (команды), которые выполняются на компьютерах. Тогда частное определение файла может быть таким.

Файл – это поименованная и упорядоченная совокупность команд (инструкций, операторов, процедур, функций, подпрограмм и т.д.), которые предназначены для выполнения на компьютере и написанные на определенном языке программирования, включая и машинный язык. Такие файлы называются также исполнимыми программами (модулями). Наиболее распространенные типы таких программ - *.exe, *.com или *.dll (динамические библиотеки).

Файлы могут иметь самую разнообразную и сложную структуру, о которой “знают” разработчики программ и сами программы, работающие с этими файлами. Например, данный документ (“Методические указания ...”) является файлом типа “Документ MS Word”, имеет сложную структуру (кстати, в разных версиях MS Word структура документа может отличаться). Сложность структуры и организации файлов в конечном счете, определяет сложность программ, работающих с этими файлами.

В данной лабораторной работе мы будем иметь место с относительно простыми по организации последовательными файлами: текстовыми и двоичными. Мы освоим все основные операции и технологии работы с файлами.

Необходимо также учесть, что при работе одной программы с конкретным файлом, другая программа (процесс) может запросить возможность работы с этим же файлом. Поэтому для работы с файлом должны использоваться процедуры, регламентирующие такие возможности и технологии одновременного доступа к файлам. Обычно такой контроль обеспечивает операционная система.

Подведем итог. Понятие файла, занимает значительное место в программировании и в информационных технологиях место. Файлы служат для хранения информации после и во время работы программ, являются хранилищами информации в базах данных и способом передачи информации. Файлы представляются в “невидимом” электронном формате. Их содержимое можно увидеть только с помощью специальных программ. Самыми простыми по организации являются текстовые файлы. Они запоминаются побайтно или построчно. В принципе, текстовые файлы могут рассматриваться как одна длинная строка символов. Ниже мы рассмотрим операции работы с файлами, но сначала уточним понятие операционной системы, без которой работа с файлами невозможна.

8.1.3. Операционная система, потоки и файлы

Операционная система (ОС) современного компьютера выполняет много действий связанных с файлами, как важнейшими ресурсами вычислительной системы: управление данными (непосредственно файлы), управление заданиями и процессами (программные файлы), управление устройствами (файлы располагаются на устройствах). В отдельной дисциплине на старших курсах вы будете изучать эти функции ОС. Здесь мы выделим только те особенности, которые связаны непосредственно с программированием на языках высокого уровня.

Во-первых, любая работа с файлом выполняется под управлением ОС. Это объясняется тем, что претендовать на работу с конкретным файлом в один момент могут несколько разных процессов. Для этого ОС имеет специальные списки – таблицы, разрешает или запрещает работу с файлом (открытие файла), контролирует операции работы с файлом (чтение и запись). Для работы с каждым файлом ОС создает специальный управляющий блок в ОП: блок управления файлом – FCB (File Control Block). Во-вторых, в ОС включены специальные встроенные библиотеки, обеспечивающие работу с файлами: физическое чтение и запись данных с устройств, где расположены файлы, защиту файлов и т.д.

Понятия устройств и файлов обобщены в ОС до понятия потоки ввода и вывода. Под потоком ввода понимается чтение информации с клавиатуры или любого файла (**stdin**). Под потоком вывода понимается вывод информации на экран монитора или в любой текстовый файл (**stdout**). Стандартные потоки ввода и вывода имеют фиксированные названия и могут использоваться в программах. Кроме перечисленных (**stdin**, **stdout**), в языке СИ доступен стандартный поток вывода информации об ошибках – **stderr**. Кроме стандартных пользователи могут описать произвольное число собственных потоков ввода и вывода, связанных с файлами (структура - **FILE**).

Описание потоков в заголовочных файлах ввода и вывода СИ имеет вид:

```
FILE *stdin; // Стандартный ввод данных, по-умолчанию это клавиатура
```

FILE *stdout; // Стандартный вывод данных, по-умолчанию это экран монитора

FILE *stderr; // Стандартный вывод информации об ошибках, по-умолчанию это экран монитора

Где **FILE** специальная структура для работы с файлами. В частности в нее записывается уникальный дескриптор (см. ниже) открытого файла. Она объявлена в заголовочном файле библиотеки ввода – вывода (**<stdio.h>**).

8.1.4. Имена и расширения файлов

В файловой системе компьютера файлы хранятся в каталогах, размеры которых ограничены только физическими возможностями жестких дисков (HDD). В разных каталогах могут быть записаны одноименные файлы, наличие файлов с одинаковыми именами в одном каталоге недопустимо. Суммарный размер хранимых файлов не может превышать размер физического жесткого диска.

Имя конкретного файла в ОС задается текстовой строкой в формате:

<имя файла>[.<расширение имени файла>]

Имя файла – это основное название файла, необходимое для поиска и использования файла. Расширение имени файла (необязательная часть) указывает на тип файла, его структуру и может быть использовано в программах для контроля содержания файла. Ранее в операционных системах и файловых системах старого поколения для имени файла можно было выделить до 8-ми символов, а по расширению до 3-х символов (структура “8.3”). В настоящее время ограничения на длину файла и его расширения практически сняты: сначала до 32 символов (FAT32), а затем до 255 символов (NTFS). В ОС типа Unix допустимый размер имени файла еще выше, хотя, отметим, практически это редко используется.

Примеры имен файлов приведены ниже:

Winword.exe – имя программы MS WORD,

“Отчет по ЛР студента Петрова группы ИУ5-31.DOC” – имя документа,

“Баллада о детстве - В.Высоцкий.mp3” – имя звукового файла,

DATABASE.MDB – файл БД MS Access.

Sample.txt – двоичный текстовый файл.

Пример – имя файла без расширения.

В кавычках взяты длинные имена файлов, в которых присутствуют пробелы.

8.1.5. Открытие и закрытие файлов

Важнейшими операциями для работы с файлами являются операции открытия и закрытия файлов. Необходимость таких действий обоснована выше, при характеристике действий ОС по работе с файлами. Любая программа, работающая с файлом, должна выполнять эти операции.

Первоначально для работы с файлом программа и ОС должны проверить следующее:

- существует ли файл с заданным именем в указанном каталоге,

- доступен ли он для работы в программе, и в каких допустимых режимах,
- не занят ли файл в данное время другой задачей, если система многозадачная,
- и т.д.

Такие действия выполняются специальными командами и функциями, которые под управлением ОС, и называются открытием файла (**open file**). Для работы с файлом он должен быть открыт. Функции библиотеки ввода и вывода **fopen** и **_open** и другие применяют для открытия файлов. Если файл успешно открыт, то он становится доступным для работы в программе в заданном режиме (чтение, запись, изменение, удаление и т.д.). Если файл не был успешно открыт (функция открытия сообщает о невозможности открытия – код ошибки), то необходимо искать причину ошибки открытия и устранить ее причину. После успешного открытия файла ОС выделяет для этого файла специальный дескриптор файла (передает в программу в структуру **FILE**) и выделяет специальный блок описания файла (**FCB – File Control Block**). Блок **FCB** – операционная система использует для контроля использования файла.

После работы с файлом он должен быть обязательно закрыт. После закрытия ОС освобождает файл для дальнейшего использования, удаляет блок **FCB**. Функции библиотеки ввода и вывода **fclose** и **_close** и другие применяют для закрытия файлов. Функции открытия и закрытия текстовых файлов имеют следующие прототипы:

Для открытия файла:

```
FILE * fopen( <Имя файла>, <Режим открытия файла>);
```

Или прототип:

```
FILE * fopen(const char * Filename, const char * Mode);
```

Для закрытия файла:

```
fclose( <дескриптор открытого файла>);
```

или прототип:

```
int FILE * fclose(FILE * Pf);
```

Для работы с файлами на низком уровне (без форматирования и буферизации) при открытии и закрытии файлов используются функции **_open** и **_close** соответственно (см. ниже и документацию).

Задание имени файла (**Filename**) мы уже обсуждали выше, оно задается строкой или строковой переменной. Режим открытия файлов (**Mode**) может быть задан в виде строки или перепенной следующим образом, для различных условия открытия файлов:

"r" - открыть текстовый файл только для чтения, если файла нет то ошибка.

"w" - открыть или создать текстовый файл для записи, для созданных файлов содержимое очищается.

"a" - открыть текстовый файл для добавления записей в его конец, файл создается, если он не был создан.

"rb" - открыть ,бинарный файл только для чтения, если файла нет то ошибка.

"wb" - открыть или создать бинарный файл для записи, для созданных файлов содержимое очищается.

"ab" - открыть бинарный файл для добавления записей в его конец, файл создается, если он не был создан.

"r+" - открыть текстовый файл для чтения и записи, если файла нет то ошибка.

"w+" - открыть или создать текстовый файл для чтения и записи, для созданных файлов содержимое очищается.

"a+" - открыть текстовый файл для чтения записей или добавления записей в его конец, файл создается, если он не был создан.

"r+b" - открыть бинарный файл для чтения и записи, если файла нет то ошибка.

"w+b" - открыть или создать бинарный файл для чтения и записи, для созданных файлов содержимое очищается.

"a+b" - открыть бинарный файл для чтения записей или добавления записей в его конец, файл создается, если он не был создан.

Примеры функций для открытия файлов в разных режимах:

```
FILE *pF; // Дескриптор описания файла
pF = fopen( "fputc.out" , "w+"); // Открытие текстового файла для записи и чтения
pF = fopen( "fputc.out" , "r"); // Открытие для записи
pF = fopen( "fwrite.out" , "r+b"); // Открытие двоичного файла для записи и чтения
pF = fopen( "fprintf.out" , "w+b"); // Открытие или создание двоичного файла для чтения
и записи
```

```
int pFBin = 0; // Дескриптор файла для низкоуровневого ввода/вывода
pFBin = _open( "write.bin", _O_RDWR | _O_BINARY ); // Открытие двоичного для
прямого доступа
```

Для закрытия файлов можно использовать следующие обращения к функциям:

```
fclose(pF); // закрытие текстового файла
_close( pFBin ); // закрытие бинарного/двоичного файла
```

8.1.6. Библиотеки и заголовочные файлы

Заголовочными файлами в системе программирования базового языка СИ для библиотек ввода и вывода являются следующие файлы:

```
#include <stdio.h> // Основная библиотека ввода и вывода Си - RTL
#include <fcntl.h> // Константы ввода и вывода
#include <io.h> // Функции низкоуровневого ввода и вывода
#include <conio.h> // Консольный ввод и вывод
#include <sys/types.h> // Константы в/в
#include <sys/stat.h> // Константы в/в
```

Содержание этих библиотек можно узнать: изучив документацию и литературу [1,2, 11], желательно посмотреть все примеры данных методических указаний к ЛР. Можно обратиться для помощи в **MSDN** [4] локальной и Интернет версии. Можно также открыть сам заголовочный файл (он имеет текстовый формат) и изучить его содержимое самостоятельно – прототипы функций, константы и перечисления (!!!).

8.1.7. Основные операции и функции для работы с файлами

Операции над файлами могут быть разделены на две группы:

- Действия над внутренними данными (его содержимым) конкретного файла и
- Действия над файлом в целом как отдельной единицей.

