

Большаков С.А. , к.т.н., доц. Каф. ИУ5

**Методические указания для выполнения ДЗ/КЛР
по дисциплине Основы программирования
кафедры ИУ5 (ГУИМЦ)**

Предисловие

Данные методические указания предназначены для студентов кафедры ИУ5 (СУЦ) обучающихся на 3-м семестре и изучающих дисциплину “ Основы программирования” - ГУИМЦ. Для выполнения лабораторных работ по курсу необходим определенный набор базовых знаний и умений, но, к сожалению, не все студенты ими обладают, хотя я надеюсь, что для большинства студентов в этом пособии не будет много новых сведений. С другой стороны я надеюсь, что части студентов данная информация будет, несомненно, полезна и они с успехом будут использовать данный материал для выполнения и защиты лабораторных работ курса.

В данном пособии я буду излагать только самые важные и необходимые материалы разделов, а для детального знакомства с предметом читайте рекомендованную литературу, ориентируясь на те страницы, которые указаны в ссылках.

В отдельных случаях, например справок о программах, я не буду переводить пояснительную информацию, рассчитывая на то, что студенты 2-го курса смогут самостоятельно перевести и понять текст на английском языке в предметной области программирования. Те, кто будут испытывать затруднения в этом, смогут обратиться к рекомендованной литературе, а, в лучшем случае, потренироваться в переводе.

Для текста программ и документальных справок о них, полученных автоматически, с целью более легкого ориентирования в материале я следующим образом буду цветом фона текста выделять его фрагменты:

Работа в режиме командной строки (фон серый, а текст черный):

```
> DIR
```

Вставка текста программ (светло - голубой) будет представляться либо так:

```
For (int i = 0 ; i <= MAX; i++)  
{  
Sum = Sum + i;  
}
```

Или как в системе программирования MS VS:

```
int num;  
fscanf( pF , "%s%s%d" , buf1,buf2, &num); //Ввод форматированных данных - разд.пробел
```

Вставка справок и подсказок, полученных автоматически (желтый) или из документации будет представляться так:

```
while ( expression )  
statement
```

Формализованные описания языка и синтаксические правила будем записывать зеленым цветом:

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

Если в тексте встречается слово, которое подчеркнуто, то это означает, что дается определение важного понятия и это понятие, скорее всего, встречается в данном тексте первый раз. Кроме того, подчеркивание в тексте используется для вытеления терминов, на которые в данный момент нужно обратить особое внимание. Например:

Программа – это упорядоченная совокупность операторов ... или

Мы рассмотрим в этом разделе переменные типа указатель ...

Фрагменты исходного текста, включаемые в программы, я ,будут выделяться цветом шрифта представления программы в MS VS. Например, вставка заголовочного файла:

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

Вывод результата в консольное окно, который формируется программой, буду помечать коричневым цветом и устанавливать непропорциональный шрифт **Courier New** или **Татона**. Например:

Так как данные разрабатывается оперативно на основе пожелания студентов и находится сейчас в стадии проработки, то некоторые его разделы могут быть еще не написаны. Они появятся в нужный момент, когда студенты должны приступить к конкретной лабораторной работе. По мере написания, я на сайте буду обновлять данный документ, о чем буду информировать студентов.

Содержание

Предисловие	2
Содержание	4
1. Общие требования к ЛР по курсу ОП	9
2. Линейные вычисления в программировании.....	11
3. Циклические программы	18
4. Массивы и указатели в программах	25
5. Использование переменных типа строка	32
6. Функции и процедуры в программах	41
7. Структуры данных в программах	52
8. Хранение данных и использование файлов	61
9. Списковые структуры данных	75
10. Требования к КЛР/ДЗ и варианты	92
11. Работа в режиме командной строки	97
12. Технология создания исполнимых программ	101
13. Формальное описание синтаксиса в БНФ	104
14. Работа с интегрированными файловыми менеджерами	108
15. Разработка блок-схем программ	111
16. Коды их назначение и виды	123
17. Литература	132
Приложение 1 Блок-схемы	133
Приложение 2 . Примеры программной реализации КЛР/ДЗ	135

Детальное содержание

Предисловие	2
Содержание	4
1. Общие требования к ЛР по курсу ОП	9
1.1. Подготовка к выполнению ЛР по ОП.....	9
1.2. Выполнение ЛР по ОП.....	9
1.3. Оформление отчета по ЛР по ОП	9
1.4. Контрольные вопросы по ЛР по ОП.....	9
1.5. Защита ЛР по ОП.....	9
1.6. Дополнительные требования к ЛР	10
1.7. Сроки выполнения ЛР по ОП.....	10
1.8. Отработка ЛР по ОП	10
1.9. Комплексная ЛР по ОП и ее защита	10
2. Линейные вычисления в программировании.....	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.	11
2.5. Русификация проекта консольного ввода и вывода.	12
2.6. Программы и программирование.....	13
2.7. Переменные	13
2.8. Константы	14
2.9. Операторы и составные операторы	15
2.10. Форматированный ввод-вывод.	15
2.11. Отладка программ.	16
3. Циклические программы	18
3.1. Разветвляющиеся вычислительные процессы.	18
3.2. Безусловный переход (goto)	18
3.3. Ветвление и операторы ветвления (if - else)	19
3.4. Переключатели (switch - case)	20
3.5. Циклы (for, while, do)	20
3.6. Модули	22
3.7. Понятие о блок-схемах	23
3.8. Библиотеки функций.....	24
4. Массивы и указатели в программах	25
4.1. Массивы	25
4.2. Указатели	26
4.3. Динамическая память.....	27
4.4. Отладка программ.	28
4.5. Модули	28
4.6. Библиотеки функций.....	29
4.7. Библиотеки функций и шаблонов классов: RTL, STL, MFC, ATL	30
5. Использование переменных типа строка	32
5.1. Строки в СИ.....	32
5.2. Описание строк и инициализация строк	32
5.3. Основные операции со строками	33

5.4. Ввод и вывод строк	34
5.5. Манипуляция со строками и в строке.....	35
5.6. Преобразование к нижнему или верхнему регистру	36
5.7. Дублирование динамических строк.....	36
5.8. Выделение подстрок на множестве разделителей.....	36
5.9. Преобразование данных в строку и обратно.....	37
5.10. Сортировка строк	37
5.11. Динамические строки.....	38
5.12. Библиотеки функций и классы для строк.....	40
6. Функции и процедуры в программах	41
6.1. Функции – основа процедурного программирования.....	41
6.2. Понятие функции	41
6.3. Понятия, связанные с функциями в программировании	42
6.4. Описания и определения функций.....	43
6.5. Прототипы функций.....	44
6.6. Вызовы функций и возврат значений функции	44
6.7. Переменные в функциях	46
6.8. Параметр массив в функции.....	46
6.9. Размещение функций	48
6.10. Рекурсивные функции.....	48
6.11. Макросы и переменные этапа компиляции.....	48
6.12. Параметры главной функции main	50
6.13. Inline функции.....	50
6.14. Указатели на функции.....	51
7. Структуры данных в программах	52
7.1. Проблемы хранения и обработки данных	52
7.2. Структура - элемент хранения разнородных данных - struct.....	52
7.3. Инициализация структурных переменных	53
7.4. Работа с полями структуры через структуру и указатель на структуру	53
7.5. Многоуровневая квалификация в структурах.....	54
7.6. Указатели на структуры и на динамические структуры	54
7.7. Передача структур в функцию	55
7.8. Передача указателя на структуру в функции.....	55
7.9. Массивы структур	56
7.10. Вложенные структуры	56
7.11. Размер и размещение структур в ОП.....	57
7.12. Динамической структуры	57
7.13. Создание и удаление динамической структуры со строками.....	58
7.14. Структуры со ссылками на себя.....	59
7.15. Перечисления - enum.....	59
7.16. Союзы – union - объединения.....	59
8. Хранение данных и использование файлов	61
8.1. Данные в программах.....	61
8.2. Понятие файла, его определения и разновидности	61
8.3. Операционная система, потоки и файлы.....	62
8.4. Имена и расширения файлов	63
8.5. Открытие и закрытие файлов	64
8.6. Библиотеки и заголовочные файлы	65
8.7. Основные операции и функции для работы с файлами.....	65
8.8. Уровни ввода/вывода и типы файлов	66

8.9. Описание файла в программе. Структура FILE.....	66
8.10. Текстовые и бинарные (двоичные) файлы	67
8.11. Проверка конца файла и указатель чтения файла	68
8.12. Работа с текстовым/двоичным файлом с байтами - символами.....	68
8.13. Работа с текстовым файлом построчно	69
8.14. Двоичные/двоичные файлы и функции fread и fwrite.	70
8.15. Форматированный ввод и вывод в файлы.....	70
8.16. Низкоуровневый ввод и вывод в СИ	71
8.17. Навигация по файлу fseek, lseek.....	72
8.18. Перенаправление потоков ввода и вывода.....	73
8.19. Файл менеджеры	74
8.20. Работа с файлами целиком	74
8.21. Работа со строками и консолью : sscanf, sprintf и printf	74
9. Списковые структуры данных	75
9.1. Понятие список.....	75
9.2. Особенности структур списков и их назначение.....	75
9.3. Особенности списков по сравнению с массивами	76
9.4. Простейший список, созданный вручную.....	76
9.5. Структуры списков и связь элементов списков.....	77
9.6. Структура элемента списка с вложенными и внешними данными	79
9.7. Однонаправленные и двунаправленные списки	79
9.8. Статические и динамические списки.....	80
9.9. Операции для работы со списком	81
9.10. Ручная работа с однонаправленным списком.....	81
9.11. Добавление в голову списка	82
9.12. Добавление в конец списка (в хвост).....	83
9.13. Функция распечатки однонаправленного списка	84
9.14. Удаление из головы списка	84
9.15. Удаление из хвоста списка	85
9.16. Очистка статического списка	86
9.17. Простая ручная работа с динамическим списком	87
9.18. Простая ручная работа с двунаправленным списком	89
10. Требования к КЛР/ДЗ и варианты	92
10.1. Цель КЛР/ДЗ (ЛР №10).....	92
10.2. Требования к заданию на ЛР №10	92
10.3. Особенности описания разделения заданий по уровням ЛР №10	92
10.4. Функции и структуры в домашнем задании	92
10.5. Варианты для выполнения ЛР.....	93
10.6. Порядок выполнения работы (Уровень А и В).....	94
11. Работа в режиме командной строки	97
11.1. Режим командной строки и его назначение	97
11.2. Разновидности командных интерпретаторов.....	97
11.3. Запуск и завершение работы режима командной строки	98
11.4. Запуск команд и программ в режиме командной строки.....	98
11.5. Получение справок о командах в режиме командной строки	99
12. Технология создания исполнимых программ	101
12.1. Модули представления программ	101
12.2. Компоненты и стадии обработки программ	102
12.3. Создание консольного проекта в VS 2005 (DZ_XXXXX_XDD).....	102

12.4. Обеспечить русификацию консольного ввода и вывода.	103
13. Формальное описание синтаксиса в БНФ	104
13.1. Назначение и состав языка БНФ	104
13.2. Правила, нетерминальные переменные и метасимволы	105
13.3. Примеры описания на БНФ	106
14. Работа с интегрированными файловыми менеджерами	108
14.1. Dos Navigator.....	109
14.2. Far manager.....	109
14.3. Windows/Total Commander	110
15. Разработка блок-схем программ	111
15.1. Назначение блок-схем программ	111
15.2. Элементы блок-схем программ.....	112
15.3. Примеры блок-схем программ	117
15.4. Оформление блок-схемы программы	122
15.5. Блок-схемы и описания данных	123
16. Коды их назначение и виды	123
16.1. ASCII	124
16.2. Кодировка ANSI	125
16.3. Русификаторы.....	126
16.4. Перекодировка символов.....	127
16.5. SCAN – коды.....	128
16.6. Кодировка UNICODE.....	130
16.7. Программы для получения списка кодов	130
17. Литература	132
Приложение 1 Блок-схемы.....	133
Приложение 2 . Примеры программной реализации КЛР/ДЗ	135
17.1. Главный модуль File_P3.cpp.....	136
17.2. Заголовочный файл проекта File_P3.h.....	146
17.3. Результаты работы примера (текст).....	154
17.4. Шаблон отчета по ЛР10	158

1. Общие требования к ЛР по курсу ОП

Выполнение цикла ЛР по ПКШ является обязательным для получения зачета и экзамена по курсу. В перечень ЛР по курсу входят: основные ЛР (1-8) и комплексная ЛР. Выполнение основных ЛР и комплексной ЛР является обязательным. В рамках отдельных ЛР могут быть выделены дополнительные требования, самостоятельное выполнение которых позволяет студенту рассчитывать на отличную отметку по результатам семестра. Окончательная отметка по дисциплине определяется: результатами выполнения ЛР, результатами рейтингов и контрольных работ, ответами на экзамене.