Для работы с содержимым файла предусматриваются следующие основные операции и функции:

- Создание файла, обычно происходит при открытии файла (**open, fopen, create**).
- Открытие файла (**_open, fopen**).
- Заккрытие файла (**_close, fclose**).
- Операция чтения части данных из файла (**_read, fread, fgetc, fgets, fscanf**).
- Операция записи части данных в файл (**_write, fwrite, fputc, fputs, fprintf**).
- Перемещение текущего указателя для файла и его определение (**fsetpos, fseek**).
- Определение ситуации достижения конца файла после чтения записей (**_eof, feof**).
- Получение дескриптора файла (**_fileno**).

Для работы с файлами как отдельной единицей предусматриваются следующие основные операции и функции:

- Переименование файла (**rename**)
- Удаление файла (**remove**)
- Установка и изменение атрибутов файла (**chmod, access**).
- Определение размера файла (**_filelength**)
- Определение имени файла и его местоположения (**_fullpath, div**)
- Изменение потока для файла (**reopen**)
- Системные команды ОС для работы с файлами (**system**).

Для чтения и записи используются различные возможности и функции. Работа с файлами возможна на основе стандартной библиотеки Run-Time Libraries (CRT) или библиотек потокового ввода вывода Standard C++ Library (STL). Эти библиотеки необходимо внимательно изучить и использовать при программировании. Здесь ограничимся конкретным примером для создания, чтения и записи файла. Для первого знакомства с библиотеками приводим примеры некоторых функций для работы с

файлами. Комментарии в тексте примеров поясняют работу вызываемых функций ввода и вывода.

```
#include <stdio.h> // Основная библиотека ввода и вывода Си - RTL
#include <fcntl.h> // Константы ввода и вывода
#include <io.h> // Функции низкоуровневого ввода и вывода
#include <conio.h> // Консольный ввод и вывод
#include <sys/types.h> // Константы в/в
#include <sys/stat.h> // Константы в/в

...

FILE *pF; // Описание указателя – дескриптора файла pF
pF= fopen("test.txt" , "w+"); // открытие файла для записи и создания
fputs("Пример вывода строки!!!" , pF); // вывод в файл строки
fclose(pF); // Закрытие файла с дескриптором pF
pF= fopen("test.txt" , "r+"); // Открытие файла для чтения
char FBuf[200]; // Описание буфера FBuf для ввода строки из файла
fgets((char *)FBuf, 100, pF); // чтений строки из файла в FBuf
printf( "Из файла строка = %s \n", FBuf ); // вывод на консоль буфера FBuf
fclose(pF); //Закрытие файла с дескриптором pF
```

Результат работы этой простой программы имеют вид:

Из файла строка = Пример вывода строки!!!

В тексте красным помечены места использования дескриптора, жирным выделен буфер ввода данных из файла.

В различных примерах данных методических указаний (см. ниже) и контрольных заданий вы можете использовать данные примеры.

8.1.8. Уровни ввода/вывода и типы файлов

В языке программирования СИ предусматриваются, в зависимости от поставленной задачи и характера проекта, различные технологии (и функции) для работы с файлами. Можно выделить следующие основные возможности:

- Посимвольный ввод вывод в файлах, когда текстовый файл рассматривается как одна длинная строка;
- Ввод/вывод среднего уровня (строки), в этом случае текстовый файл рассматривается как совокупность строк;
- Ввод/вывод верхнего уровня, текстовые файлы формируются на основе шаблонов форматированного вывода (Потоки, форматирование и буферизация);
- Ввод/вывод нижнего уровня, в этом случае используются бинарные файлы, а представление информации задается во внутреннем представлении компьютера (работа с байтами, структурами с внутренним представлением). В этом случае не предусмотрено форматирование данных и буферизация ввода и вывода.

- Ввод/вывод с консоли (прямой мониторный вывод и прямой ввод с клавиатуры), а также для портов ввода и вывода (другие устройства ввода и вывода – принтер, динамик и т.д.).

Ниже мы рассмотрим разные уровни и примеры работы с файлами.

8.1.9. Описание файла в программе. Структура FILE.

Для работы с файлом его дескриптор нужно описать (тип FILE *). Выше было отмечено, что само значение дескриптора формируется только при открытии файла. Возможны два варианта запоминания дескриптора в зависимости от технологии работы с файлом (ввод/вывод на нижнем уровне и ввод/вывод на верхнем уровне). Для работы на верхнем уровне можем записать так:

```
FILE *pF; // Описание указателя pF – с дескриптором файла _file
pF= fopen("test.txt" , "w+"); // открытие файла для записи и создания
```

Указатель **pF**, если файл открыт без ошибок, может использоваться до тех пор, пока файл не был закрыт. На верхнем уровне это делается так:

```
fclose(pF); //Закрытие файла с дескриптором pF
```

При открытии с ошибкой значение **pF** становится равным нулю (NULL), а специальная системная переменная **errno**, в которой фиксируется ошибка:

```
extern int errno; // Открытие доступа к глобальной переменной из другого файла
pF= fopen("test333.txt" , "r+");//Открытие файла для чтения с ошибкой 2 (ENOENT)–
файл не найден
```

Структура **FILE** для работы с файлами в этом режиме показана ниже. Как видно из описания это имя объявлено описателем **typedef**.

```
struct _iobuf {
    char *_ptr;
    int _cnt;
    char *_base;
    int _flag;
    int _file; // Декриптор открытого файла
    int _charbuf;
    int _bufsiz;
    char *_tmpfname;
};

typedef struct _iobuf FILE; // Задание типа структуры FILE
```

Описание дескриптора на нижнем уровне выполняется так, здесь нет никакой специальной структуры, вся информация о файле сохраняется на уровне операционной системы:

```
// Низкоуровневый - двоичный ввод - вывод
int pFBin = 0; // Дескриптор для работы с файлами на нижнем уровне
```

Тогда открытие файла выполняется следующим вызовом:

```
pFBin = _open( "write.bin", _O_RDWR | _O_BINARY | _O_APPEND );
```

Здесь файл открывается для чтения и записи, а также с возможностью добавления записи. При ошибках открытия функция возвращает (-1), а номер ошибки также фиксируется в переменной **errno**. Отметим, что при первоначальном открытии файл приобретает статус только чтение (**readonly**). Для записи нужно переключить атрибут чтения файла после создания. Это можно сделать системными командами:

```
// Изменение атрибутов файлов
#include <process.h> // Библиотека функции system
#include <string.h>

...

char Comand[40];
strcpy (Comand , "attrib -A ");
strcat (Comand , "write.bin");

system( Comand ); // в ОС выполняется команда attrib -R write.bin снимаем атр. "только чтение"

...
```

Закрытие файла выполняется следующим вызовом с указанием дескриптора:

```
_close( pFBin ); // Низкоуровневое закрытие файла
```

8.1.10. Текстовые и бинарные (двоичные) файлы

Мы уже упоминали основные разновидности файлов: текстовые файлы и бинарные файлы. Рассмотрим теперь, в чем заключается различие между ними. **Во-первых**, текстовые файлы могут быть использованы и в бинарном режиме. Все зависит от того, как файл был открыт. Главное различие текстовых и бинарных файлов заключается в способе представления цифровых данных. Символьные данные в этих двух случаях представляются одинаково (посимвольно). В текстовых файлах цифровые данные тоже представляются символами, т.е. фактически неявно выполняется перевод в символьное представление данных (это выполняют функции такие как - **fprintf**, **itoa** и др.). В бинарных файлах цифровое представление данных из оперативной памяти копируется в файл во внутреннем представлении данных, т.е. копируются цифровые байты и биты, а не числовые символы. Различие между способами представления информации можно увидеть на совместной распечатке шестнадцатеричного и символьного представления данных.

Для текстового файла такая распечатка выглядит так (целые числа представлены символами – код “1” - “31” и “2” - “32”, между ними пробел “20”):

```
Адр.: Шестнадцатеричное представление | Символьное предст.
0000: D1 F2 F0 EE EA E0 31 20| 21 21 21 20 31 20 32 20|Строка1 !!! 1 2
```

Такая строка в текстовый файл может быть записана следующим оператором программы (дескриптор файла - **pF**):

```
fprintf (pF , "Строка1 !!! %d %d " , 1,2);
```


Файл затем нужно закрыть и выполнить его просмотр файловым менеджером (например FAR) в комбинированном режиме.

Для бинарного файла такая распечатка выглядит так (целое число **2** – представлено во внутреннем машинном виде двумя байтами с измененным порядком следования – “**0200**”):

Адр. : Шестнадцатеричное представление	Символьное предст.
0000: C8 E2 E0 ED EE E2 00 00 00 00 00 00 00 00 00 00	Иванов.....
0000: 00 00 00 00 00 00 00 00 00 00 C9 42 02 00 00 00

Такие строки могут быть получены следующим фрагментом текста программы:

```
// Структура для вывода
struct Person { // Структура для примеров
char Name[24]; // Фамилия
float Stipen; // Стипендия
int Kurs; // Курс
};

// Описание и вывод
Person P1 = {"Иванов", 100.5f, 2 };

...

fwrite( &P1, sizeof(Person), 1, pF);

...
```

Файл затем нужно закрыть и выполнить его просмотр файловым менеджером (например, **FAR**) в комбинированном режиме.

В примерах текста и распечатках **красным** цветом отмечены значения, соответствующие друг другу. Обратите внимание еще раз, что в бинарном файле “2” кодируется как “**0200**”, младший и старший байты типа **int** (двухбайтовый тип) расположены в обратном порядке.

8.1.11. Проверка конца файла и указатель чтения файла

При обработке файлов (особенно при чтении) необходимо знать, при каких условиях цикл чтения записей должен быть завершен. Другими словами нужно задать специальное условие для проверки достижения конца файла. В общем случае, это может быть выполнено в программе следующими способами:

- Узнать заранее размер файла и считать число символов прочитанных из файла.
- Проверить конец файла специальной функцией (**_eof** или **feof** – End Of File).
- Проверить после чтения специальный байт конца файла – **EOF**- End Of File.

Проверить значение текущего числа байтов, полученных из файла к данному моменту цикла чтения, и оценить по этой величине факт завершения или продолжения цикла обработки файла (должно быть прочитано нуль байт) – например, функции **fread** или **_read**.

В примерах, рассматриваемых ниже, мы будем использовать различные способы проверки конца файла. Здесь покажем оператор для вывода размера открытого файла.

```
printf("Длина файла = %ld \n",_filelength (pF->_file));
```

Для вывода размера требуется дескриптор файла (**_file**), поэтому используется ссылка по указателю (**pF->_file**).

8.1.12. Работа с текстовым/двоичным файлом побайтно - символами

Для посимвольной записи и чтения текстовых файлов используются функции: записи одного символа - **fputc (putc)** и чтения одного символа - **fgetc (getc)**. Для проверки конца файла при чтении используется специальная функция - **feof**. Отметим, что таким способом можно работать и с текстовыми и с бинарными файлами (открытие "**w+b**").

Запись файла в цикле побайтно на основе строки (StrOut):

```
FILE *pF; // Структура описания файлов
...
// Вывод в файл
char StrOut[]="Пример строки для вывода в файл!\n";
pF = fopen( "fputc.out" , "w+"); // Открываем файл для вывода
char ch;
// Цикл вывода строки StrOut в файл
for ( int i = 0 ; StrOut[i] != '\0' ; i++)
{
    fputc( StrOut[i], pF ); // вывод одного символа
};
fclose(pF);
```

Чтение и посимвольная распечатка файла:

```
// Ввод из файла
pF = fopen( "fputc.out" , "r"); // Открытие файла для чтения
while ( !feof(pF))
{
    char ch = fgetc(pF); // можно и просто так getc(pF)
    if ( !feof(pF) ) // Для вывода маленького 'я' если ch== EOF (это 'я')
        printf( "%c", ch );
};
fclose(pF);
```

Результат посимвольного чтения файла и посимвольного вывода в окно командной строки будет таким:

Пример строки для вывода в файл!