1.1. Подготовка к выполнению ЛР по ОП

Студенты обязаны предварительно подготовиться к выполнению ЛР до начала ЛР в дисплейном зале. Для этого необходимо с сайта дисциплины скачать методические указания по ЛР и шаблон отчета по данной ЛР. Прочитать теоретический раздел ЛР (“Основные понятия”). Познакомиться с порядком выполнения ЛР. Преподаватель, ведущий ЛР, имеет право не допускать к выполнению ЛР студентов, которые не подготовились к ее выполнению. Желательно подготовить и заполнить для себя шаблон отчета по ЛР.

1.2. Выполнение ЛР по ОП

Выполнение ЛР основано на предложенном порядке по шагам. Для более глубокого понимания отдельных шагов, после создания проекта, можно скопировать примеры их теоретического раздела и проверить их в своем проекте. Обязательным является использование отладчика при выполнении и защите ЛР. После завершения ЛР студент должен продемонстрировать работающую программу преподавателю, о чем тот обязан сделать отметку в журнале ЛР. При демонстрации программы преподаватель может задать вопросы по ее построению и отладки, а также контрольные вопросы ЛР.

1.3. Оформление отчета по ЛР по ОП

После завершения выполнения ЛР, а еще лучше в процессе ее выполнения в соответствии с требованиями МУ данной ЛР должен быть оформлен отчет по ЛР, который должен быть распечатан. Отчет должен быть представлен для защиты ЛР.

1.4. Контрольные вопросы по ЛР по ОП

В процессе выполнения ЛР (!) студенты должны научиться отвечать на контрольные вопросы по данной ЛР. Ответы могут быть получены: из теоретической части работы, экспериментально в процессе отладки программы и из литературы (документации).

1.5. Защита ЛР по ОП

Защита ЛР может быть выполнена на следующем занятии в аудитории или в специальное время, назначенное преподавателем, проводящим ЛР по курсу. Для допуска к защите в журнале ЛР должна стоять отметка об успешной демонстрации программы. Для защиты должен быть представлен правильно оформленный отчет и устные ответы на контрольные вопросы данной ЛР. При защите нужно быть готовым выполнить демонстрацию программы, если по каким-либо причинам она не была продемонстрирована в дисплейном зале.

1.6. Дополнительные требования к ЛР

Если работа была выполнена с дополнительными требованиями, то на титульном листе отчета должно быть это отмечено, должна быть поставлена отметка в журнале ЛР. Самостоятельное выполнение дополнительных требований способствует повышению итоговой отметки на экзамене. Если самостоятельно выполнены все дополнительные ЛР (9-11), то студент вправе рассчитывать на отличную отметку, при своевременном выполнении других заданий по курсу.

1.7. Сроки выполнения ЛР по ОП

В среднем каждую неделю, начиная со второй, студент должен выполнить и защитить одну ЛР по курсу ПКШ.

1.8. Отработка ЛР по ОП

При невыполнении ЛР в срок по уважительным причинам, студенты могут отработать ЛР в течении семестра в рамках других занятий в дисплейных залах по дисциплине ПКШ. При многократном невыполнении в срок ЛР, включая и их защиту, отметки на экзамене будут снижаться.

1.9. Комплексная ЛР по ОП и ее защита

Комплексная лабораторная работа (КЛР) выполняется самостоятельно по индивидуальному заданию. Ее выполнение включает: разработку системы классов (контейнер и элементный класс), разработку программы тестового примера, использующей созданную систему классов. Разработку комплекта документации на систему классов. Проведение приемно-сдаточных испытаний программного продукта – системы классов. Работа выполняется индивидуально под руководством преподавателя, проводящего ЛР. ЛР 12-15 заключаются в разработке и проверке документации по КЛР. Разработка программ КЛР выполняется студентами также в рамках часов самостоятельной работы. Завершается выполнение КЛР проведением приемно-сдаточных испытаний при предоставлении полного комплекта документации на программу.

2. Линейные вычисления в программировании

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 *.c или *.cpp) и заголовочные (файлы имеют расширения *.h или *.hpp). 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.

2.4. Создание консольного программного проекта в СИ 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. Осиальные галочки должны быть выключены.
- Далее необходимо нажать кнопку “Finish”. Новый проект будет создан.
- Необходимо убрать из главных моделей проекта (LAB1_XDD.CPP и LAB1_XDD.H) все лишнее. Этого: в файле LAB1_XDD.H (у нас в примере LAB1_15.H) уберем все, а в файле LAB1_XDD.CPP (LAB1_15.CPP) оставим только следующий текст:


```
#include "lab1_15.h"
#include <process.h>
#include <stdio.h>
void main(void)
{ ... }
```
- Для контроля правильности создания пустого проекта, нажмем клавишу “F7” для проверки создания программы (build) и “F5” для проверки ее выполнения (run/debug). Все перечисленные действия должны быть выполнены безошибочно.

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 . . .

Обратите внимание на то, что при другом способе локализации (setlocale(0,"rus");) не все 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 классов и шаблонов выполнять в заголовочном файле LAB1_XDD.H (у нас в примере LAB1_15.H). Описания объектов и вызов методов в основном файле: LAB1_XDD.CPP (LAB1_15.CPP).

Далее будет представлено несколько разделов, понятия которых будут разъяснены на лекции, но 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 S_i 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++, 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
- Разработка формализованных описаний алгоритма и логического проекта (блок-схемы и диаграммы классов, диаграммы объектов).
- Написание программы на языке программирования
- Отладка программы, включающая шаги (компиляции, редактирования связей, создание исполнимого модуля, пошагового исполнения программы с контролем результатов).
- Оформление документации на программный продукт.

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
string s; // переменная типа строка из библиотеки STL
```

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,
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
```

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
```

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 S_i - любые операторы и директивы описания языка C++.

Примеры составного оператора:

```
{ 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 () ? :).

2.10. Форматированный ввод-вывод.

Функция **printf** (вывод по формату), является функцией вывода в языке СИ. Первый аргумент функции - строка в двойных кавычках, которая печатается в том виде, в котором указана в функции. Пример:

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

Пример сообщения

Однако функция **printf("Длина равна %4dcm",1);** выведет на печать уже следующее сообщение:

Длина равна 1cm

Таким образом, запись "%4d" означает: впечатать заданную вторым аргументом величину в десятичной (d) форме в следующие 4 позиции. Кроме десятичного спецификатора формата вывода могут быть использованы следующие:

о-преобразование в восьмеричный формат;

2024 год 2 курс 3-й семестр Большаков С.А. ОП ГУИМЦ (УЦ5-31Б,УЦ5-32Б)
 х-преобразование в шестнадцатеричный формат;
 n-преобразование в беззнаковый десятичный формат;
 s-печать в строковом формате;
 c-печать в символьном виде.

Пусть р-указатель на строку "Первое значение", а переменная х- содержит число 5.813, 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 - под дробную часть. Это справедливо для всех типов данных. Спецификатор f используется для вывода данных типа float (число с плавающей точкой). Строковая переменная в функции printf может содержать управляющие символы:

\n-новая строка;
 \f-новая страница;
 \t-табуляция;
 \b-стереть предыдущий символ и т.д.

В языке СИ для ввода данных используются различные функции, в том числе getch и scanf. `c=getch();` Здесь с - переменная типа char или unsigned, ей будет присвоено значение, введенное с клавиатуры. Пусть переменные a, letter и n объявлены как int a,n; char letter; входная строка имеет вид

3162 y 47,

тогда функция `scanf("%4d%c%2d",&a, &letter,&n);` присвоит переменным 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.

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

3. Циклические программы

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) // проверка условия
{ iMas[i] = 0; Flag = false;} // Старшим элементам массива присвоено значение
else
```

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 [5] 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 значения 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");
};
```

Другие примеры использования операторов цикла.

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. 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].

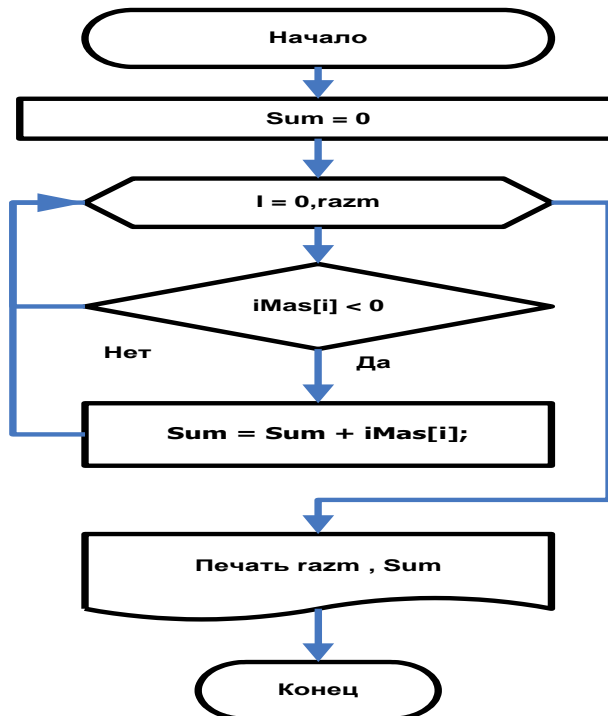
Для объединения модулей функционально и по данным в C++ используются прототипы функций и описание внешних переменных. Покажем это на простом примере:

```
// 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.

3.7. 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.



3.8. 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. Массивы и указатели в программах

4.1. 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, познакомиться 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 на место [2] значения
3 (2+1)
l = PtrMas[l] ; // l = 3
```

4.2. 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; // именование - в указатель записывается адрес
i = *pInt; // разыменование – берем значение переменной по указателю (j)
ppInt = &pInt;
k = **ppInt; // двойное разыменование 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).

4.3. 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 'ktvtynjd 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.

4.4. 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:

- 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!

4.5. 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 междомульных 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.

4.6. 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

4.7. 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 системе 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 делать 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.

5. Использование переменных типа строка

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

5.1. 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.

5.2. 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
```

Последним содержательным символом строки должен быть нулевой символ (отмечу не цифра '0', а символ '\0'), поэтому максимальная длина строки, которую можно разместить в данном массиве равна 9-ти. Одно место теряется. Строка с такими свойствами называется “нультерминатед” (Null Terminated – завершенная нулем). Такое соглашение о конце строки позволяет работать со строками, не указывая явно ее длину. Во многих программах, системах для работы с данными (Базах Данных - БД) используются нультерминатед строки. Поэтому используются и такие соглашения в СИ. Для заполнения массива символов (строки) можно использовать обычные операторы присваивания. Размер массивов в СИ фиксирован, поэтому максимальный размер должен быть определен заранее. Каждый элемент массива заполняется отдельно:

```
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] = 'р';
```

```
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 элемента – 1 символ
```

Когда исходный размер массива не так существенен, 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**.

5.3. 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.

5.4. 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'
```

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.

5.5. 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

5.6. 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!

5.7. 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!!

5.8. 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 парсингом (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:
```

2024 год 2 курс 3-й семестр Большаков

```

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

```

5.9. 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'

```

5.10. 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

2024 год 2 курс 3-й семестр Большаков С.А. ОП ГУИМЦ (УЦ5-31Б,УЦ5-32Б)
 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
```

5.11. 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

2024 год 2 курс 3-й семестр Большаков С.А. ОП ГУИМЦ (УЦ5-31Б,УЦ5-32Б)
 новый большой 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 втроек донны 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 !

5.12. 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
```

6. Функции и процедуры в программах

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

6.1. Функции – основа процедурного программирования.

Возможно, термин функция введенный в язык программирования СИ несколько сбивает с толку студентов, которые усиленно изучали и изучают математику. В программировании под функцией понимается отдельно записанный фрагмент текста программы, который можно вызывать многократно, задавая разные параметры. Кроме того, для каждой функции, в зависимости от ее назначения мы можем присвоить ей имя (или название), соответствующее ее назначению. Например: **PrintArray** (печать массива) или **SortArray** (сортировка массива), что во-первых легче запоминается, а во-вторых, делает программу более наглядной и читаемой (обозримой). Последний аспект относится к понятиям абстракции действий (функций или процедур). Это позволяет, в свою очередь, абстрагироваться (отвлекаться – не учитывать) от деталей внутреннего устройства функции.

Абстракция функций или процедур является основой концепции программирования, которая называется также концепцией процедурно – ориентированного программирования (ПОП). Исторически сложилось так, что ПОП была ранее разработана и в некоторой степени повторяла работу вычислителя – компьютера, основанного на модели Машины Тьюринга. Отметим также, что концепцию ПОП 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 использованием функций.

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

6.2. 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, расположенных в разных частях последовательности операторов в программе. Ниже эти повторы помечены красным:

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

Эти группы операторов могут быть и большими, и, кроме того, выполняться в разных условиях с разными переменными и исходными данными. Возникает естественное желание (кстати, как в математике новое обозначение) дать им отдельное имя, однократно разместить их в отдельном месте ($S_\Phi = S_{i+1}, S_{i+2}$) и обращаться к ним при необходимости. Если иметь специальный оператор вызова, с возможностью передачи параметров ($S_{\Phi 1}$ –

$\langle S_1, S_{\text{вф}1}, S_4, \dots, S_i, S_{\text{вф}1} \dots S_{\text{вф}1}, S_k, \dots \rangle \langle S_{\Phi 1} \dots \rangle$

Размер программы при этом значительно сокращается, она становится более наглядной, но самое главное отлаживать теперь необходимо не все группы операторов (S_{i+1} , S_{i+2}), а только одну группу ($S_{\Phi 1}$), которая называется **функцией** (или процедурой). В разных языках существуют и другие названия: 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).
- Использование функций повышают уровень структурированности программы и облегчают её проектирование.
- Использование функций облегчает чтение и понимание программы, ускоряет поиск и исправление ошибок.

Следствием процедурного подхода является предписание на 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.

6.3. 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
- Формальные параметры функции
- Вызов функции
- Фактические параметры функции
- Тело функции

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

```
// Описание-определение функции
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 задает возвращаемое функцией значение. В главной программе **main** выполнено (в исходном модуле - **first.cpp**) обращение (вызов) к этой функции непосредственно в качестве параметра функции **printf** при печати результата:

```
#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 . . .

6.4. 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) при первом вызове этой функции. При описании функции задаются:

- Название функции, уникальное имя в пределах всей программы.
- Тип возврата функции.
- Список формальных параметров функции
- Тело функции – составной оператор

Название функции это уникальное имя в программе, которое однозначно определяет действия, выполняемые данной функцией. Могут быть хорошие и плохие названия. Например, хорошие названия:

PrintArray, FindKey, PoiskMaximum и др.

Плохие названия функций:

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. 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. Если спецификатор возврата отсутствует, то подразумевается 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 функции и хорошо подобраны по смыслу. Число формальных параметров не ограничивается, но не должно быть большим, для наглядности. Пример описания функции с формальными параметрами:

```
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;
};
```

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

6.5. 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 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.

6.6. Вызовы функций и возврат значений функции

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

Формально вызов функции можно записать так:

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

Или (чисто терминологически) можно записать так:

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

Пример вызова функции **Summa**:

```
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 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, а это означает, что в явном виде изменить параметр в функции нельзя. 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
// Значение SUM = 10 , а Sum = 5 ;
```

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

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

};

//

...

В главной программе (**first.cpp**):

...

```
int Summ3 (int a , int b, int * psum);
```

```
//...
```

```
Sum = Summ3 (2 , 3, &SUM); // Вызов с указателем
// Sum = 5 и SUM = 5
```

В дополнение к двум рассмотренным вариантам возврата значений из функций (через тип функции и через указатель), принципиально, можно вернуть значение 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;
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 , a Sum = 5 GSum = 5
```

6.7. 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, описанные в теле функции.
- Константы разного типа.
- Глобальные параметры данного исходного модуля (где описана функция) и тех, которые получены с помощью спецификатора **extern**.

Из одной функции могут быть вызваны другие функции и т.д., что позволяет проектировать сложную иерархическую систему функций.

В функции могут быть заданы такие формальные параметры, которые не могут быть в ней изменены. Для этого используется спецификатор **const** для 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 в функции. Модификатор **const** может быть использован для характеристики всей функции, это означает, что данная функция не может изменять данные объекта, но об этом вы больше узнаете в другом курсе.

6.8. 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 ноль 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 иметь ввиду что в СИ границы индексов не контролируются. Если массив имеет несколько измерений (например, двумерный массив), то размер каждого измерения передается отдельно.

6.9. 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.

6.10. Рекурсивные функции

В СИ допускается использовать рекурсивные функции, которые могут вызывать сами себя. Наглядно 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.

6.11. Макросы и переменные этапа компиляции

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

#define <имя этапа компиляции> <пробел " "> <выражение этапа компиляции>

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.

6.12. Параметры главной функции main

Главная функция программы 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.

6.13. 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:

```
//описания 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
```

6.14. Указатели на функции

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
```

7. Структуры данных в программах

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

7.1. 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 (ФИО) данные и числовые (даты и время) данные и вещественные данные (зарплата, стипендия). Для реализации технологии совместного хранения разнотипных данных в языки программирования добавлено понятие структуры данных. Технология структур данных позволяет в программах реализовать более высокий уровень абстракции данных и сделать программы более наглядными.

7.2. 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**). 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. Формализовано это выглядит так:

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

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  
} Student1, Group31[30]; // Необязательное описание переменных и массивов этой  
структуры
```

Обратите внимание, что после описания структуры дано описание конкретных структурных переменных. Из-за 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 { // комплексная переменная
double re; // действительная часть
double im; // мнимая часть
};
struct Date { // дата
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 отдельных полей структуры совпадает с описанием обычных переменных. Поля могут быть любого типа, в том числе и структурными переменными, за исключением типа самой описываемой структуры, хотя допустимо объявлять указатели на данный структурный тип. Описание структурных переменных производится с указанием имени структуры как нового типа данных и может быть выполнено многократно в программе. Шаблон структуры должен быть описан 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;
```

7.3. Инициализация структурных переменных

Инициализация структуры, как и для массивов и простых переменных, 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 лица
struct Face {
int MasF [5]; // Массив 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 вложенную структурную переменную: в структуре **Face** объявлена **Date**.

7.4. Работа с полями структуры через структуру и 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. 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.

Формализованная ссылка на поле структуры выглядит так:

<имя структуры>.<имя поля структуры>

Например:

```
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 структурных переменных в операторах присваивания.:

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

Можно использовать указатели на структурную переменную:

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

Тогда доступ к ее полю задается специальной операцией (->):

```
pStud->Stipen = 2000.0; // 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.

7.5. Многоуровневая квалификация в структурах

При использовании других структурных переменных (**Date** – см. выше) в качестве полей структуры, доступ к полю (поля) выполняется через двойную квалификацию:

```
struct Date { // дата
int day; // День
int month; // месяц
int year; // Год
};
// Описание структуры лица
struct Face {
int MasF [5]; // Массив целых
Date birthdate; // Структура даты
char name[50]; // Строка
};
...
Face Face2;
...
Face2.birthdate.day = 30; // Двойная квалификация
...
```

7.6. Указатели на структуры и на динамические структуры

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

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

```

Person p2 = { "Сидоров", {10, 3, 1978}, 1500.48 };
...
Person * pPerson = &p2;
...
// Изменить оклад
pPerson -> salary = 10.50;
(*pPerson).salary = 12.50; // Можно и так, но скобки обязательны

```

Ниже для вложенных структурных переменных мы увидим, в которых объявлены указатели на другие структуры, мы увидим, что допустима запись и такого вида:

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

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

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

7.7. Передача структур в функцию

Передать структуру в функцию, в качестве фактического параметра можно двумя способами: по значению, тогда изменить структуру в функции нельзя, и передать указатель на структуру. Рассмотрим первый случай. Пусть есть функция для печати поля структуры:

```

//
// Функция печати
void PrintPersonName( Person per)
{
    printf ("Печать в функции per.name = %s\n" , per.name);
    per.salary = 5.0;
};

```

Прототип в главной программе

```
void PrintPersonName( Person per);
```

В программе описания и вызов:

```

// Описание структуры
Person p2 = { "Сидоров", {10, 3, 1978}, 1500.48 };
//Вызов функции параметром структура
printf( "До функции p2.salary = %f \n" ,p2 .salary );
PrintPersonName( p2 ); // p2
printf( "После функции p2.salary = %f \n" ,p2 .salary );

```

Получим результат:

```

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

```

Значение полей структуры (в частности **p2.salary**) не изменяются.

7.8. Передача указателя на структуру в функции

Если мы хотим изменить значения полей в структурной переменной, то необходимо передать в нее указатель на нее. Для функции:

```

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

```

Прототип в главной программе

```
void PrintPersonName( Person per);
```

```

// Описание структуры

```

```

Person p2 = { "Сидоров", {10, 3, 1978}, 15.00 };

...
// Передача указателя
printf( "До функции p2.salary = %f \n", p2 .salary );
ChangePersonSalary( &p2 , 30.0); // Передача указателя = &p2
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

```

7.9. Массивы структур

Так как структура определяет новый тип переменной, то разрешается описывать массивы структур. Например:

```

// Описание массива структур типа Prepod
Prepod KafIU[30];
// Работа с элементами массива структур
KafIU[0].Oklad = 10.0;
KafIU[10].Oklad = 10.0;
// Цикл занесения
for (int i=0 ; i < 30 ; i++)
{
    KafIU[i].Oklad = 10.0;
};

```

Можно с массивом работать через указатель (**ptrMas**), при этом инициализация его должна быть проведена с начальным адресом массива структур (**&KafIU[0]**) или именем этого массива (**KafIU**), так как имя массива задает адрес начала массива.

```

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

```

Возможны способы: индексации указателя (**ptrMas[2].Oklad**), или вычислением нового адресного выражения для адреса (**ptrMas + 2**) или выбором конкретного элемента массива – структуры посредством операции разыменования (*). Смотрите ниже примеры:

```

ptrMas[2].Oklad = 15.0; // для второго элемента массива структур
(ptrMas + 2) -> Oklad = 25.0; // можно и так
(*(ptrMas + 2)).Oklad = 35.0; // можно и так
ptrMas = ptrMas + 2;
ptrMas->Oklad = 45.0; // можно и так
(*ptrMas).Oklad = 55.0; // можно и так

```

Во всех рассмотренных случаях будет изменена одно и тоже поле элемента структуры с номером 2.

7.10. Вложенные структуры

Внутри одной структуры может быть описана другая структура (тип **Prepod - DecPrep**) или указатель (тип **Person - pStudent**).

```

// Структура со вложенными указателями
struct Person {
    char name[50];
    Date birthdate; // структурная переменная дата рожденич
    double salary; // Оклад
};
//
struct Prepod {
    char fam[50]; // Фамилия
    Person * pStudent; // Указатель на структурную переменную Person
    double Oklad; // Оклад

```

```
};
    // Структура со вложенными структурами
    struct Decan {
        char fam[50]; // Фамилия
        Prepod DecPrep; // Структурная переменная Prepod вложенная
        double Oklad; // Оклад
    };

```

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

```
Prepod *ptrPerson = (Prepod *) malloc ( sizeof (Prepod));
ptrPerson ->pStudent = &p2;
ptrPerson ->pStudent ->salary = 15.00;
// Доступ возможен разными способами
printf( "p2 .salary = %f \n" ,p2 .salary );
printf( "ptrPerson ->pStudent ->salary = %f \n" ,ptrPerson ->pStudent ->salary );

```

В результате получим:

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

```

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

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

```

7.11. 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 и типу полей размер структуры может отличаться. Это показано на примере структур **First** и **Second**.

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

```

В программе определим актуальный размер структуры:

```
// Размещение структур в ОП
printf( "Размер структуры First = %d \n" ,sizeof (First) );
printf( "Размер структуры Second = %d \n" ,sizeof (Second) );

```

...

В результате получим:

```
Размер структуры First = 16
Размер структуры Second = 24

```

7.12. Динамической структуры

Работа с динамическими структурами и их массивами выполняется посредством указателей. Выделение памяти производится 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 полями уктур
strcpy( pDec ->fam , "Фамилия декана");
pDec ->Oklad = 50.00;
pDec = (Decan *) realloc( pDec , sizeof (Decan) * 2 ); // Изменение размера
//
pDec = pDec + 1;
strcpy( pDec ->fam , "Новая Фамилия декана");
pDec = pDec - 1; // Восстановление указателя для освобождения памяти (можно и
/запомнить)
free ( pDec );

```

Обратите внимание на использование функции **realloc**, позволяющей изменить размер выделенной памяти в два раза, а также добавление к указателю 1-цы. При операциях целого типа с указателями определенного вида 1-ца соответствует размеру типа, для которого объявлен данный указатель.

7.13. Создание и удаление динамической структуры со строками

В динамических структурах часто возникает задача работы со строками (символьными массивами). 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 памяти. Поэтому при инициализации таких структур экономнее выделить столько памяти, сколько необходимо, то есть динамической памяти. Аналогичная процедура необходима и для изменения значений строковых полей: сначала идет освобождение (**free**), а затем новый захват (**malloc**). Покажем это на примере. Пусть есть структура, в которой два указателя на строки (**pName** и **pAvtor**):

```

struct Book{
char * pName; // Название книги
char * pAvtor; // Автор книги
int StrCount; // Число страниц в книге
};

```

При ее инициализации нужно захватить память и записать строку:

```

Book Book1;
char * pStr = (char *) malloc ( strlen ("Три мушкетера") + 1);
Book1.pName = pStr;
strcpy(pStr , "Три мушкетера" );
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:

```

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

//

```

Если вся структура динамически порождается, текст программы будет выглядеть так:

```

// 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 отдельную строку, то память предварительно освобождаем:

```

        free (pBook->pName);
        pStr = (char *) malloc ( strlen ("Дама с камелиями") + 1);
        pBook->pName = pStr;
        strcpy(pStr , " Дама с камелиями" );

```

7.14. Структуры со ссылками на себя

В структурах нельзя использовать поля - структурные переменные такого же типа, однако поля - указатели на эти структуры допускаются. Например, элемент двухсвязного списка может выглядеть так:

```

// структуры со ссылками на самих себя
struct Node {
    Node * pNext;
    Node * pPrev;
    int ValList;
};

```

В данном случае **pNext** и **pPrev** являются указателями на структуру **Node**.