Можно проверять и просто специальный символ конца файла (EOF):

...

```
if ( ! (ch== EOF) ) //
    printf( "%c", ch ); };
```

...

В этом случае, обратите внимание, символ “я” не выводится на печать, хотя в файле он есть. На экране получим:

Пример строки для вывода в файл!

Попробуйте самостоятельно разобраться, в чем заключается проблема с малой буквой “я”.

8.1.13. Работа с текстовым файлом построчно

Для построчной записи и чтения текстовых файлов используются функции: вывода строки - **fputs (puts)** и ввода строки - **fgets (gets)**. Для проверки конца файла при чтении также используется функция **feof**.

Запись файла в цикле построчно на основе чисел случайно сгенерированных данных для каждой выводимой строки. Чтобы они отличались в символьной строке, добавим случайный номер в пределах от 1 до 100. Цикл генерации и записи в файл показан ниже:

```
//////// Работа СО СТРОКАМИ
// Буфер ввода
FILE *pF; // Структура описания файлов
char line[81]; // память для 80 символов + символ '\0'
...

// Вывод в файл построчно (puts)
srand( (unsigned) time( NULL ) ); // для случайного числа и случайного начала
последовательности
// srand( (unsigned) 1 ); // Если 1 то при новом запуске программы случайная
// последовательность возобновляется
rand(); // для сброса первого числа последовательности
//
pF = fopen( "fputs.out" , "w+"); // Открытие текстового файла для записи
// Генерация числа 1-100 , перевод его в символьное и добавление символьного к
строке
for ( int i = 1 ; i < 6 ; i++) // цикл для 5-ти строк 1-5
{
    strcpy( line , "Строка для puts - "); // начальная строка
    unsigned int n = (unsigned int) (((double)rand()*99.0)/ (double)RAND_MAX); // 0 - 99 –
номер добавки
    char Num[10]; // Символьный буфер для перевода целого числа
    // Функция itoa переводит целое выражение (n+1) в символьное и заносит в строку
Num
    strcat(line , itoa (n + 1 ,Num, 10 )); // при переводе + 1
```

```

        strcat(line , " \n"); // добавим конец строки к строке, нужно в файле построчного
хранения
        fputs( line, pF ); // Вывод строки в файл
    };
    fclose(pF); // Закрытие файла

```

Чтение и построчная распечатка текстового файла:

```

// Ввод из файла (gets)
pF = fopen( "fputs.out" , "r"); //Открытие текстового файла для чтения
//
while (!feof(pF) )
{
    fgets( line , 80 , pF ); // Построчное чтение файла строками
    if ( !feof(pF) ) // Проверка конца файла
        printf( "Строка из файла:%s", line); // Печать. Перевод строки (\n) уже в файле!
    };
    fclose(pF); // Закрытие файла

```

Результат чтения текстового файла по строкам (случайное число 1-100 приклеено к строке) приведен ниже. Обратите внимание, что символ конца строки (\n) считывается в строке из файла, он записан в предыдущем цикле. Если его не поставить, то функция **fgets** будет считывать по 80-т символов и может достигнуть конца файла (feof) раньше. Поэтому все строки не будут распечатаны, а печать, возможно, не будет записана столбиком. Кроме этого, размер буфера в этом случае для чтения нужно увеличить.

```

Строка из файла: Строка для puts - 38
Строка из файла: Строка для puts - 29
Строка из файла: Строка для puts - 10
Строка из файла: Строка для puts - 35
Строка из файла: Строка для puts - 20

```

8.1.14. Двоичные файлы и функции fread и fwrite.

Для работы с двоичными файлами используется специальный набор функций (чтение - **fread** и запись - **fwrite**). Эти функции позволяют читать и записывать блоки данных, непосредственно в структурные переменные и из структурных переменных. Строго говоря, файлы для этих функций могут быть даже текстовыми, или просто открыты как текстовые. Для чисто двоичных файлов (нижний уровень) характерно использование других специальных функций (**_read** и **_write**). В этом случае используется специальный механизм непосредственного чтения и записи из файлов посредством операционной системы, причем без буферизации. Для демонстрации возможностей этих функций (**fread** и **fwrite**) будем использовать специальную структуру - **Person** (низкоуровневый ввод вывод будет рассмотрен ниже):

```

// Структура для вывода и вывода

```

```

struct Person { // Структура для примеров МУ
char Name[24]; // Фамилия
float Stipen; // Стипендия
int Kurs; // Курс
};

// Дескриптор файла
FILE * pF;

...

// Массив структур для записи в файл – безразмерная инициализация
Person MasPers [] = { {"Иванов", 10.5f, 1 }, {"Петров", 100.5f, 2 } , {"Сидоров",
1000.5f, 3 }};

...

```

Цикл записи файла на основе этого инициализированного массива структур. Файл открываем для записи как двоичный ("w+b"). Число циклов вычисляется от размера структурного массива:

```

// Цикл для fwrite
pF = fopen( "fwrite.out" , "w+b"); // Открытие файла для записи
for ( int i = 0 ; i < sizeof(MasPers)/sizeof(Person); i++)
    fwrite( &MasPers[i], sizeof(Person) , 1, pF); // Вывод одной строки в файл-
записи

fclose(pF);

```

Цикл чтения бинарного файла из структурных записей:

```

// Ввод записей из файла
pF = fopen( "fwrite.out" , "rb"); // Открытие файла для чтения
while (!feof(pF) ) // Проверка конца файла
{
    fread( &PWork, sizeof(Person) , 1, pF); // Чтение одной записи из файла
    if ( !feof(pF) ) // Можно ли печатать? Нет ли уже конца файла?
        printf( "Персона из файла: %s  %f  %d\n", PWork.Name , PWork.Stipen ,
PWork.Kurs );
};

fclose(pF); // Закрытие файла

```

Результат чтения файла:

```

Персона из файла: Иванов    10.500000    1
Персона из файла: Петров   100.500000    2
Персона из файла: Сидоров 1000.500000    3

```

8.1.15. Форматированный ввод и вывод в файлы

Принципы и особенности форматированного ввода и вывода были рассмотрены в лабораторной работе № 1 по курсу ОП. Отметим, что форматирование выводимой в файл информации полностью идентично (форматная строка, типы данных и списки вывода). Для вывода в файлы функция **printf** заменяется на функцию **fprintf** (вней добавляется один новый параметр - дескриптор открытого файла), а функция **scanf** должна быть заменена на функцию **fscanf**. Проверка конца файла может быть выполнена функцией – **feof**.

Запись файла **fprintf2.out** с помощью форматированного вывода:

```
// Вывод в файл fprintf
pF = fopen( "fprintf2.out" , "w+"); // Открытие файла для чтения
for (int i = 0 ; i < 5; i++)
    fprintf (pF , "Строка - %d\n" , i + 1); // Пробелы между подстроками вывода
fclose(pF); // Закрытие файла
```

В результате мы получим файл **fprintf2.out** (распечатано с помощью копии из notepad):

```
Строка - 1
Строка - 2
Строка - 3
Строка - 4
Строка - 5
```

Чтение форматированного файла. Данные в этом файле должны быть разделены пробелами, иначе:

```
// Чтение из файла fscanf
pF = fopen( "fprintf2.out" , "r+"); // Открытие файла для чтения
for ( int i = 0 ; !feof(pF) ; i++)
{
    char  buf1[80];
    char  buf2[80];
    int num;
    fscanf( pF , "%s%s%d" , buf1,buf2, &num); // Ввод форматированных данных -
разделитель пробел
    if ( !feof(pF) )
        printf( "'%s' '%s' '%d'\n" , buf1, buf2 , num ); // Печать данных из файла. ' –
одиночная кавычка
}
fclose(pF); // Закрытие файла
```

Результат чтения файла (прочитанные данные специально для наглядности поставлены в одиночные кавычки '):

```
'Строка' '-' '1'
'Строка' '-' '2'
```

```
'Строка' '-' '3'
'Строка' '-' '4'
'Строка' '-' '5'
```

Все возможности форматированного ввода/вывода для стандартных потоков (дисплей и клавиатура) доступны для файлового форматирования.

8.1.16. Низкоуровневый ввод и вывод в СИ

Для низкоуровневого ввода и вывода в СИ ("**Low-Level I/O**") используются специальные функции для открытия/закрытия файлов и функции для работы с файлами. Для этого уровня характерно использование библиотек операционной системы напрямую, что, в конечном счете, обеспечивает более эффективную работу с файлами. Для открытия закрытия используют функции: **_open** и **_close**. Для ввода и вывода: **_read** и **write**. Для проверки конца файла функция **_eof**. В качестве дескриптора файла здесь используется переменная типа **int**. Рассмотрим примеры использования такого механизма ввода и вывода.

Библиотеки и константы низкоуровневого ввода и вывода подключаются так:

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
```

Для описания и создание пустого файла нужно выполнить операторы:

```
int pFBin = 0; // Дескриптор файла
// Открытие файла для записи/чтения, создания и в двоичном режиме
pFBin = _open( "write.bin", _O_RDWR | _O_BINARY | _O_CREAT | _O_TRUNC ,
_S_IREAD | _S_IWRITE);
_close( pFBin ); // Закрытие файла
```

Вспомогательная структура "Студент" для демонстрации функций ввода вывода (Low-Level I/O) имеет следующее описание:

```
struct Student { // Сведения о студенте
char Name[20]; // Имя
int Num; // Номер
float Oklad; // Оклад
};
```

Создадим цикл записи в файл, при этом будем изменять только цифровые данные, строку **Name** изменять не будем:

```
// Структуры для циклов
Student S1 = {"Петров", 1, 1000.0f}; // Структура для записи в файл
// Открытие файла для записи/чтения и добавления
pFBin = _open( "write.bin", _O_RDWR | _O_BINARY | _O_APPEND );
// запись файла
for (int i = 1 ; i <= 5 ; i++)
{
```

```

S1.Num =i; // Меняем номер
S1.Oklad =1000.0* i; // Меняем значение стипендии
_write (pFBin , &S1 , sizeof(Student));
};
_close( pFBin ); // Закрытие файла

```

В некоторых ситуациях возникает необходимость в изменении атрибутов файлов (см. выше). Такое изменение может быть выполнено следующими командами.:

```

// Изменение атрибутов файлов
char Comand[40];
strcpy (Comand , "attrib -R ");
strcat (Comand , "write.bin");
system( Comand );
//
strcpy (Comand , "attrib -A ");
strcat (Comand , "write.bin");
system( Comand );

```

Чтение файла из структурных переменных на низком уровне и его распечатка:

```

Student SBuf = { "" , 0 , 0.0f }; // Пустая структура для чтения
pFBin = _open( "write.bin", _O_RDWR | _O_BINARY | _O_APPEND ); //Открытие
для Чтения и записи
while ( _eof(pFBin) == NULL) // Функция _eof возвращает NULL, если не конец
файла, иначе - 1
{
_read( pFBin , &SBuf , sizeof(Student)); // Чтение из файла
printf ("ФИО = %s Курс = %d Стипендия = %f \n" , SBuf.Name, SBuf.Num ,
SBuf.Oklad);
};
_close( pFBin ); // Закрытие файла

```

Результат чтения и распечатки файла:

```

ФИО = Петров Курс = 1 Стипендия = 1000,000000
ФИО = Петров Курс = 2 Стипендия = 2000,000000
ФИО = Петров Курс = 3 Стипендия = 3000,000000
ФИО = Петров Курс = 4 Стипендия = 4000,000000
ФИО = Петров Курс = 5 Стипендия = 5000,000000

```

8.1.17. Навигация по файлам. Функции - fseek, lseek

Часто чтение и запись файлов выполняется последовательно. После чтения текущей записи (байта, строки, структуры) специальный указатель (указатель расположения текущей записи) устанавливается на следующую позицию автоматически. Иногда необходимо прочитать часть файла (запись, поле, строку, блок или байт) не последовательно, а выборочно. Для этого указатель текущей записи нужно передвинуть в

нужное место. Для этого предусмотрены специальные функции перемещения текущего указателя: **fseek** – для потокового ввода/вывода и **lseek** – для низкоуровневого ввода/вывода. После перемещения указателя в другое место мы можем прочитать ту информацию (байт, строку, структуру, запись и т.д.) или заменить существующую информацию другой. Такие операции возможны для файлов, открытых в двоичном режиме. Покажем на примерах применение такой технологии.

При перемещении указателя мы можем смещаться от позиции начала файла (**SEEK_SET**), позиции конца файла (**SEEK_END**) и текущей позиции в файле (**SEEK_CUR**). В функциях задается смещение для нового чтения или записи. В нашем примере читается вторая запись от начала (для работы с записями на уровне потоков ввода и вывода).

Чтение второй записи из файла:

```
// Структура для вывода файла и позиционирования
struct Person { // Структура для примеров
char Name[24]; // Фамилия
float Stipen; // Стипендия
int Kurs; // Курс
};
```

...

Исходный файл (**fwrite.out**) распечатаем и получим:

```
Иванов 10.500000 1
Петров 100.500000 2
Сидоров 1000.500000 3
```

...

```
// Позиционирование чтение №2
int Pos = 2, PosNew; // новые указателя читаем запись (Pos - 1) = 1 (2-я т.к.
начнем с нуля )
Person PWork;
pF = fopen( "fwrite.out", "r+b"); // Открытие для чтения
// Перемещение указателя на новую позицию 2 (Pos - 1), подсчет идет с нуля
PosNew = fseek( pF, (Pos - 1)*sizeof(Person), SEEK_SET);
fread( &PWork, sizeof(Person), 1, pF);
printf( "Позиция № 2 записи из файла %s %f %d\n", PWork.Name, PWork.Stipen
, PWork.Kurs );
fclose(pF); // Закрытие файла
```

...

Результат печати:

```
Позиция № 2 записи из файла Петров 100.500000 2
```

...

Запись на место второй записи от начала:

```

// Запись по позиции Pos = 2
pF = fopen( "fwrite.out" , "r+b"); // Открытие двоичного файла для записи и
чтения
Person PNew = {"Попов", 10000.0f , 10 }; // Новая запись
PosNew = fseek( pF, (Pos - 1)*sizeof(Person), SEEK_SET);
fwrite( &PNew, sizeof(Person) , 1, pF);

...

//Новое чтение 2-й записи
PosNew = fseek( pF, (Pos - 1)*sizeof(Person), SEEK_SET);
fread( &PWork, sizeof(Person) , 1, pF);
printf( "Измененная № 2 записи из файла %s %f %d\n", PWork.Name ,
PWork.Stipen , PWork.Kurs );

```

Результат печати:

```
Измененная № 2 записи из файла Попов 10000.000000 10
```

```

...

//Новое чтение 1-й записи
flush( pF ); // Запись буфера на диск
rewind(pF); // указатель в начало файла
fread( &PWork, sizeof(Person) , 1, pF); // Контрольное чтение сначала
printf( " 1-я запись из файла %s %f %d\n", PWork.Name ,
PWork.Stipen , PWork.Kurs );
fclose(pF); // Заккрытие файла

```

Результат печати:

```
1-я запись из файла Иванов 10.500000 1
```

Для низкоуровневого ввода вывода для файла двоичного **write.bin** перемещение указателя и прямое чтение будут выглядеть так:

Исходный двоичный файл предварительно распечатаем:

```

ФИО = Петров Курс = 1 Стипендия = 1000,000000
ФИО = Петров Курс = 2 Стипендия = 2000,000000
ФИО = Петров Курс = 3 Стипендия = 3000,000000
ФИО = Петров Курс = 4 Стипендия = 4000,000000
ФИО = Петров Курс = 5 Стипендия = 5000,000000

```

```
// Чтение по позиции
```

```
long pos1 = 3 , pos2; // Читаем вторую запись
```

```
pFBin = _open( "write.bin", _O_RDWR | _O_BINARY | _O_APPEND );
```

```
pos2 = _lseek( pFBin, sizeof(Student)* (pos1 - 1), SEEK_SET );
```

```
_read( pFBin , &SBuf , sizeof(Student));
```

```

_close( pFBin ); // Заккрытие файла
printf ("ФИО = %s Курс = %d Стипендия = %f \n" , SBuf.Name, SBuf.Num ,
SBuf.Oklad);
...

```

Результат печати:

```

ФИО = Петров Курс = 2 Стипендия = 2000,000000
...

```

Запись в середину и в конец файла:

```

// Изменение записи
pFBin = _open( "write.bin", _O_RDWR | _O_BINARY ); // Нет _O_APPEND
pos2 = _lseek( pFBin, sizeof(Student)*( pos1 - 1), SEEK_SET );
_read( pFBin , &SBuf , sizeof(Student)); // Чтение 3-й записи
pos2 = _lseek( pFBin, sizeof(Student)*( pos1 - 1), SEEK_SET );
strcpy(SBuf.Name, "Изменение"); // Запись на место 3-й записи
_write (pFBin , &SBuf , sizeof(Student));

// Добавление в конец файла
pos2 = _lseek( pFBin, 0L, SEEK_END );
Student S4 = {"APPEND END" , 4 , 15000.0f}; // Новая структурная переменная для
записи
_write (pFBin , &S4 , sizeof(Student)); // Добавление в конец файла
_commit ( pFBin ); // Запись на диск после изменений
_close( pFBin ); // Заккрытие файла

```

Результат чтения файла после всех изменений:

```

ФИО = Петров Курс = 1 Стипендия = 1000,000000
ФИО = Петров Курс = 2 Стипендия = 2000,000000
ФИО = Изменение Курс = 3 Стипендия = 3000,000000
ФИО = Петров Курс = 4 Стипендия = 4000,000000
ФИО = Петров Курс = 5 Стипендия = 5000,000000
ФИО = APPEND END Курс = 4 Стипендия = 15000,000000

```

8.1.18. Перенаправление потоков ввода и вывода

С помощью специальных функций можно стандартный поток перенаправить в другой файл. Для этого используется функция **reopen**. Пример:

```

/// ПЕРЕНАПРАВЛЕНИЕ ПОТОКА
pF = freopen( "out.txt", "w", stdout );
printf( "Позиция персона из файла %s %f %d\n", PWork.Name , PWork.Stipen ,
PWork.Kurs );
fclose(pF); // Заккрытие файла

```

Такую возможность целесообразно использовать также при отладке программ для запоминания вывода и ввода при поиске ошибок.

8.1.19. Файл менеджеры

Для работы с файлами программисты, системные программисты и обычные пользователи могут использовать специальные программы, называемые файл менеджерами (**FileManager**), в частности это: **Total Comander**, **Windows Comander**, **Far Manager** и так далее. Для более подробного знакомства с такими утилитами операционной системы обратитесь к пособию по курсу [6], к разделу № 6. Кроме этого, вы можете воспользоваться справочными системами и информацией Интернет.

Эти системные программные средства позволяют выполнить в удобной форме множество операций с файлами и каталогами: от поиска файлов, их просмотра и до их ручной модификации.

8.1.20. Работа с файлами целиком

При работе с файлами из программы могут понадобиться специальные действия: удаления файлов, копирование, их переименование. Для таких действий в системе ввода вывода Си есть специальные функции: **remove** (удалить), **rename** (переименовать) и другие. Кроме этого можно воспользоваться функцией **system** для выполнения любой доступной команды ОС. Примеры:

```
rename("Test1.txt" , "Test.txt");// Файл переименовать
remove("Test.txt"); // Нужно добавить файл "Test.txt" в текущий каталог
system(" chcp 1251 > nul");
system(" PAUSE");
system( "type fprintf.out" ); // Печатает весь файл
```

8.1.21. Работа со строками и консолью : sscanf, cprintf и sprintf

В заключение теоретической части ЛР обратим внимание на использование специальных функций (**sscanf** и **sprintf**), которые позволяют работать со строками в режиме форматированного ввода и вывода. Кроме того, возможен чисто консольный ввод/вывод (<**conio.h**>): функции **cprintf** и **cputs**. Покажем это на примере:

```
char    Str [80];
char    buf1[80];
char    buf2[80];
int num;

fgets( Str , 80 ,pF ); // ввод из файла в строку

sscanf( Str, "%s%s%d", buf1, buf2 , &num ); // ввод данных из строки Str
sprintf( Str, " ФИО=%s buf2 = %s num =%d", buf1, buf2 , &num ); // вывод данных в
строку Str

// Консольный ввод и вывод
setlocale( LC_ALL, "" ); // Обязательная для консоли руссификация
cprintf( "Консольный -- '%s' '%s' '%d'\n" , buf1, buf2 , num );
_cputs( buf1 );
```

8.2. 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 ветвления (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 нужно сделать в отладчике.

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.

8.2.1. 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 части, там представлены следующие примеры:

1. Работа с текстовыми и бинарными файлами (просмотр текста в файл менеджере) - Раздел 8.1.10
2. Вычисление длины файла (Раздел 8.1.11).
3. Запись и чтение символьного файла (Раздел 8.1.12).
4. Запись и чтение файла построчно случайными данными (Раздел 8.1.13).
5. Работа с функциями **fread** и **fwrite** (Раздел 8.1.14).
6. Форматированный ввод и вывод в файлы (Раздел 8.1.15).
7. Низкоуровневый ввод вывод в файлы (Раздел 8.1.16).
8. Навигация по файлу **fseek** (Раздел 8.1.17).
9. Работа с консольными функциями (Раздел 8.1.21).

Кроме этого ниже представлены примеры, которые могут быть полезными для изучения технологии работы с файлами, в том числе и при выполнении контрольных заданий. Их тоже желательно изучить и проверить.

8.2.2. Формирование нового файла с вычисленными данными (fprintf).

Создаем файл и записываем в него 5 строк:

```
pF = fopen( "fprintf3.out" , "w+"); // Открытие файла для чтения
for (int i = 0 ; i < 5; i++)
    fprintf (pF , "Строка - %d\n" , i + 1);
fclose(pF); // Закрытие файла
```

Файл (**fprintf3.out**) распечатан программой следующего раздела:

```
'Строка' '-' '1'
```

```
'Строка' '-' '2'
'Строка' '-' '3'
'Строка' '-' '4'
'Строка' '-' '5'
```

8.2.3. Чтение и распечатка текстового файла (файл создан и добавлен выше).

Программа чтения и распечатки текстового файла:

```
// Чтение из файла fscanf
pF = fopen( "fprintf3.out", "r+"); // Открытие файла для чтения
for ( int i = 0 ; !feof(pF) ; i++)
{
    char buf1[80];
    char buf2[80];
    int num;
    fscanf( pF , "%s%s%d" , buf1,buf2, &num); // Ввод форматированных данных -
    // разделитель пробел
    if ( !feof(pF) )
        printf( "'%s' '%s' '%d'\n" , buf1, buf2 , num ); // Печать данных из файла
};
fclose(pF); // Закрытие файла
```

Распечатан файл (**fprintf3.out**), сформированный в предыдущем разделе:

```
'Строка' '-' '1'
'Строка' '-' '2'
'Строка' '-' '3'
'Строка' '-' '4'
'Строка' '-' '5'
```

8.2.4. Добавление в файл вычисленными данными (fprintf).

Добавляем в файл еще три строки с новыми параметрами:

```
pF = fopen( "fprintf3.out", "a+"); // Открытие файла для чтения и добавления
for (int i = 0 ; i < 3; i++)
    fprintf (pF , "СТРОКА - %d\n" , 10 + i + 1); // при добавлении ПРОПИСНЫЕ Буквы
fclose(pF); // Закрытие файла
```

Файл (**fprintf3.out**) распечатан программой предыдущего раздела:

```
'Строка' '-' '1'
'Строка' '-' '2'
'Строка' '-' '3'
'Строка' '-' '4'
'Строка' '-' '5'
'СТРОКА' '-' '11'
'СТРОКА' '-' '12'
```

'СТРОКА' '-' '13'

8.2.5. Запись из массива структур в файл с помощью функции

В нашем случае используется функция печати файла (**StudPrintFileWR**), печати структуры (**PrintStudent**) и функция перезаписи массива в файл (**StudMasToFileWR**). Для примера используем структуру **Student**:

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

// ...
//Перезапись массива структур в файл (Файл предварительно очищается!) WR
void StudMasToFileWR( const char * FileName , Student * pMas , int Razm)
{
    FILE * pF;
    pF = fopen( FileName, "w+b"); // Открытие файла для записи
    for ( int i = 0 ; i < Razm; i++)
        fwrite( pMas + i, sizeof(Student) , 1, pF);
    fclose(pF);
};

// Распечатка отдельной структурной переменной студента
void PrintStudent(Student * pS)
{
    printf( "Запись: Имя = %-15s Номер = %2d Стипендия = %8.2lf \n",
           pS->Name , pS->Num, pS->Oklad );
};

// Печать файла для структур типа Student
void StudPrintFileWR( const char * FileName)
{
    FILE * pF;
    pF = fopen( FileName , "rb"); // Открытие файла для чтения
    Student SBuf;
    while (!feof(pF) ) // Проверка конца файла
    {
        fread( &SBuf, sizeof(Student) , 1, pF); // чтение одной записи
        if ( !feof(pF) )
            PrintStudent(&SBuf);
    };
};
```

```

// Заккрытие файла
fclose( pF );
};
...
// Прототипы для WR – fwrite –fread
void StudMasToFileWR( const char * FileName , Student * pMas , int Razm);
void StudPrintFileWR( const char * FileName)ж
...
// Определение размера файла
FILE * pF;
Student * pStudMas;
Student SMas[] ={{"Первый" , 1 , 1000.0}, {"Второй" , 2 , 2000.0}, {"Третий" , 3 ,
3000.0}, {"Четвертый" , 4 , 4000.0}};
...
RazmS = sizeof(SMas)/ sizeof(Student);
// Вызов функции , RazmS тоже д.б. известен
StudMasToFileWR( "BDStud4.bin" , SMas, RazmS); // Массив записываем в файл!!!!
StudPrintFileWR( "BDStud4.bin");// Файл рапечатываем!!!!

```

Результат работы программы:

Запись: Имя = Первый	Номер = 1	С типендия = 1000,00
Запись: Имя = Второй	Номер = 2	Стипендия = 2000,00
Запись: Имя = Третий	Номер = 3	Стипендия = 3000,00
Запись: Имя = Четвертый	Номер = 4	Стипендия = 4000,00

Пример низкоуровневой перезаписи массива в файл приведен в приложении.

8.2.6. Запись из файла в массив (структуры).

В нашем случае используется функция печати файла (**StudPrintMas**), структуры (**PrintStudent**) и функция перезаписи файл в массив (**StudFileToMasWR**). Для примера используем структуру **Student**:

```

// Структура для демонстрации
struct Student {
    char Name[20];
    int Num;
    float Oklad;
};

// Выборка из файла в динамический массив WR
void StudFileToMasWR( const char * FileName , Student ** pMas , int * pRazm)
{
    FILE * pF;
    pF = fopen( FileName , "rb"); // Открытие файла для чтения

```



```

*pRazm = _filelength(pF ->_file) / sizeof(Student);
Student SBuf;
int i = 0;
while (!feof(pF) ) // Проверка конца файла
{
    fread( &SBuf, sizeof(Student) , 1, pF); // чтение одной записи
    if ( !feof(pF) )
    { memcpy( *pMas + i , &SBuf , sizeof(Student)); i++; };
    };
// Заккрытие файла
    fclose( pF );
};
// Распечатка отдельной структурной переменной студента
void PrintStudent(Student * pS)
{
    printf( "Запись: Имя = %-15s Номер = %2d Стипендия = %8.2lf \n",
           pS->Name , pS->Num, pS->Oklad );
};

// Печать массива структур
void StudPrintMas( Student * pMas , int Razm)
{
    for (int i = 0 ; i < Razm ; i++ )
    {
        PrintStudent(pMas + i);
    };
};
//
...
// Прототипы
void StudPrintMas( Student * pMas , int Razm);
void StudFileToMasWR( const char * FileName , Student ** pMas , int * pRazm);
...
// Если файл защищен от записи (после низкоуровневой записи!), то изменим его
атрибут
system("attrib -R BDStud.bin ");
// Временное открытие файла для определение размера файла
FILE * pF;
pF = fopen( "BDStudbin", "r+b"); // Открытие файла для записи
// Вычисление числа записей через функцию _filelength

```

```

int RazmS = _filelength(pFBin)/sizeof(Student); // число записей в файле!
fclose(pF);
...
///// Функция файл в массив структур Student
StudFileToMasWR( "BDStud.bin" , &pStudMas , &RazmS);
StudPrintMas( pStudMas , RazmS);

```

Результат работы программы:

Запись: Имя = Первый	Номер = 1	Стипендия = 1000,00
Запись: Имя = Второй	Номер = 2	Стипендия = 2000,00
Запись: Имя = Третий	Номер = 3	Стипендия = 3000,00
Запись: Имя = Четвертый	Номер = 4	Стипендия = 4000,00

Пример низкоуровневой перезаписи файла в массив приведен в приложении.

8.2.7. Распечатка массива структур.

В нашем случае используется функции печати файла (**StudPrintMas**) и печати одной структуры (**PrintStudent**).

```

// Распечатка отдельной структурной переменной студента
void PrintStudent(Student * pS)
{
    printf( "Запись: Имя = %-15s Номер = %2d Стипендия = %8.2lf \n",
           pS->Name , pS->Num, pS->Oklad );

};

// Функция распечатки массива структур
void StudPrintMas( Student * pMas , int Razm)
{
    for (int i = 0 ; i < Razm ; i++ )
    {
        PrintStudent(pMas + i);
    };
};

// ...
// Функция распечатки массива структур
StudPrintMas( pStudMas , RazmS);

```

Запись: Имя = Первый	Номер = 1	Стипендия = 1000,00
Запись: Имя = Второй	Номер = 2	Стипендия = 2000,00
Запись: Имя = Третий	Номер = 3	Стипендия = 3000,00
Запись: Имя = Четвертый	Номер = 4	Стипендия = 4000,00

8.2.8. Распечатка файла структур.

В нашем случае используется функция печати файла (**StudPrintFileWR**):

```
// Печать файла для структуры Student
void StudPrintFileWR( const char * FileName)
{
    FILE * pF;
    pF = fopen( FileName , "rb"); // Открытие файла для чтения
    Student SBuf;
    while (!feof(pF) ) // Проверка конца файла
    {
        fread( &SBuf, sizeof(Student) , 1, pF); // чтение одной записи
        if ( !feof(pF) )
            PrintStudent(&SBuf);
    };
    // Закрытие файла
    fclose( pF );
};
...
// Прототип функции
void StudPrintFileWR( const char * FileName);
...
// Печать файла по имени файла
StudPrintFileWR( "BDStud4.bin");
```

В нашем случае используется функция печати файла (**StudPrintFileWR**), структуры

Запись: Имя = Первый	Номер = 1	Стипендия = 1000,00
Запись: Имя = Второй	Номер = 2	Стипендия = 2000,00
Запись: Имя = Третий	Номер = 3	Стипендия = 3000,00
Запись: Имя = Четвертый	Номер = 4	Стипендия = 4000,00

8.2.9. Сортировка по строкам в массиве структур.

Сортировка выполняется на основе функции обмена записей (**SwapStudent**):

```
// обмен на основе адресов структурных переменных (возмоает только без динамики!!)
void SwapStudent ( Student * pA , Student * pB )
{
    Student Temp;
    Temp = *pA;
```

```

    *pA = *pB;
    *pB = Temp;
    /* можно и так
        memcpy( Temp , *pA, sizeof(Student) );
        memcpy( *pA , *pB, sizeof(Student) );
        memcpy( *pB , Temp, sizeof(Student) );
    */
};

```

Сортировка выполняется на основе функции обмена записей (**SwapStudent**):

Сортировка в массиве **pStudMas**:

```

    for (int i = 0 ; i < RazmS - 1 ; i++ )
        for (int k = 0 ; k < RazmS - 1 ; k++ )
            // Сортировка разные режимы!!!!!!!!!!!!!!!!!!!!!!
            //      if ( ((Student *)(pStudMas + k))->Num < ((Student *)(pStudMas + k + 1))-
>Num ) // Убывание курс
            //      if ( ((Student *)(pStudMas + k))->Num < ((Student *)(pStudMas + k + 1))-
>Num ) // Убывание курс
            if ( strcmp(((Student *)(pStudMas + k))->Name , ((Student *)(pStudMas + k
+ 1))->Name) > 0 ) // Убывание имя
                { SwapStudent ( pStudMas + k , pStudMas + k+1 ); };
            //
    StudPrintMas( pStudMas , RazmS);

```

Результат сортировки массива:

Запись : Имя = Второй	Номер = 2	Стипендия = 2000,00
Запись : Имя = Первый	Номер = 1	Стипендия = 1000,00
Запись : Имя = Третий	Номер = 3	Стипендия = 3000,00
Запись : Имя = Четвертый	Номер = 4	Стипендия = 4000,00

8.2.10. Сортировка строк в файле по номеру.

Сортировка файла может быть выполнена так (см. предыдущие примеры в них описаны функции для этого примера):

- Взять из файла в массив
- Сортировка массива
- Запись новая в файл
- Распечатка файла

В программе это будет с использованием функций выглядеть так:

```

// СОРТИРОВКА
printf( "ДО СОРТИРОВКИ (по номеру)!\n" ); //

```

```

StudPrintFileWR( "BDStud4.bin");
StudFileToMasWR( "BDStud4.bin" , &pStudMas , &RazmS);
// Сортировка пузырьковая
// Сортировка массива по имени и номеру курса по убыванию/возрастанию
for (int i = 0 ; i < RazmS - 1 ; i++ )
    for (int k = 0 ; k < RazmS - 1 ; k++ )
        if ( ((Student *)(pStudMas + k))->Num < ((Student *)(pStudMas + k + 1))->Num
) // Убывание { SwapStudent ( pStudMas + k , pStudMas + k+1 ); };
// В файл
StudMasToFileWR( "BDStud4.bin" , pStudMas, RazmS);
// Печать файла
printf( "ПОСЛЕ СОРТИРОВКИ!\n" ); //
StudPrintFileWR( "BDStud4.bin");

```

Результат сортировки файла:

ДО СОРТИРОВКИ (по номеру) !