7.15. Перечисления - enum

Это набор именованных целых констант, используемый для большей наглядности программы и ее переносимости. Перечисление может быть использовано для объявления переменных, которые принимают значения на заданном множестве значений. Примеры для перечислений: дни недели, месяцы в году, времена года, типы переменных и т.д. Формально перечисления (**enum**) могут быть описаны так:

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

Например, для дня недели можно задать список констант:

```

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

```

Можно задать имя для объявления переменных перечисления:

```

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

```

Можно задать перечисления с произвольными значениями:

```

enum { Con1=5 , Con2=8, Con3=15 };
printf("Перечисления с произвольными значениями: Con1=%d Con2=%d Con3=%d\n"
, Con1 , Con2, Con3 );
// Проверка значений перечислений в операторе ветвления
int m = Con1;
if( m ==Con1 )
    printf("значение: Con1 , m =%d\n" , m );
else
    printf("значение: не Con1 \n" );

```

Можно задать перечисления с одним базовым значением:

```

enum { Const1=5 , Const2, Const3 };

```

Результат будет таким:

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

7.16. Союзы – union - объединения

Объединение — это место в памяти, которое используется для хранения переменных, разных типов. Можно задавать разные имена, одним и тем же полям оперативной памяти. Покажем на примере. Пусть есть объединение (три разных типа: **int** , **float** и **char**

*).

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

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

```
u_tag U1; // u_tag – это почти 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 );
```

```
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

8. Хранение данных и использование файлов

8.1. 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 ними операторы программы (команды микропроцессора) выполняют различные действия или операции. Когда программа завершилась сохранить результат ее работы можно запомнить так:

- Вывести результаты на экран дисплея.
- Вывести результат на печать (бумажная твердая копия).
- Записать результаты на внешний носитель (диски, флэшки и т.д.).

Первые две возможности не совсем удобны: с экрана информация стирается, бумага занимает много места, трудно в дальнейшем использовать полученную информацию и т.д. Вывод информации на диск требует выполнения определенных правил: нужно указать место достаточное для заполнения информации, нужно присвоить имя этой информации нужно задать правила ее записи и организации, для того чтобы в дальнейшем можно было бы уметь ее прочитать. Кроме того, запоминается вспомогательная информация: даты создания и изменения, количество информации (размер) и т.д. Для корректной работы с информацией в ИТ системах введено фундаментальное понятие – файл. Это понятие заложило основу теории для построения программных систем обработки, хранения и преобразования информации.

8.2. Понятие файла, его определения и разновидности

Существует много различных определений понятия файл. Это объясняется тем, что в разных условиях (пользователь, программист, ИТ - технолог, техник по оборудованию и т.д.) для работы необходимы разные свойства. Для пользователя важно знать название и расположение конкретного файла, его тип и возможно его размер. Заметим, что не нужно путать понятие файл с бытовым понятием файла – кармашка (из полиэтилена), хотя определенные моменты и здесь сходны (совокупность информации, хранимой вместе). Для программиста помимо названных выше свойств очень важна внутренняя организация файла, способы записи и чтения информации, наличие буферизации, форматирования. 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.

Обобщая все необходимое для программиста и пользователя, 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.

В некоторых литературных источниках, например в толковом словаре [11], 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 управления файлами – эффективно размещать файлы на внешних носителях и обеспечивать к ним доступ для чтения и записи информации. Различают следующие файловые системы: FAT, FAT32, NTFS,

HTFS и другие. Они используются в различных ОС и постоянно развиваются. Можно несколько “приземлить” определения файла следующим образом:

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

Из такого определения можно дать более частное определение понятие 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 наложено ограничение, и двоичными (бинарными), когда ограничений не предусмотрено.

Файлы можно представить 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').

Файл может рассматриваться как совокупность страниц (Page) или как совокупность бит информации. Файл может представлять закодированные определенным способом рисунки (совокупность разноцветных точек) или звуки (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. В комплексной лабораторной работе по дисциплине “Основы программирования” вы будете работать с такими файлами по своему варианту. Необходимо будет создать БД, добавить туда записи, отсортировать БД и так далее. Итогом работы будет завершенная программа работающая с БД.

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

Файл – это поименованная и упорядоченная совокупность команд (инструкций, операторов, процедур, функций, подпрограмм и т.д.), которые предназначены для выполнения на компьютере. 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 и программы, работающие с этими файлами. Например, данный документ (“Методические указания ...”) является файлом типа “Документ MS Word”, имеет сложную структуру, что, в конечном счете, определяет сложность программ (кстати, тоже файлов), работающих с такими файлами.

В данной лабораторной работе мы будем иметь место с относительно простыми, последовательными файлами, но освоим все основные операции и технологии работы с файлами.

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 длинная строка символов.

8.3. Операционная система, потоки и файлы

Операционная система (ОС) современного компьютера выполняет много действий связанных с файлами, как ресурсами 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 данные 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). Во-вторых, в ОС включены встроенные библиотеки, обеспечивающие чтение и запись данных с устройств, где расположены файлы.

Понятия устройств и файлов обобщены в ОС до понятия поток 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 имеет вид:

`FILE *stdin; // Стандартный ввод данных, умолчание клавиатура`

`FILE *stdout; // Стандартный вывод данных, умолчание экран монитора`

`FILE *stderr; // Стандартный вывод информации об ошибках, умолчание экран монитора`

Где **FILE** специальная структура для работы с файлами. В частности в нее записывается уникальный дескриптор (см. ниже) открытого файла.

8.4. 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 выше:

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.

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**). Если файл успешно открыт, то он становится доступным для работы в программе в заданном режиме. Если файл не открыт, то необходимо искать ошибку открытия и ее причину. После успешного открытия файла ОС выделяет для этого файла специальный дескриптор файла (передает в программу в структуру **FILE**) и специальный блок описания файла (FCB – File Control Block). Блок FCB – операционная система использует для контроля использования файла.

После работы с файлом он должен быть закрыт. Ос освобождает файл для дальнейшего использования, удаляет блок FCB. Функции библиотеки ввода и вывода **fclose** и **_close** и другие применяют для открытия файлов. Функция открытия файлов имеет следующий прототип:

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

Имя файла мы уже обсуждали, оно задается строкой, а режим открытия файлов может быть задан так:

"**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.

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

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

"**ab+**" - 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; // Структура описания файлов
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
```

```
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
```

8.6. 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], посмотреть примеры данных МУ к ЛР, или обратиться для помощи в **MSDN** [4] локальной и Интернет версии. Можно также открыть сам заголовочный файл (он имеет текстовый формат) и изучить его самостоятельно (!!!).

8.7. Основные операции и функции для работы с файлами

Операции над файлами могут быть разделены на две группы:

- Операции на содержимом конкретного файла и
- Операции над файлом в целом.

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**).
- Закрытие файла (**_close, fclose**).
- Операция чтения из файла (**_read, fread, fgetc, fgets, fscanf**).
- Операция записи в файл (**_write, fwrite, fputc, fputs, fprintf**).
- Перемещение текущего указателя файла и его определение (**fsetpos , fseek**).
- Определение конца файла (**_eof, feof**)
- Получение дескриптора файла (**_fileno**).

Для работы с файлами в целом 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 расположения (**_fullpath, div**)
- Изменение потока (**reopen**)
- Системные команды 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.

```
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
```

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

8.8. Уровни ввода/вывода и типы файлов

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

8.9. Описание файла в программе. Структура FILE.

Для работы с файлом его дескриптор нужно описать. Выше было отмечено, что значение дескриптора формируется 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
```

Структура **FILE** для работы с файлами в этом режиме показана ниже. Как видно из

```
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 структуры, вся информация о файле сохраняется на уровне операционной системы:

```
// Низкоуровневый - двоичный ввод - вывод
int pFBin = 0;
```

А закрытие файла выполняется следующим вызовом:

```
_close( pFBin ); // Низкоуровневое закрытие файла
```

8.10. Текстовые и бинарные (двоичные) файлы

Мы уже не один раз упоминали текстовые и бинарные файлы, но не уточнили, в чем заключается различие между ними. Во-первых, текстовые файлы могут быть использованы и в текстовом режиме. Главное различие текстовых и бинарных файлов заключается в способе представления цифровых данных. Символьные данные в этих двух случаях представляются одинаково. В текстовых файлах цифровые данные представляются символами, т.е. выполняется перевод в символьное представление (**fprintf**, **itoa** и др.). В бинарных файлах цифровое представление данных из оперативной памяти копируется в файл во внутреннем представлении, т.е. копируются байты, а не символьное представление данных. 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 символьно – “31” и “32”):

```
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 может быть получена следующим оператором программы (файл - pF):

```
fprintf( pF , "Строка1 !!! %d %d " , 1,2);
```

Для бинарного файла такая распечатка выглядит так (целое число 2 – представлено во внутреннем машинном виде – “02”):

```
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);

...
```

В примерах текста и распечатках **красным** цветом отмечены значения, соответствующие друг другу. Обратите внимание, что в бинарном файле “2” кодируется как “0200”, младший и старший байты типа **int** (двухбайтовый тип) расположены в обратном порядке.

8.11. Проверка конца файла и указатель чтения файла

При обработке (при чтении) файлов необходимо знать, при каких условиях цикл чтения записей может быть завершен. Другими словами нужно определить специальное условие конца файла. В общем, это может быть установлено 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 чтение специального байта – EOF- End Of File.
- Проверить значение числа байтов, полученных из файла к данному моменту, и оценить по этому значению фактическое завершение 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**).

8.12. Работа с текстовым/двоичным файлом с байтами - символами

Для записи и чтения байтовых текстовых файлов используются функции **fputc** (**putc**) и **fgetc** (**getc**). Для проверки конца файла при чтении используется 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+");
char ch;
// Вывод в файл
for ( int i = 0 ; StrOut[i] != '\0' ; i++)
{
    //putc( StrOut[i], pF ); // вывод одного символа
    fputc( StrOut[i], pF ); // вывод одного символа
};
fclose(pF);
```

Чтение и посимвольная распечатка файла:

```
// Ввод из файла
pF = fopen( "fputc.out" , "r"); // Открытие для чтения
while ( !feof(pF))
{
    char ch = getc(pF);
    if ( !feof(pF) ) // Для вывода маленького 'я' если ch== EOF (это 'я')
        printf( "%c", ch ); };
fclose(pF);
```

Результат чтения файла:

Пример строки для вывода в файл!

Если проверять просто символ конца файла (EOF):

```
...
    if ( ! (ch== EOF) ) //
        printf( "%c", ch ); };
...
```

то получим так ("я" не выводится на печать, а в файле есть!):

Пример строки для вывода в файл!

8.13. Работа с текстовым файлом построчно

Для записи и чтения строковых текстовых файлов используются функции **fputs** (**puts**) и **fgets** (**gets**). Для проверки конца файла при чтении используется функция **feof**.

Запись файла в цикле построчно на основе чисел случайно сгенерированных для каждой строки:

```

//////// Работа СО СТРОКАМИ
// Буфер ввода
char line[81]; // память для 80 символов + символ '\0'
...
// Вывод в файл (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
//
pF = fopen( "fputs.out" , "w+" ); // Открытие текстового файла для записи
// Генерация числа 1-100 и перевод в символьное и добавление 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
    strcat(line , itoa (n + 1 ,Num, 10 )); // при переводе + 1
    strcat(line , " \n"); // добавим конец строки
    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 ); // Чтение строки из файла
    if ( !feof(pF) ) // Проверка конца файла
        printf( "Строка из файла:%s" , line); // Перевод строки 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. Все строки не будут распечатаны. Кроме этого, размер буфера 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

8.14. Двоичные/двоичные файлы и функции fread и fwrite.

Для работы с двоичными файлами используется специальный набор функций (**fread** и **fwrite**). Эти функции позволяют читать и записывать записи, непосредственно в/из структурные переменные. Строго говоря, файлы для этих функций могут быть и текстовыми, или открыты как текстовые. Для чисто двоичных файлов характерно использование специальных функций (`_read` и `_write`) и механизм непосредственного чтения и записи из файлов посредством операционной системы, причем без буферизации. Для демонстрации возможностей будем использовать специальную структуру - **Person**:

```
// Структура для вывода и вывода
struct Person { // Структура для примеров
char Name[24]; // Фамилия
float Stipen; // Стипендия
int Kurs; // Курс
}; //
FILE * pF;

...
Person MasPers [] = { {"Иванов", 10.5f, 1 }, {"Петров", 100.5f, 2 } , {"Сидоров",
1000.5f, 3 } };
```

Запись файла на основе инициализированного массива структур:

```
// 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);
```

```
fclose(pF);
```

Цикл чтения файла бинарного файла:

```
// Ввод 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
    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
```

8.15. Форматированный ввод и вывод в файлы

Принципы и особенности форматированного ввода и вывода были рассмотрены в лабораторной работе № 1 по данному курсу. Здесь мы отметим, что форматирование полностью идентично. Для вывода в файлы функция **printf** должна быть заменена на функцию **fprintf**, а функция **scanf** должна быть заменена на функцию **fscanf**. Проверка конца файла может быть выполнена функцией – **feof**.

Запись файла **fprintf2.out**:

```
// 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);
fclose(pF); // Закрытие файла
```

Получили файл **fprintf2.out** (распечатано с помощью копии из notepad):

Строка - 1
Строка - 2
Строка - 3
Строка - 4
Строка - 5

Чтение записанного файла, данные в файле должны быть разделены пробелами:

```
// 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
}
fclose(pF); // Заккрытие файла
```

Результат чтения файла (прочитанные данные 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 и для файлового форматирования.

8.16. 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 работы с ними. Для этого уровня характерно использование 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 ); // Заккрытие файла
```

Структура “Студент” для демонстрации функций ввода вывода (Low-Level I/O):

```
struct Student { // Сведения о студенте
    char Name[20]; // Имя
    int Num; // Номер
    float Oklad; // Оклад
};
```

Запись файла с изменением цифровых данных строку **Name** не изменяем:

```
// Структуры для циклов
Student S1 = {"Петров", 1, 1000.0f}; // Для записи
// 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; // Номер
    S1.Oklad = 1000.0 * i; // Стипендия
    _write (pFBin, &S1, sizeof(Student));
};
_close( pFBin ); // Закрытие файла
```

В некоторых ситуациях возникает необходимость в изменении атрибутов файлов. 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:

```
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 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
```

8.17. Навигация по файлу fseek, lseek

Обычно чтение и запись файлов выполняется последовательно. После чтения текущей записи (байта, строки, структуры) специальный указатель устанавливается на следующую позицию автоматически. Часто необходимо прочитать часть файла (запись, поле, строку, блок или байт) не последовательно, а выборочно. Для этого предусмотрены специальные функции перемещения текущего указателя: **fseek** – для потокового ввода/вывода и **lseek** – для низкоуровневого. После перемещения указателя в другое место мы можем прочитать любую запись или заменить существующую запись другой. Такие операции возможны для файлов, открытых в двоичном режиме. Покажем на примерах их применение.

При перемещении указателя мы можем сместиться от начала файла (**SEEK_SET**), конца файла(**SEEK_END**) и текущей позиции в файле (**SEEK_CUR**). Далее в функции 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):

```

// Позиционирование чтение
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
PosNew = fseek( pF, (Pos - 1)*sizeof(Person), SEEK_SET); // Перемещение указателя
на новую позицию
fread( &PWork, sizeof(Person) , 1, pF);
printf( "Позиция персона из файла %s %f %d\n", PWork.Name ,
PWork.Stipen , PWork.Kurs );
fclose(pF); // Закрытие файла
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 };
PosNew = fseek( pF, (Pos - 1)*sizeof(Person), SEEK_SET);
fwrite( &PNew, sizeof(Person) , 1, pF);
flush( pF ); // Запись буфера на диск
rewind(pF); // указатель в начало файла
fread( &PWork, sizeof(Person) , 1, pF); // Контрольное чтение сначала
printf( "Позиция персона из файла %s %f %d\n", PWork.Name ,
PWork.Stipen , PWork.Kurs );
fclose(pF); // Закрытие файла
Для низкоуровневого ввода вывода перемещение указателя и прямое чтение будут
выглядеть так:

```

```

// 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 ); // Закрытие файла
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 ); // Закрытие 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

```

8.18. Перенаправление потоков ввода и вывода

С помощью специальных функций можно стандартный поток перенаправить в файл. Для этого используется функция **freopen**. 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.

8.19. 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 [5], к разделу № 6. Кроме этого, вы можете воспользоваться справочными системами и информацией из Интернет.

Эти средства позволяют выполнить в удобной форме множества операций с файлами и каталогами: от поиска файлов, их просмотра и до их ручной модификации.

8.20. 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
```

8.21. Работа со строками и консолью : sscanf, sprintf и printf

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 );
```

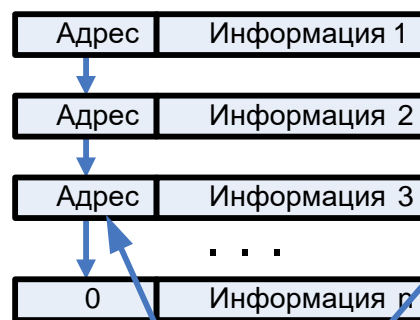
9. Списковые структуры данных

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

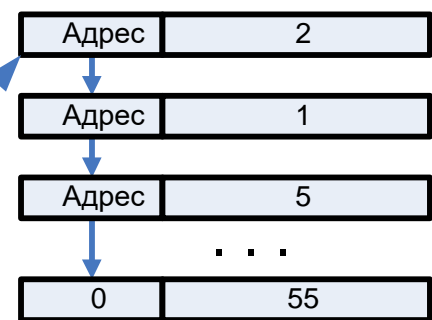
9.1. Понятие список

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

Однонаправленный список



Список целых значений



Указатели

Данные

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

...

// Описание и инициализация элемента LFirst
Elem LFirst = { NULL , 3 }; // NULL – нулевой указатель – нет ссылки на другие элементы
// Значение ListVal = 3
```

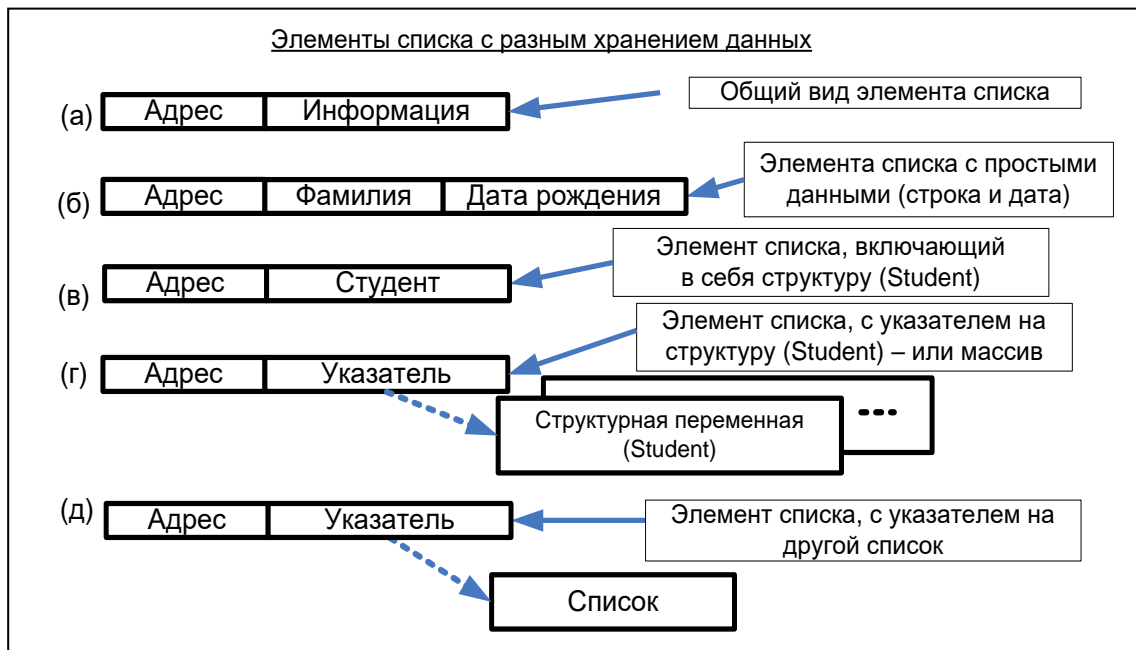
9.2. Особенности структур списков и их назначение

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.

9.3. 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.

9.4. 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:

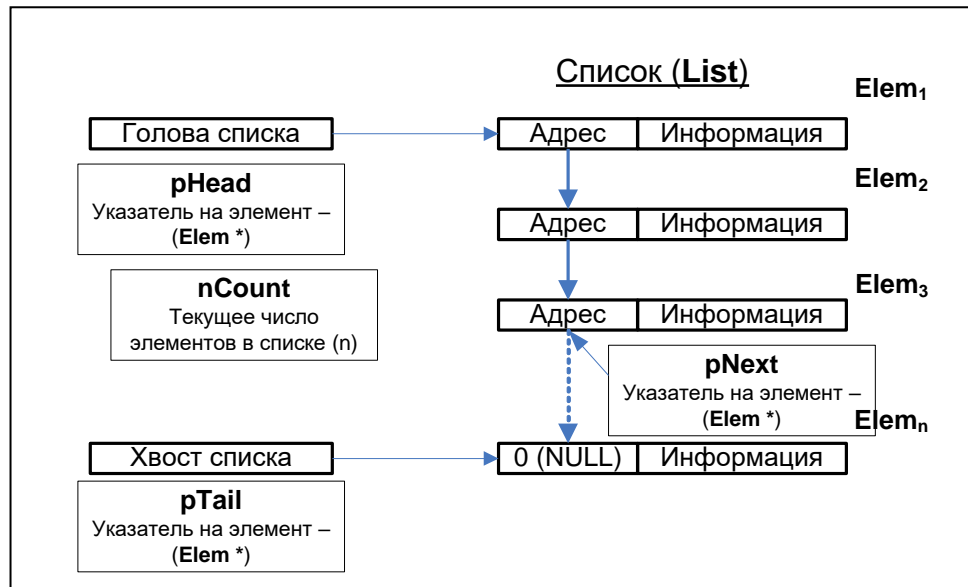
Элемент простого списка Elem: 1

Элемент простого списка Elem: 2

Элемент простого списка Elem: 3

9.5. 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
головы 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
```

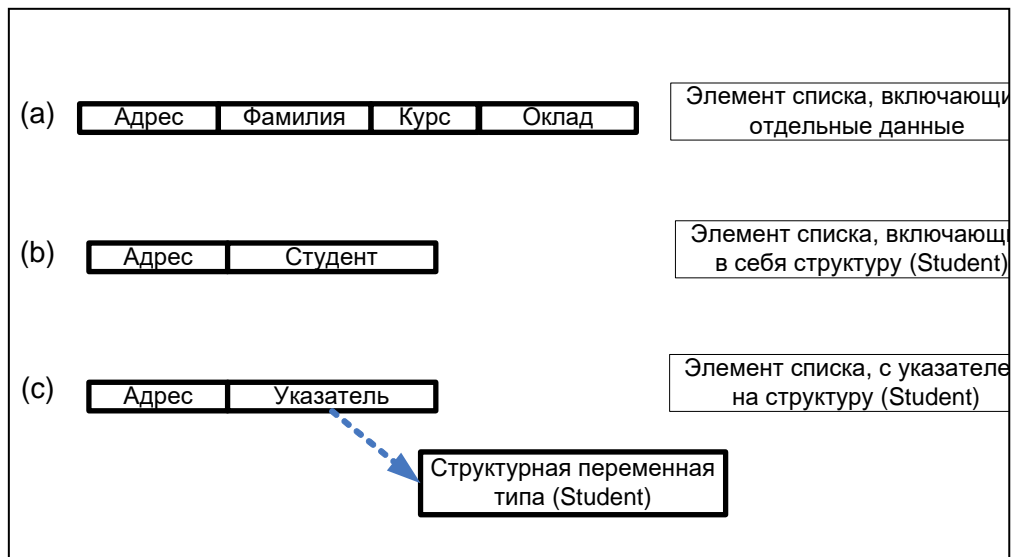
Важно в дополнение отметить следующее. Данный список является однонаправленным: движение по списку возможно только в одном направлении – от головы к хвосту, так заданы указатели. У последнего элемента списка (хвоста списка), по стандартному соглашению, указатель-адрес задается равным нулю (0, NULL – 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 формируем 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.

9.6. Структура элемента списка с вложенными и внешними данными

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

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

На рисунке, представленном ниже, показаны эти варианты.