Запись: Имя = Первый	Номер = 1	Стипендия = 1000,00
Запись: Имя = Второй	Номер = 2	Стипендия = 2000,00
Запись: Имя = Третий	Номер = 3	Стипендия = 3000,00
Запись: Имя = Четвертый	Номер = 4	Стипендия = 4000,00

ПОСЛЕ СОРТИРОВКИ!

Запись: Имя = Четвертый	Номер = 4	Стипендия = 4000,00
Запись: Имя = Третий	Номер = 3	Стипендия = 3000,00
Запись: Имя = Второй	Номер = 2	Стипендия = 2000,00
Запись: Имя = Первый	Номер = 1	Стипендия = 1000,00

8.2.11. Поиск и чтение записи по номеру.

Для поиска нужно проконтролировать номер записи (**nFind**):

```

// Поиск и чтение записи по номеру.
StudPrintFileWR( "BDStud.bin");
int nFind = 2 ;
int Pos1;
pF = fopen( "BDStud.bin" , "rb"); // Открытие файла для чтения
// Перемещение указателя
if (filelength(pF->_file)/sizeof(Student) >= nFind )
{
    Pos1 = fseek( pF, (nFind - 1)*sizeof(Student), SEEK_SET); // Перемещение
указателя на нужную запись для чтения
    fread( &SBuf, sizeof(Student) , 1, pF); // чтение одной записи
    PrintStudent(&SBuf);
}
else

```

```
printf( "Ошибка номера записи- %d !\n" , nFind ); // Ошибка номера
```

```
// Закрытие файла
```

```
fclose( pF );
```

Результат работы фрагмента текста (nFind = 2):

```

1 - Запись: Имя = Второй          Номер = 2  Стипендия =
2000,00
2 - Запись: Имя = Первый          Номер = 1  Стипендия =
1000,00
3 - Запись: Имя = Третий          Номер = 3  Стипендия =
3000,00
4 - Запись: Имя = Четвертый       Номер = 4  Стипендия =
4000,00
```

Полученное значение из файла:

```
Запись: Имя = Первый          Номер = 1  Стипендия = 1000,00
```

Результат работы фрагмента текста (nFind = 10):

```

1 - Запись: Имя = Второй          Номер = 2  Стипендия =
2000,00
2 - Запись: Имя = Первый          Номер = 1  Стипендия =
1000,00
3 - Запись: Имя = Третий          Номер = 3  Стипендия =
3000,00
4 - Запись: Имя = Четвертый       Номер = 4  Стипендия =
4000,00
```

```
Ошибка номера записи- 10 !
```

8.2.12. Удаление записи по номеру.

Для удаления одной записи из файла: считываем файл в массив, удаляем запись(номер удаляемой записи считается с 1-цы - **NumDel**), а затем перезаписываем массив в файл.

```

printf( "ДО УДАЛЕНИЯ (по номеру)!\n" ); //
StudPrintFileWR( "BDStud.bin");
Student * pMas;
int RazmF;
int NumDel = 2;
StudFileToMasWR( "BDStud.bin" , &pMas , &RazmF);
///
for ( int i = 0, k = 0 ; i < RazmF ; i++)
{
    if ( (NumDel - 1) != i )
    { memcpy( pMas + k , pMas + i , sizeof(Student)); k++;}
};
///
StudMasToFileWR( "BDStud.bin" , pMas , ( RazmF > NumDel ) ? RazmF - 1 : RazmF );
printf( "ПОСЛЕ УДАЛЕНИЯ (по номеру)!\n" ); //
```

```
StudPrintFileWR( "BDStud.bin");
```

Результат работы фрагмента текста (**NumDel** = 2):

ДО УДАЛЕНИЯ (по номеру) !

Запись: Имя = Второй Номер = 2 Стипендия = 2000,00

Запись: Имя = Первый Номер = 1 Стипендия = 1000,00

Запись: Имя = Третий Номер = 3 Стипендия = 3000,00

Запись: Имя = Четвертый Номер = 4 Стипендия = 4000,00

ПОСЛЕ УДАЛЕНИЯ (по номеру) !

Запись: Имя = Второй Номер = 2 Стипендия = 2000,00

Запись: Имя = Третий Номер = 3 Стипендия = 3000,00

Запись: Имя = Четвертый Номер = 4 Стипендия = 4000,00

8.2.13. Добавление записи по номеру.

Для добавления записи в файл по номеру: считываем файл в массив, добавляем в массив запись (номер добавляемой записи считается с 1-цы - **NumAdd**), а затем перезаписываем массив в файл. Массив создаем по размеру на единицу больше чем размер файла.

```
//Добавление записи по номеру
```

```
printf( "ДО ДОБАВЛЕНИЯ (по номеру)!\n" ); //
```

```
StudPrintFileWR( "BDStud.bin");
```

```
int NumAdd = 2;
```

```
Student SAdd = {"ADDED", 33, 15.00};
```

```
StudFileToMasWR( "BDStud.bin" , &pMas , &RazmF);
```

Student * pMasNew;

```
pMasNew = ( Student * ) calloc( RazmF + 1 , sizeof(Student));
```

```
int p = 0;
```

```
int m =0 ;
```

```
for ( m = 0 ; m < RazmF ; m++ , p++)
```

{

```
if ( (NumAdd - 1) != p )
```

```
{ memcpy( pMasNew + p , pMas + m , sizeof(Student)); }
```

else

```
{ memcpy( pMasNew + p , &SAdd , sizeof(Student)); m--;}
}
```

 $\}:$

```
if ( p == m )
```

```
memcpy( pMasNew + p , &SAdd , sizeof(Student));
```

```
StudMasToFileWR( "BDStud.bin" , pMasNew , RazmF + 1 );
```

```
free ( pMasNew );
```

```
printf( "ПОСЛЕ ДОБАВЛЕНИЯ (по номеру)!\n" ); //
```

```
StudPrintFileWR( "BDStud.bin");
```

Результат работы фрагмента текста (NumAdd= 2):

ДО ДОБАВЛЕНИЯ (по номеру) !

Запись: Имя = Второй Номер = 2 Стипендия = 2000,00

Запись: Имя = Третий Номер = 3 Стипендия = 3000,00

Запись: Имя = Четвертый Номер = 4 Стипендия = 4000,00

ПОСЛЕ ДОБАВЛЕНИЯ (по номеру) !

Запись: Имя = Второй Номер = 2 Стипендия = 2000,00

Запись: Имя = ADDED Номер = 33 Стипендия = 15,00

Запись: Имя = Третий Номер = 3 Стипендия = 3000,00

Запись: Имя = Четвертый Номер = 4 Стипендия = 4000,00

8.2.14. Модификация записи в двоичном файле.

Фрагмент программы для модификации одной записи в существующем двоичном файле структур. Операция выполняется через прямое чтение записи по номеру, ее изменения и запись на тоже место модифицированной записи. Нумерация записей при удалении начинается с единицы.

```

...
// ИЗМЕНЕНИЕ ЗАПИСИ
int nChange = 3;
Student SBuf;
printf( "ДО ИЗМЕНЕНИЯ (по номеру = %d)!\n" , nChange ); //
system("attrib -R BDStud.bin ");
StudPrintFileWR( "BDStud.bin");
pF = fopen( "BDStud.bin" , "r+b"); // Открытие файла для чтения и записи
if (filelength(pF->_file)/sizeof(Student) >= nChange )
{
    Pos1 = fseek( pF, (nChange - 1)*sizeof(Student), SEEK_SET); //Перемещение указателя
на (nChange-1)
    fread( &SBuf, sizeof(Student) , 1, pF); // чтение одной записи
    strcpy(SBuf.Name , "ИЗМЕНЕНИЕ!");
    Pos1 = fseek( pF, (nChange - 1)*sizeof(Student), SEEK_SET); // Перемещение указателя
на (nChange-1)
    fwrite( &SBuf, sizeof(Student) , 1, pF); // запись одной записи
}
else
    printf( "Ошибка номера записи- %d !\n" , nChange ); // Ошибка номера
    fclose( pF );
printf( "ПОСЛЕ ИЗМЕНЕНИЯ (по номеру = %d)!\n" , nChange ); //
StudPrintFileWR( "BDStud.bin");
...

```

Результат работы фрагмента текста (nChange = 3):

ДО ИЗМЕНЕНИЯ (по номеру = 3) !

Запись: Имя = Второй	Номер = 2	Стипендия = 2000,00
Запись: Имя = Первый	Номер = 1	Стипендия = 1000,00
Запись: Имя = Третий	Номер = 3	Стипендия = 3000,00
Запись: Имя = Четвертый	Номер = 4	Стипендия = 4000,00
ПОСЛЕ ИЗМЕНЕНИЯ (по номеру = 3) !		
Запись: Имя = Второй	Номер = 2	Стипендия = 2000,00
Запись: Имя = Первый	Номер = 1	Стипендия = 1000,00
Запись: Имя = ИЗМЕНЕНИЕ !	Номер = 3	Стипендия = 3000,00
Запись: Имя = Четвертый	Номер = 4	Стипендия = 4000,00

8.2.15. Программа сравнения двух файлов.

/0.12. Функция сравнения двух файлов.

```

char FileName1[]="File1.txt";
char FileName2[]="File2.txt";
FILE * pF1;
FILE * pF2;
int flag = 0;
pF1 = fopen( FileName1 , "r"); // Открытие для чтения
pF2 = fopen( FileName2 , "r"); // Открытие для чтения
if ( pF1 != NULL || pF2 != NULL)
{
while (!feof(pF1)&&!feof(pF2))
{
    char ch1 = getc(pF1);
    char ch2 = getc(pF2);
    int fl1 =feof(pF1);
    int fl2 =feof(pF2);
    if ( ch1 != ch2 ) { flag = 1; break ;};
    if ( feof(pF1) != feof(pF2)) { flag = 1; break ;};

};
if( flag == 1) printf( "Файлы %s и %s не равны!\n" , FileName1 , FileName2 );
else printf( "Файлы %s и %s равны!\n" , FileName1 , FileName2 );
fclose(pF1);
fclose(pF2);
};

```

8.2.16. Программа печати списка текстовых файлов из параметров командной строки.

```

// Ввод из файл ов по списку аргументов (gets)
char FName[32] = "";

```

```

int ArgCount = 1;
while ( ArgCount < Argc )
{
strcpy ( FName , Argv[ArgCount]);
///
pF = fopen( FName , "r"); //Открытие текстового файла для чтения
//
printf( "\nФайл - %s\n" , FName);
fgets( line , 80 , pF );
while (!feof(pF)|| strlen(line) !=NULL )
{
    printf( "%s", line);
    line[0] = '\0'; // для обнуления строки при новом чтении из файла
    fgets( line , 80 , pF ); // Чтение строки из файла
};
fclose(pF); // Заккрытие файла
///
ArgCount ++; // Новый файл из списка аргументов командной строки
};