Примерами вариантов таких структур могут служить следующие:

```
// Структура 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 bElem{
    bElem * 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
};
```

9.7. Однонаправленные и двунаправленные списки

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

```

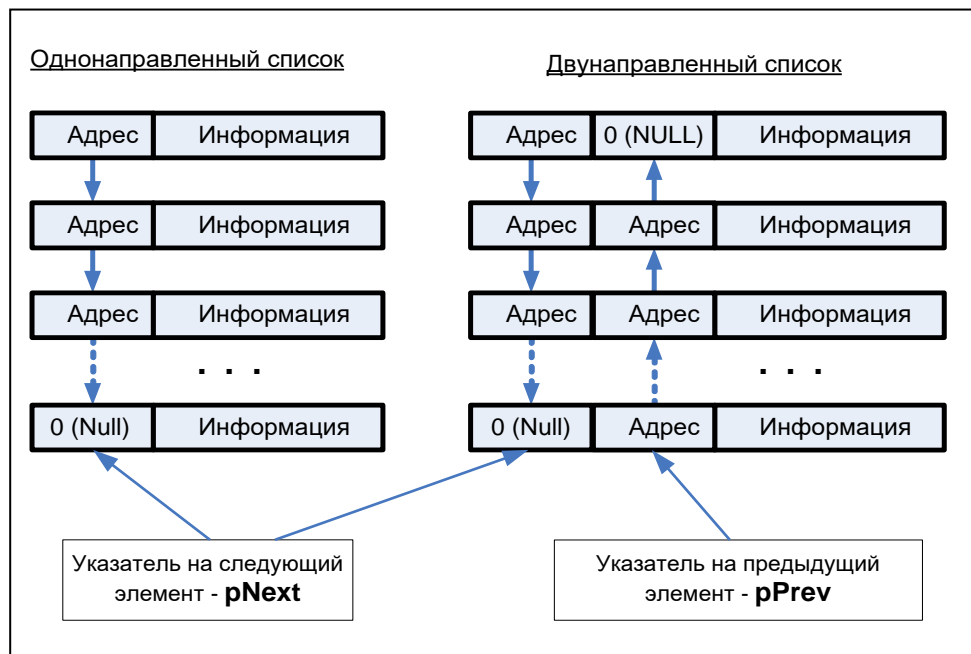
struct DElem{
    DElem * pNext; // адрес следующего элемента списка (Заметьте, указатель имеет тип (Elem *))
    DElem * pPrev; // адрес ghtisleotuj элемента списка (Заметьте, указатель имеет тип (Elem *))
    int ListVal; // информация, содержащаяся в списке
};

```

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:



9.8. Статические и динамические списки

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

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

```

// Самый простой элемент списка
struct ListElem{
    ListElem * pNext; // адрес следующего элемента списка
    int ListVal; // информация, содержащаяся в списке
};
// Структура простого списка
struct List {

```

```
ListElem Head; // Голова списка 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 элемннга 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.

9.9. Операции для работы со списком

Основные общие операции для работы со списками:

- Создание нового списка
- Добавление нового элемента в список (в голову, в хвост и по номеру)
- Удаление элемента из списка (из головы, с хвоста и по номеру)
- Очистка всего списка
- Распечатка всего списка
- 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.

9.10. Ручная работа с однонаправленным списком

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

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

```
};
```

```
...
```

Фрагмент программы для ручного связывания статического однонаправленного списка:

```
// Ручное связывание и распечатка списка
ListElem E11;
ListElem E21;
```

```

ListElem E31;
E11.ListVal = 1; // Значение информации 1-го элемента
E11.pNext = &E21; // Ссылается на 2-й элемент
E21.ListVal = 2; // Значение информации 2-го элемента
E21.pNext = &E31; // Ссылается на 3-й элемент
E31.ListVal = 3; // Значение информации 3-го элемента
E31.pNext = NULL; // не ссылается ни на кого
// Печать списка
ListElem ETemp = {&E11, NULL}; // Временный элемент для навигации
printf("Содержимое ручного списка: \n");
while (ETemp.pNext != NULL)
{
    printf("Элемент = %d \n", ETemp.pNext->ListVal);
    ETemp.pNext = ETemp.pNext->pNext; // Навигация
}

```

...

Результат работы программы:

Содержимое ручного списка:

Элемент = 1

Элемент = 2

Элемент = 3

9.11. Добавление в голову списка

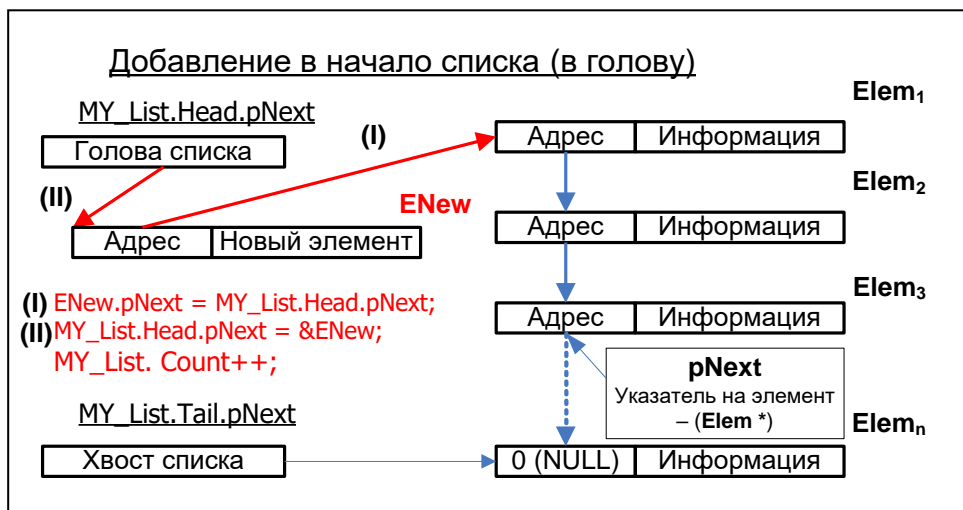
Для добавления элемента в голову списка объявим структурную переменную список (**MY_List** типа **List**) и вручную запоем значения для головы и хвоста списка.

```

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

struct List {
    ListElem Head; // Голова списка
    ListElem Tail; // Хвост списка
    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:

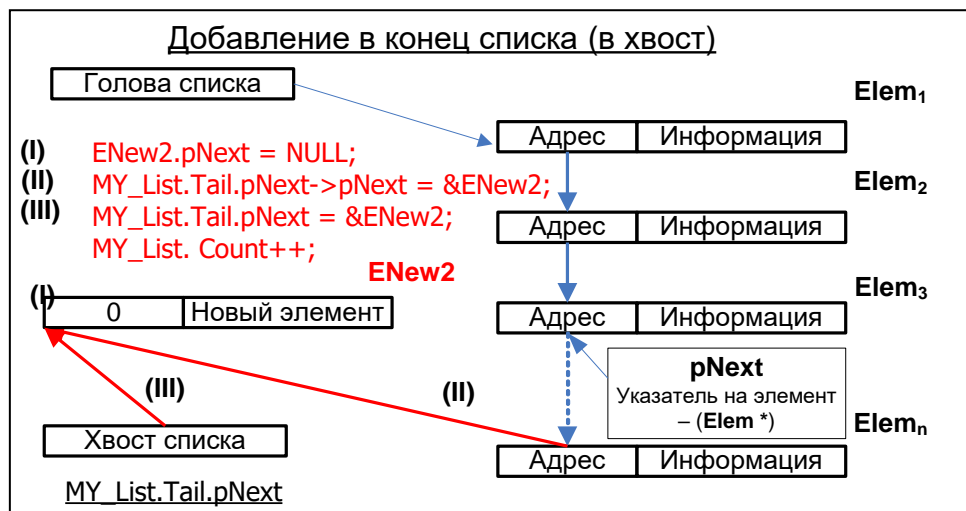
```

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

```

9.12. Добавление в конец списка (в хвост)

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



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

```

9.13. Функция распечатки однонаправленного 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:

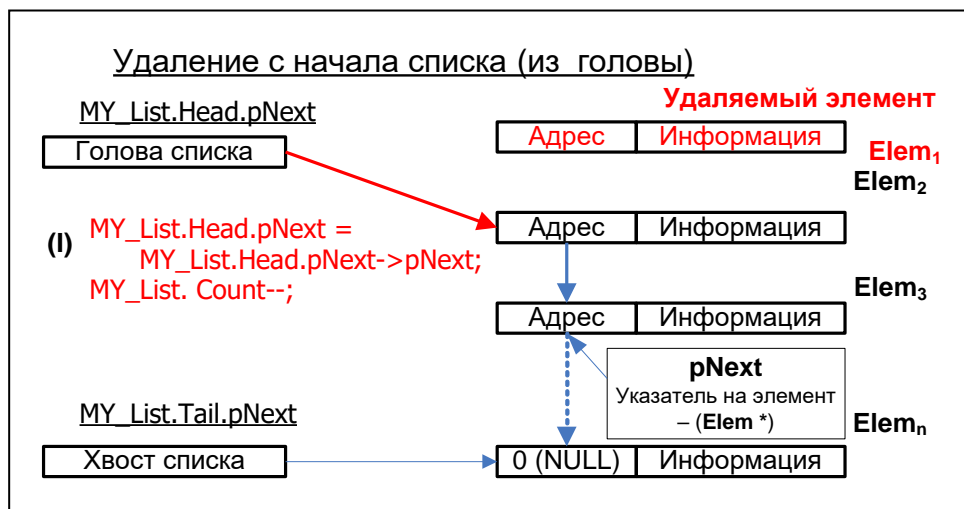
```
PrintElemList ( E11 );
```

Результат вывода:

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

9.14. Удаление из головы списка

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

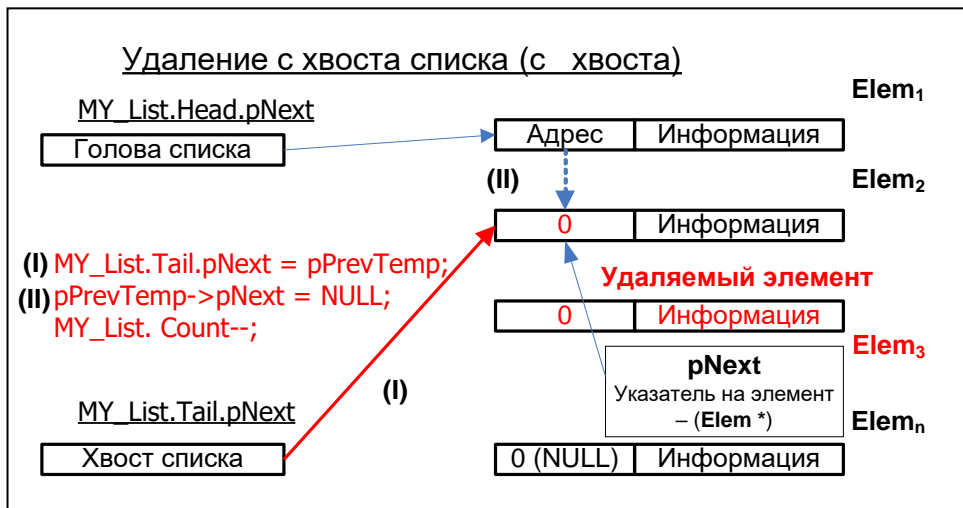
Элемент = 2

Элемент = 3

Элемент = 10

9.15. Удаление из хвоста списка

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



ляется 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.

9.16. Очистка статического списка

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

9.17. 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

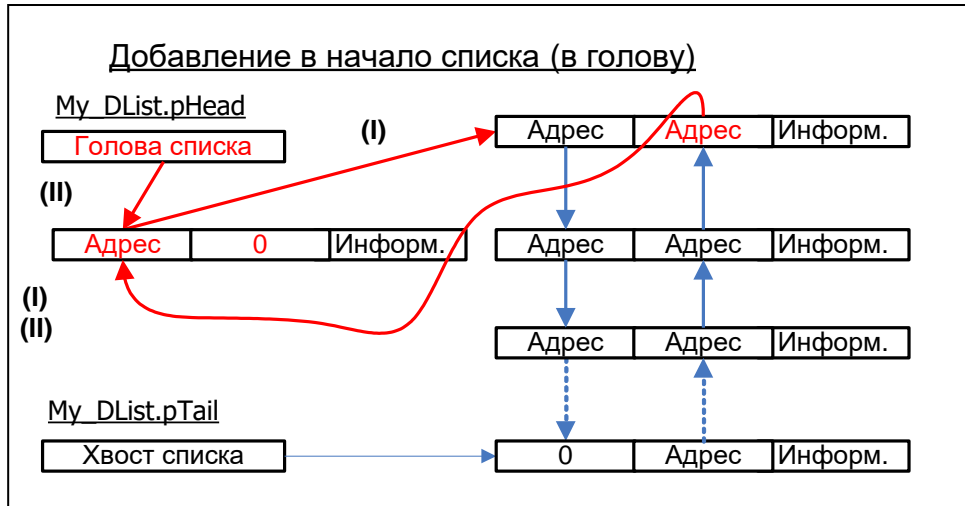
Элемент списка: - 10

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.

9.18. 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
```

10. Требования к КЛР/ДЗ и варианты

10.1. Цель КЛР/ДЗ (ЛР №10)

Комплексная лабораторная работа/Домашнее задание (КЛР/ДЗ) № 10 выполняется для получения навыков совместного использования знаний по отдельным приемам программирования, изучаемым в курсе основы программирования, после выполнения комплекса отдельных лабораторных работ дисциплины “**Основы программирования**”.

В предыдущих работах студенты изучают следующие темы: операторы, структуры, указатели, строки, файлы, массивы, алгоритмы и списки. Студенты создают файл своих структурных переменных (набор записей) и комплекс специальных функций-операций для работы с этим файлом и этими записями. Предусматриваются важные операции с файлом: запись файла, его распечатки, его сортировки, доступа к записям в файле, изменение записей, поиск записей и перезапись его в массив записей и обратно. Фактически, на уровне файлов, студенты осваивают работу с прообразом однотабличной базой данных (БД), что в дальнейшем поможет более глубоко понять функционирование сложных систем управления базами и банками данных (СУБД).

10.2. Требования к заданию на ЛР №10

Данное описание ЛР и задание на ЛР может быть выполнено на трех уровнях сложности, в зависимости от выбора студента:

- базовый уровень (уровень А),
- продвинутый уровень (уровень В),
- уровень для сильных студентов (уровень С).

Пункты задания выделены цветом, соответствующим выбранному уровню. В дальнейшем, конкретизирую требования на ЛР, мы будем использовать эти цвета для привязки к одному из уровней задания. Проще - цвет пункта задания определяет его уровень сложности. Студент может и частично использовать пункты задания из разных уровней, однако **базовый уровень** является обязательным.

Примечание 1. Данную лабораторную работу (№ 10) мы будем также именовать комплексной лабораторной работой (КЛР) либо домашним заданием (ДЗ), так как в отдельных учебных планах на семестр иногда появляется такая позиция. В общем возможен и комбинированный вариант КЛР/ДЗ.

10.3. Особенности описания разделения заданий по уровням ЛР №10

Описание шагов выполнения КЛР и требований к ней намеренно предлагается в двух вариантах: вариант (№1), в котором примеры располагаются отдельно от текста и вариант (№2), в котором примеры и решения вставлены в текст методических указаний. На сайте для работы студентам доступны оба варианта. Если студент стремится и способен получить максимальный уровень знаний (даже более чем по плану) он должен попробовать использовать первый вариант методических указаний и обращаться к подсказкам во втором варианте только в крайнем случае. Если студенту пока трудно самостоятельно выполнить какой-то конкретный шаг задания, то он может использовать подсказку (извините, шпаргалку) и на основе ее сделать собственное задание.

10.4. Функции и структуры в домашнем задании

Как было отмечено выше, студенты в рамках ЛР разрабатывают программный

комплекс, основанный на файловой системе, иллюстрирующий основные операции работы с базами данных (БД). Программный комплекс состоит из: набора специальных функций (фактически библиотеки), и фрагментов программ, для демонстрации работы этих функций. Например, должны быть разработаны функции заполнения файла записями, его распечатки, изменения записей и другие. Кроме этого, в главном модуле могут быть реализованы некоторые действия с файлом без оформления функций (в каждом пункте заданий указывается, когда нужно создавать функцию). Работа с файлами основывается на специальной структуре данных, которую студент самостоятельно разрабатывает в предложенной предметной области (см. варианты задания). Структура данных должна иметь осмысленные поля и представлять единое целое. Например, для описания студента можно выделить поля: фамилия студента, курс студента, размер стипендии и т.д. Часть названий полей предложены в варианте (см. ниже), хотя по согласованию с преподавателем их можно заменить. Неизменным должно оставаться только содержательное название структуры данных (студент, книга и т.д.).

10.5. Варианты для выполнения ЛР

Варианты для выполнения работы студентами представлены в таблице, в дополнение к трем указанным полям в структуре студент должен добавить еще любые два дополнительных поля, которые он придумает сам. Эти поля должны иметь смысл для собственного проекта. Например, для структуры типа студент можно добавить поля: курс обучения, год поступления, процент посещения занятий и т.д.

В таблице расположенной ниже приведены основные названия структурных переменных и обязательных полей по вариантам.

Вар.	Структура (имя структуры придумать самому)	Поля структуры и их тип			Примечание (интегральная характеристика)
		Поле-имя-тип	Поле-имя-тип	Поле-имя-тип	
1.	Кафедра	Название (Name) –char[]	Число студентов (CountS) - int	Число преподавателей (CountP) - int	Среднее Число студентов на кафедре
2.	Книга	Название (Name) –char[]	Автор (Avtor) – char[]	Число страниц (nCountS) - int	Среднее Число страниц
3.	Файл	Имя файла (NameFile) – char[]	Дата создания (DateFile) – char[]	Размер файла (SizeFile) - int	Средний Размер файла
4.	Автомобиль	Марка автомобиля (Marka) – char[]	Стоимость (Cost) - double	Мощность (Kraft) - float	Средняя Мощность
5.	Компьютер	Владелец (Fam) – char[]	Размер ОП (SizeMem) - int	Объем HDD (SizeHDD) - int	Средний Объем
6.	Группа	Индекс группы (Name) –char[]	Число студентов (CountS) - int	Средняя оценка в группе в % (AvExam) - float	Среднее число студентов в группе
7.	Человек	Фамилия (Fam) – char[]	Пол (Pol) - char	Возраст (Age) - int	Средний Возраст
8.	Стеллаж	Название (Namt) –char[]	Материал (Mat) - char	Число полок (nPol) - int	Среднее число полок

Вар.	Структура (имя структуры придумать самому)	Поля структуры и их тип			Примечание (интегральн. характеристика)
		Поле-имя-тип	Поле-имя-тип	Поле-имя-тип	
9.	Дом	Улица (Fam) – char[]	Число этажей (nStage) - int	Номер дома (Numb)- char[]	Среднее число этажей
	Вариант представленный в примере выполнения ЛР № 10				
	Студент (Student)	Имя (Name) - char[20]	Номер (Num) - int	Стипендия (Oklad)- double	Задание моего примера

Примечание 2: Номер варианта студента уточняется по журналу группы. При желании студент может предложить свой вариант темы структурной переменной, но при этом он должен согласовать его с преподавателем.

10.6. Порядок выполнения работы (Уровень А и В)

1. **Создать** консольный проект (главная программа, вспомогательный модуль для функций и заголовочный файл для структур/констант)

2. **Описать** свою структуру, придумав ее **название**. Типы и названия полей взять из таблицы вариантов (см. ниже). Все данные структуры статические. Дополнить описание структуры двумя полями, придумав их по смыслу. Описание структуры выполнить в заголовочном файле проекта (<имя проекта>.h).

3. **Описать** и заполнить одну свою простую структурную переменную с помощью инициализации при ее описании.

4. **Описать** и заполнить еще одну свою структурную переменную вручную (с помощью операторов присваивания – числовые данные и функции копирования – строчные данные). Распечатать значения ее полей двух структурных переменных (Функция - **printf**).

5. Разработать **функцию** распечатки одной данной структурной переменной, передавая в качестве параметра ее адрес, и проверить ее на описанных ранее структурных переменных.

6. **Создать** динамическую структурную (функция – **malloc**, библиотека - **<malloc.h>**) заполнить ее динамически и распечатать значение через указатель и с помощью специальной функции печати. Удалить созданную динамическую переменную.

7. **Описать** и проинициализировать массив структурных переменных не менее 4-х. Распечатать этот массив с помощью функции п.5.

8. **Разработать функцию распечатки массива** с записями структурных переменных.

Формальные параметры этой функции: указатель на массив структурных переменных и его размер. Использовать свою функцию печати одной структурной переменной. Проверить работу этой функции.

9. Создать цикл для заполнения нового динамического **массива** записей полями со случайными значениями параметров (размер массива 5-6 элементов). Предусмотреть случайное заполнение одного числового значения (использовать функцию - **rand**) и одного символьного значения с номером (использовать функции: **rand** и **atoi** вместе с функциями копирования/слияния текстов - **strcat, strcpy**). Распечатать созданный массив в отдельном цикле.

10. **Придумать** название двоичному файлу для выполнения задания (в нашем примере - "BDStud.bin", например "BDBook.bin" – для книг).

11. **Создать** цикл для заполнения файла записями (запись в файл) с разными значениями числовых параметров изменяемых в цикле (номер в журнале группы) или основанных на начальной инициализации своего массива структур.

11. **Создать** цикл чтения и распечатки сформированного двоичного файла. Данные из файла ввести в статический массив структур, задав его максимальный размер равным 100.

11. **Создать** цикл для заполнения файла записей полями со случайными значениями параметров. Предусмотреть случайное заполнение одного числового значения (использовать функцию - **rand**) и одного символьного значения (использовать функции: **rand** и **atoi** вместе с функциями копирования/слияния текстов - **strcat, strcpy**). Распечатать созданный файл в цикле заполнения.

12. **Разработать функцию** заполнения динамического массива структурных переменных на основе двоичного файла с записями структурных переменных. Проверить ее работу. Массив распечатать специальной функцией (см. выше).

13. **Разработать функцию** заполнения двоичного файла на основе массива структурных переменных (описанных и проинициализированных выше). В функцию должны передаваться: имя заполняемого файла, адрес исходного массива, размер массива. Изменить некоторые значения записей в массиве структур перед использованием. Для этого в самом массиве, заполненном выше, сделать изменения (например, имя 2-й записи: имя и номер), используя фиксированное значение индекса элемента структурного массива. Проверить работу этой функции. Заполнение файла проверить в файл менеджере (**far** или **total commander**), с помощью операции просмотра файла в шестнадцатеричном формате.

14. **Разработать** отдельную функцию распечатки файла с записями своих структурных переменных. Проверить работу этой функции.

15. Разработать функцию определения числа записей в файле. Проверить ее работу на своем файле.

16. Разработать функцию очистки файла записей. Проверить ее работу совместно с функцией распечатки файла записей – ничего не должно напечататься, кроме сообщения “Записей в файле нет!” – из функции печати. Кроме этого проверить размер файла в **файл – менеджере**. Этот размер должен быть равен нулю.

17. Проверить использование функций удаления файла и снятия атрибута защиты файла, взяв ее из проекта примера. Заполним снова и распечатаем.

18. **Разработать функцию Swap** двух структурных переменных (по адресу элемента)

19. На основе этой функции (**Swap**) выполнить сортировку массива структурных переменных по одному числовому значению, используя метод пузырьковой сортировки. Оптимизацию алгоритма сортировки не проводить. Распечатать массив до и после сортировки.

20. Разработать функцию сортировки массива структурных переменных по числовому значению. Проверить ее работу. Распечатать массив до и после сортировки.

21. Разработать функцию сортировки **файла** структурных переменных (записей). Алгоритм функции включает три шага: (1) чтение двоичного файла в массив – уже есть готовая функция; (2) пузырьковая сортировка массива структурных переменных – уже есть готовая функция и (3) перезапись массива в двоичный файл – уже есть готовая функция. Проверить сортировку файлов со структурными переменными.

22. **Прочитать** 2-ю запись из сформированного файла структурных переменных и ее распечатать с помощью собственной функции.

22. **Создать** функцию для чтения записи из сформированного файла структурных переменных по номеру и ее распечатать с помощью собственной функции.

22. Разработать и фрагмент программы для интегральных вычислений в БД (см. варианты в таблице). Интегральные вычисления для вашего файла структурных переменных заключаются в последовательном чтении всех записей файла. Из каждой из полученных записей для отдельного параметра выполняется суммирование, подсчет, сравнение и т.д. Шаги выполнения этой задачи могут быть такими: чтение файла в массив, вычисления по массиву в цикле (например, числа пятерок), обработка результата (например, получение среднего числа пятерок у студента) и распечатка результата. Можно в цикле по шагам читать последовательно все записи и вычислять интегральные характеристики.

22. Оформить в виде **функции** фрагмент программы для вычисления интегральной характеристики по вашему варианту (см. предыдущее задание).

11. Работа в режиме командной строки

Режим командной строки используется системными программистами и пользователями для выполнения команд операционной системы (ОС) и командных файлов (*.bat). Кроме того, в режиме командной строки может быть запущена любая программа для операционных систем ДОС (или в режиме эмулятора ДОС) и WINDOWS.

11.1. Режим командной строки и его назначение

Режим командной строки доступен в ОС для выполнения различного рода системных и пользовательских работ. Такой режим называют еще пакетным режимом работы программ. Он используется для следующих случаев:

- Выполнения системных работ, например инсталляции ОС.
- При выполнении ремонтных работ в ОС, когда по каким-либо причинам работа ОС WINDOWS невозможна (например, при заражении вирусами).
- Запуска программ и команд ОС, результаты которых должны поступать на дисплей непосредственно.
- Запуска командных файлов (*.bat).
- Запуска системных программ работающих в этом режиме, например утилиты MEM.EXE для просмотра состояния оперативной памяти и работающих программах.
- Выполнения многих других работ.

При запуске режима командной строки на экране появляется специальная подсказка (“>”), после которой можно вводить текст команд:

```
>DATE_↓
```

Формат подсказки можно изменить специальной командой **PROMPT**. После ввода команды, она будет выполнена, а строки будут сдвигаться вверх (перечень команд можно найти в литературе, в электронном справочнике – ЛР № 1, или получить автоматически в виде справки). Например, после ввода команды **DATE** (получения и изменения системной даты) мы получим:

```
>DATE
Текущая дата: Сб. 21.02.2009
Введите новую дату (дд-мм-гг):
>22-02-2009_↓
```



Символом “_↓” здесь я обозначил клавишу “**Enter**” (и далее буду его использовать для этой цели), которую необходимо нажать для ввода команды. Если Вы введете в режиме командной строки полное название программы, работающей под WINDOWS, то она тоже будет выполняться, но в отдельном окне.

При выполнении командных файлов, команд ОС их результаты будут размещаться в окне командной строки. Нужно иметь в виду, что при сдвиге строк их число в окне ограничено, поэтому в верхней части экрана они будут пропадать. Число строк, которое можно просмотреть в оконном режиме командной строки (см. ниже) может превышать стандартный размер экрана (25 строк) и может настраиваться.

11.2. Разновидности командных интерпретаторов

В современных операционных системах (WIN 32) сохранилась возможность запуска нескольких модификаций командных интерпретаторов (или процессоров). Ранее, при запуске операционной системы ДОС командный интерпретатор запускался автоматически при старте ОС и дальнейшая работа проходила только в этом режиме. Различают следующие разновидности командных интерпретаторов:

- COMMAND.COM – 16-ти разрядный

- CMD.EXE – обновленный командный процессор с расширенными возможностями, эмулирующий работу в MS DOS.