```

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

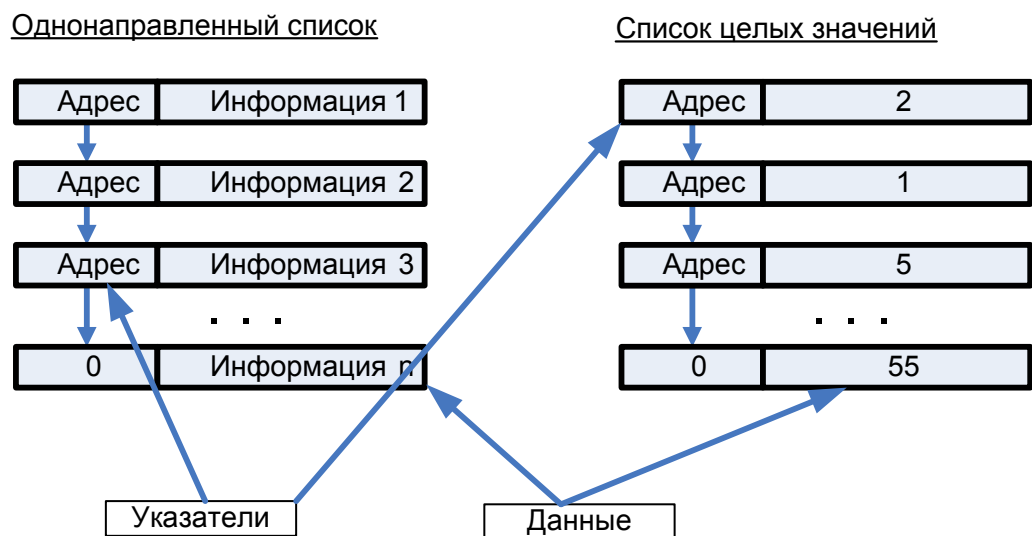
9. Лекция № 9 – Списки. Основные понятия

9.1. Основные понятия

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

9.1.1. Понятие список

Список – это структура данных, в которой данные расположены в определенной последовательности, которая задается с помощью адресации. Для этой цели используются



переменные типа указатель. Каждый фрагмент (элемент списка) должен содержать такой указатель и непосредственно информацию списка. На рисунке представлен список, содержащий целые значения. Более подробно структуру списка рассмотрим ниже. 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
```

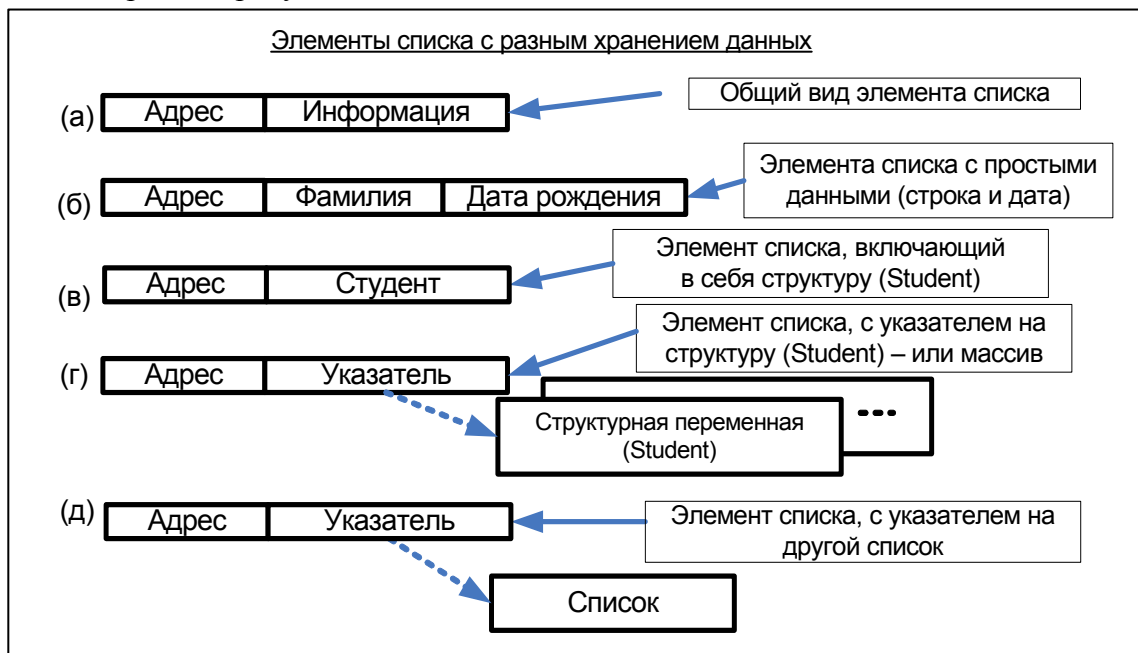
// Значение ListVal = 3

9.1.2. 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.

9.1.3. 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.

9.1.4. 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:

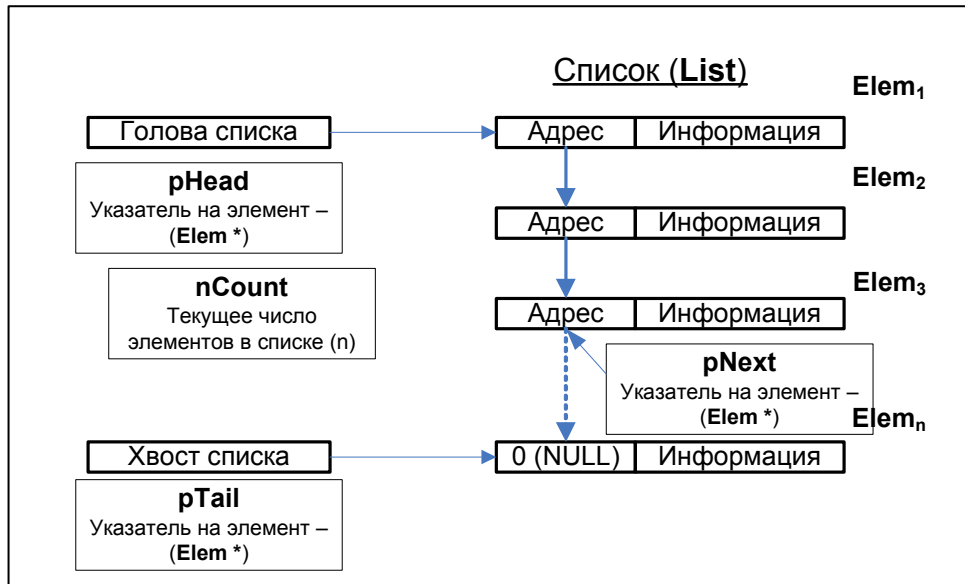
Содержимое ручного списка:

```
Элемент  = 10
Элемент  = 20
Элемент  = 30
```

9.1.4.1. 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 называется 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 инициализация 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
головой списка
while ( pETemp != NULL) // Проверка прохождения последнего элемента списка
{
    printf ("Элемент простого списка FirstList: %d \n" , pETemp->ListVal);
    pETemp = pETemp->pNext; // НАВИГАЦИЯ: Важнейший 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.

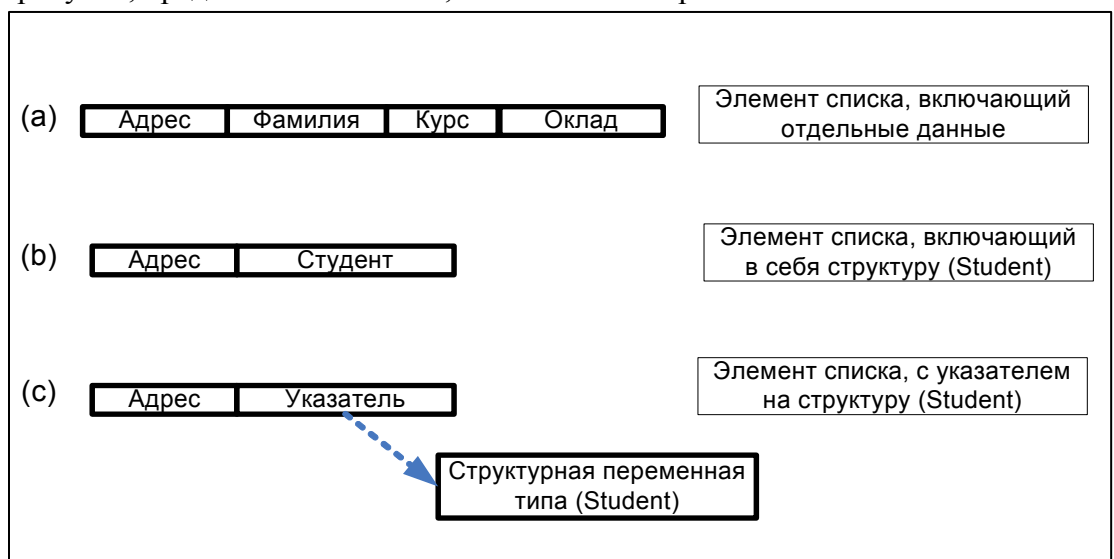
```
pETemp = pETemp->pNext;
```

9.1.5. 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

```

9.1.6. Однонаправленные и двунаправленные списки

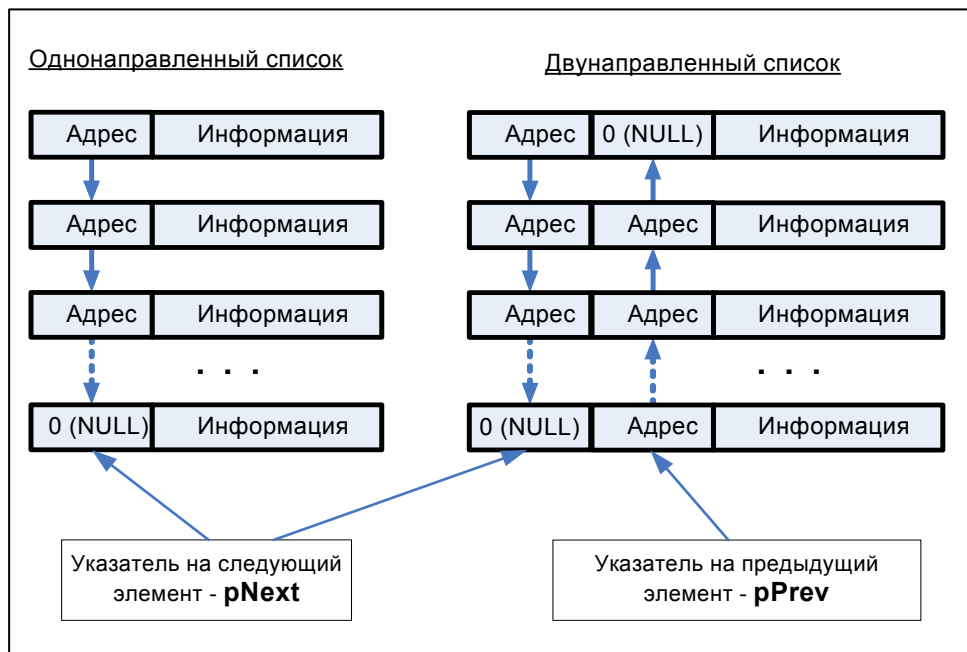
Более удобными и “быстродействующими”, по сравнению с однонаправленными списками, считаются двунаправленные списки. В них навигация по списку может осуществляться как в прямом, так и в обратном направлении. Для этого в структуре каждого элемента списка (**DElem**) предусмотрено два указателя (два адреса):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

9.1.7. 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 структурная 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 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 элемента: 5

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

9.1.8. 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
- 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.

9.1.9. 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

9.1.10. 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
};

```

9.1.11. 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 списка
    while ( pETemp != NULL) // Проверка прохождения последнего элемента списка
    {
        printf ("Элемент простого списка в функции: %d \n" , pETemp->ListVal);
        pETemp = pETemp->pNext; // НАВИГАЦИЯ: Важнейший оператор для работы со списком
    };
};

////Описание и инициализация простого списка (на основе элементарного 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

```

9.1.12. 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

```

9.1.13. 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

9.1.14. 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

9.1.15. Удаление из хвоста списка (список с указателями)

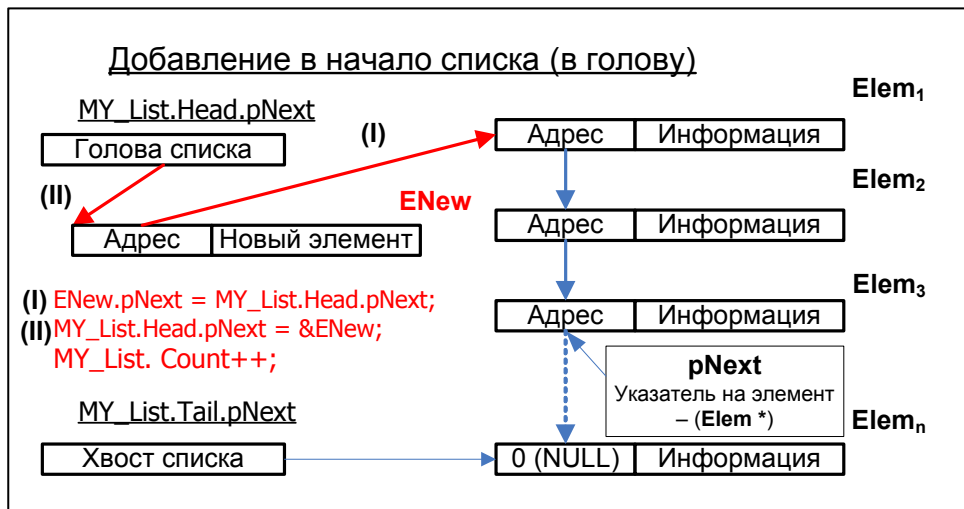
```
// УДАЛЕНИЕ из хвоста (В этом списке сложнее всего)
// Поиск предпоследнего для нового хвоста
Elem * PrevTailElem = SecondList.pHead;
Elem * ETempCurr = SecondList.pHead;
while ( ETempCurr->pNext != NULL)
{
  PrevTailElem = ETempCurr;
  ETempCurr = ETempCurr->pNext;
};
// Само удаление !!!!!!!!!!!!!!!!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

9.1.16. 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
};
```

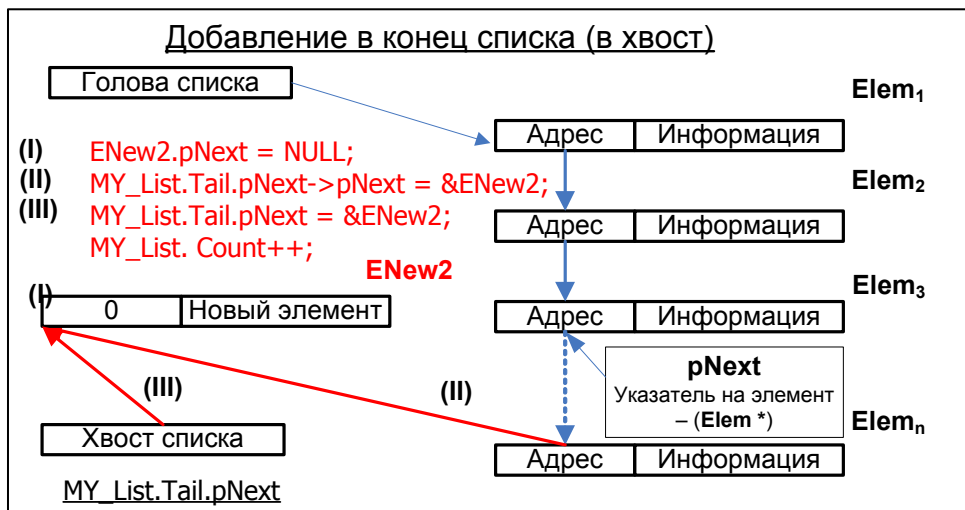
Результат распечатки:

Содержимое ручного списка после добавления в голову:

```
Элемент = 5
Элемент = 1
Элемент = 2
Элемент = 3
```

9.1.17. 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 в 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 элемента.

```
// 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 предыдущем случае)
```

Результат распечатки:

Содержимое ручного списка после добавления в хвост:

```
Элемент = 5
Элемент = 1
```

```

Элемент  = 2
Элемент  = 3
Элемент  = 10

```

9.1.18. 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. Отметим, 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:

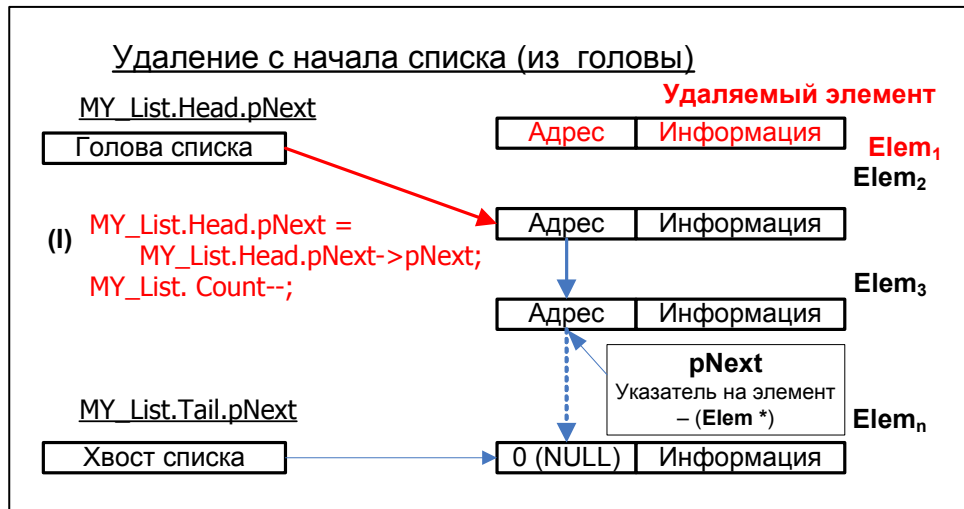
```

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

```

9.1.19. 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

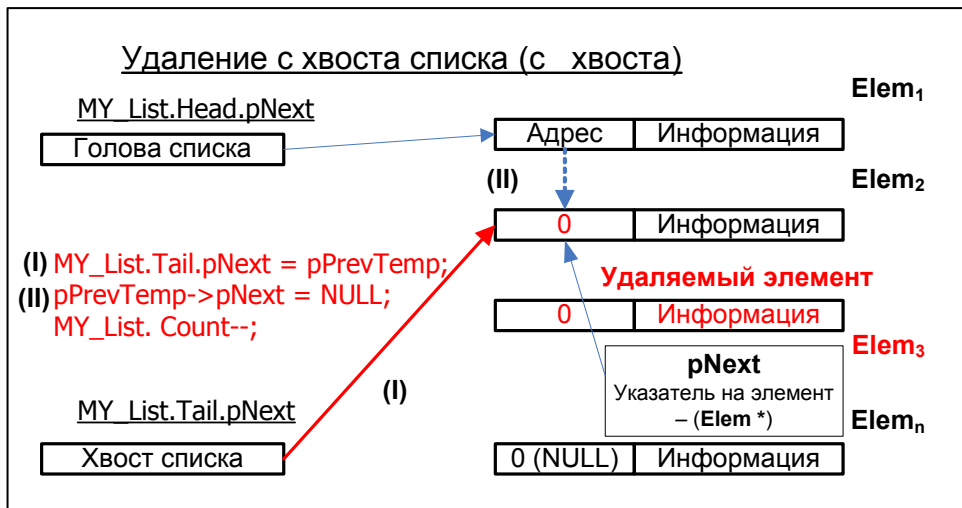
Элемент = 2

Элемент = 3

Элемент = 10

9.1.20. 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
    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.

9.1.21. 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.).

9.1.22. 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 списком.

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

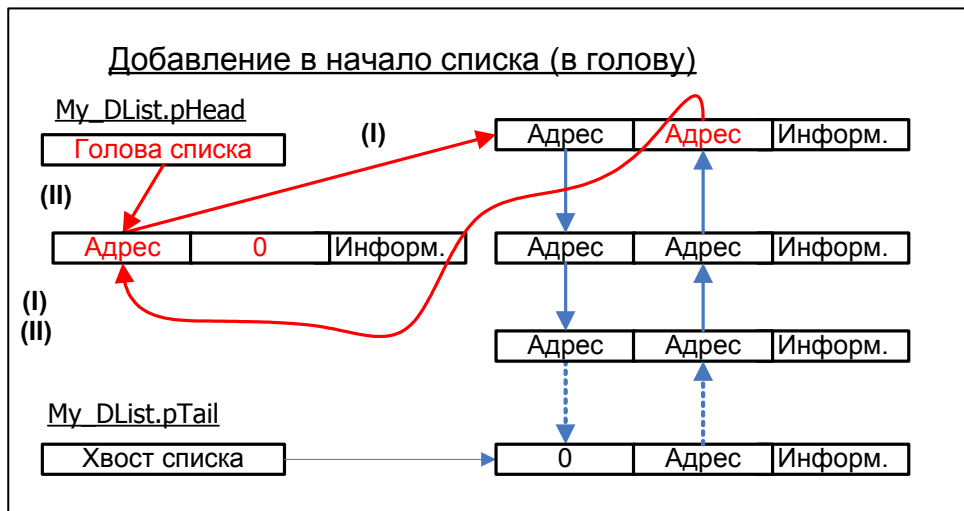
Элемент списка: - 10

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.

9.1.23. 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.

9.2. 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.

9.2.1. 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.

9.2.2. 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
};
```

9.2.3. 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 предыдущий перед 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
    };
};

```

9.2.4. Функции вставки и удаления по номеру в однонаправленном списке

Ниже приведены универсальные функции для работы с однонаправленными списками. Эти функции могут использоваться 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 однонаправленного 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 DellListNum( 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 однонаправленного 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

9.2.5. 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 по 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;}// ХВЕСТ
    // 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

9.2.6. 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 элементов - 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

9.2.7. 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;
};
```

// Структура 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 двунаправленного 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 двунаправленного 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 двунаправленного 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 двунаправленного 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 двунаправленного 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 двунаправленного 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 );

```

9.2.8. 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 хвост
DListPrint ( L1 ); // печать списка
Результат:

```

Добавление в хвост и в голову:

Содержимое списка DList:

Элемент = 3

Элемент = 2

Элемент = 1

Элемент = 55

...

9.2.9. 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

9.2.10. 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
```

9.2.11. 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

```

9.2.12. 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 VList {
    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
VList 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.

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 C, 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 – входит 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 “С++ Базовый курс”: Пер. с англ.- М., Издательский дом “Вильямс”, 2011 г. – 672с
13. Г. Шилдт “С++ Руководство для начинающих” : Пер. с англ. - М., Издательский дом “Вильямс”, 2005 г. – 672с

14. Г. Шилдт “Полный справочник по С++”: Пер. с англ.- М., Издательский дом “Вильямс”, 2006 г. – 800с
15. Бьерн Страуструп "Язык программирования С++"- М., Бином, 2010 г.