Различают также режимы работы командных интерпретаторов в оконном режиме и в полноэкранном режиме. При переключении в полноэкранный режим (клавиши Ctrl+Enter) командный интерпретатор выполняет свои функции в более полном объеме. Возврат в оконный режим выполняется также с помощью клавиш - Ctrl+Enter.

11.3. Запуск и завершение работы режима командной строки

Режим командной строки может быть запущен явно и неявно. Неявный запуск выполняется автоматически при запуске программ и утилит, работающих в режиме ДОС. Явный запуск командного интерпретатора может быть выполнен так:

ПУСК(Start) => Выполнить...(Run...) => COMMAND.COM => OK

Или

ПУСК(Start) => Программы...(Programm...) => Стандартные => Командная строка

Можно создать ярлык для запуска командного процессора, при этом становятся доступными настройки для его выполнения (посмотрите самостоятельно). Завершение работы режима командной строки выполняется: либо командой EXIT, выполняемой в этом режиме, либо, если включен оконный режим, обычным закрытием окна при нажатии кнопки в правом верхнем углу окна ("х"). Первый способ является более корректным.

При выполнении командного файла возможен вложенный вызов командного интерпретатора, например, если вызывается вложенный командный файл. В этом случае параметры его запуска нужно задать специальными командами (SHELL в файле CONFIG и COMMAND и COMSPEC – см. литературу по MS DOS и лекции по курсу ОС).

Примечание: Здесь и в дальнейшем я ориентируюсь на работу в ОС Windows XP, для других операционных систем возможны некоторые отличия, в частности в названиях системных пунктов меню.

11.4. Запуск команд и программ в режиме командной строки

Запуск команд и программ в режиме командной строки выполняется ручным набором имени файла программы или команды после подсказки.

```
>charmap./
```

Выше приведен пример запуска программы WINDOWS, которая запустится в отдельном окне.

```
>c:\dn\dn.com./
```

При запуске программ нужно удостовериться, что ОС известен путь (PATH) для запуска программы. Если программа не запускается, то необходимо указать явный путь (как в примере выше) или сделать директорию (каталог) для запуска программы текущей:

```
>c:
>cd c:\dn\
>dn.com./
```

При запуске программ нужно быть внимательным и набирать имя программы или команды точно. Чаще всего, чтобы избежать ошибок, необходимо набирать и расширение для файла. Если команда или имя программы введены неверно, то Вы получите сообщение операционной системы вида:

```
>c:\dn\dn.exe./
"DN.exe" не является внутренней или внешней
командой, исполняемой программой или пакетным файлом.
```

В этом случае нужно проверить: путь, текущий каталог, имя программы или команды и выполнить ввод заново.

11.5. Получение справок о командах в режиме командной строки

Информацию о командах режима командной строки можно получить в литературе или в электронных справочниках (см. ЛР № 1). Кроме того, оперативно можно справку о конкретной команде с помощью директивы HELP, например:

> help EXIT	
Завершает программу CMD.EXE (интерпретатор команд) или текущий пакетный файл-сценарий.	
EXIT [/B] [exitCode]	
/B	Предписывает завершить текущий пакетный файл-сценарий вместо завершения CMD.EXE. Если выполняется вне пакетного файла-сценария, то будет завершена программа CMD.EXE
exitCode	Указывает цифровое значение. Если указан ключ /B, определяет номер для ERRORLEVEL. В случае завершения работы CMD.EXE, устанавливает код завершения процесс с данным номером.

Полный перечень команд можно получить, выполняя команду HELP без параметров:

>help	
Для получения сведений об определенной команде наберите HELP <имя команды>	
ASSOC	Вывод либо изменение сопоставлений по расширениям имен файлов.
AT	Выполнение команд и запуск программ по расписанию.
ATTRIB	Отображение и изменение атрибутов файлов.
BREAK	Включение/выключение режима обработки комбинации клавиш CTRL+C.
CACLS	Отображение/редактирование списков управления доступом (ACL) к файлам.
CALL	Вызов одного пакетного файла из другого.
CD	Вывод имени либо смена текущей папки.
CHCP	Вывод либо установка активной кодовой страницы.
CHDIR	Вывод имени либо смена текущей папки.
CHKDSK	Проверка диска и вывод статистики.
CHKNTFS	Отображение или изменение выполнения проверки диска во время загрузки.
CLS	Очистка экрана.
CMD	Запуск еще одного интерпретатора командных строк Windows.
COLOR	Установка цвета текста и фона, используемых по умолчанию.
COMP	Сравнение содержимого двух файлов или двух наборов файлов.
COMPACT	Отображение/изменение сжатия файлов в разделах NTFS.
CONVERT	Преобразование дисковых томов FAT в NTFS. Нельзя выполнить преобразование текущего активного диска.
COPY	Копирование одного или нескольких файлов в другое место.
DATE	Вывод либо установка текущей даты.
DEL	Удаление одного или нескольких файлов.
DIR	Вывод списка файлов и подпапок из указанной папки.
...	

Здесь я сознательно “обрезал” список получаемых в справке команд, предлагаю Вам самим получить и скопировать этот список.

Командный интерпретатор CMD.EXE может работать и в расширенном режиме. В этом режиме доступны дополнительные возможности. Для описания этих возможностей необходимо в режиме командной строки вызвать командный интерпретатор с параметром справки:

>CMD.EXE /?	
Запуск новой копии интерпретатора команд Windows XP.	
CMD [/A /U] [/Q] [/D] [/E:ON /E:OFF] [/F:ON /F:OFF] [/V:ON /V:OFF] [[/S] [/C /K] строка]	
/C	Выполнение указанной команды (строки) с последующим завершением.

/K	Выполнение указанной команды (строки) без последующего завершения.
/S	Изменение поведения после /C или /K (см. ниже)
/Q	Отключение режима вывода команд на экран (ECHO).
/D	Отключение выполнения команд AutoRun из реестра (см. ниже)
/A	Вывод результатов выполнения команд в формате ANSI.
/U	Вывод результатов выполнения команд в формате UNICODE.
/T:цв	Выбор цвета текста/фона (более подробно см. COLOR /?)
/E:ON	Разрешение расширений команд (см. ниже)
/E:OFF	Запрет расширений команд (см. ниже)

...

Ниже в этой справке дано подробное описание возможностей расширенного режима. Запуск интерпретатора в расширенном режиме выполняется так:

```
>CMD.EXE /E:ON.␣
```

Выключение расширенного режима выполняется так:

```
>CMD.EXE /E:OFF.␣
```

Справку о работе команд в расширенном режиме можно получить, запустив команду в режиме справки, предварительно переключившись в расширенный режим, или с помощью команды HELP <команда> в обычном режиме.

Практика.

Ниже приводятся задания, которые необходимо выполнить для закрепления материала по работе в режиме командной строки.

1. Запустите командный интерпретатор COMMAND.COM.
2. Запустите команду DIR.
3. Получите справку о команде DIR.
4. Получите справку о команде SET.
5. Получите список команд для командного интерпретатора.
6. Завершите выполнение COMMAND.COM.
7. Запустите командный интерпретатор CMD.EXE.
8. Получите справку обо всех командах CMD.
9. Получите справку о команде SET.
10. Переключитесь в расширенный режим работы CMD.
11. Получите справку о команде SET.
12. Сравните все полученные справки о команде SET.
13. Отключите расширенный режим.
14. Посредством системного меню ("-" – левый верхний угол окна, правая кнопка мыши) вызовите окно настройки свойств режима командной строки. Познакомьтесь с ними.
15. Включите возможность выделения мышью текста и увеличьте размер буфера экрана до 300 строк.
16. Попробуйте выполнения разных команд из справочника.
17. Завершите выполнение CMD.EXE.

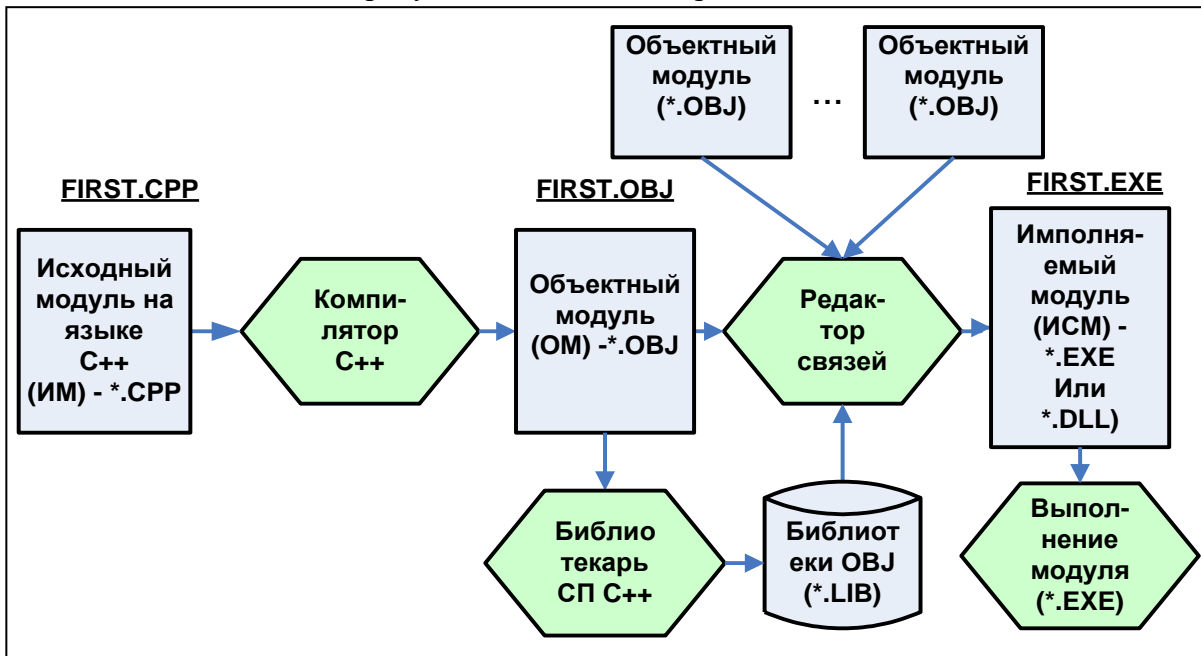
Литература по разделу:

1. Лекции по дисциплине "Операционные системы".
2. Электронные справочники - см. ЛР № 1.
3. **В.Э.Фигурнов** "IBM PC для пользователя" - М., Р и С, 1990 - 1997 г.

12. Технология создания исполнимых программ

Рассмотрим технологию создания исполнимого модуля в современных системах программирования. Процессы преобразования программ можно упрощенно представить следующим рисунком (см. ниже). На этом рисунке представлены основные компоненты систем программирования, которые участвуют в обработке программ. Кроме того, на рисунке выделены основные виды модулей и файлов, формируемых при такой обработке.

Данная технология формирования программ характерна практически для всех систем программирования, используемых в настоящее время. В тех случаях, когда мы работаем в интегрированной оболочке СП, мы можем в явном виде не увидеть промежуточных этапов и компонент, участвующих в такой обработке, однако они всегда присутствуют. Эти компоненты отмечены на рисунке двойным подчеркиванием.



12.1. Модули представления программ

Существующая технология создания исполнимых программ включает в себя следующие разновидности модулей:

- Исходные модули (ИМ), написанные на языке программирования (например, языке С++). Исходные модули, обычно, представляют собой длинную строку символов (или совокупность строк) и имеют следующее расширение файлов: ***.C**, ***.CPP**, ***.H**, ***.HPP**. Исходные модули формируются программистом или подключаются из библиотек (заголовочные файлы – **header file**).
- Объектные модули (ОМ) формируются компиляторами после успешной компиляции (без ошибок) исходных модулей. Объектные модули имеют расширение *.OBJ и имеют стандартизованную структуру. Изменять содержание объектных модулей вручную не рекомендуется.
- Исполнимые модули (ИСМ) формируются редактором связей (компоновщиком), который объединяет множество объектных модулей в единую программу. Исполнимые модули могут непосредственно выполняться на компьютере. Другая разновидность исполнимых модулей – динамические библиотеки (DLL – Dynamic Link Library), могут подключаться к основному исполняемому модулю на этапе выполнения программы. Такая технология поддерживается в среде Windows. Динамические библиотеки содержат множества

функций, которые вызываются из основной программы или других DLL. Динамические библиотеки могут подключаться или отключаться во время работы программы.

12.2. Компоненты и стадии обработки программ

Система программирования (СП) это большой комплекс программ, включающий в себя следующие основные программы:

- Компилятор (компиляция) системы программирования, который проверяет правильность написания программ (исходных модулей) и формирует объектные модули.
- Редактор связей (или компоновщик) необходим для объединения множества объектных модулей в единую программу (исполнимый модуль). Редактор связей настраивает связи между отдельными модулями, которые могут быть двух видов: связи по управлению (вызов функций и процедур) и связи по данным (использование переменных одного модуля в другом).
- Библиотекарь – программа СП (иногда называют такие программы утилита), которая позволяет создавать библиотеки объектных модулей (ОМ). Библиотеки объектных модулей подключают в программные проекты и, тем самым, обеспечивают подключение нужных объектных модулей в исполнимый модуль.
- Менеджер проектов (MAKE), позволяющий создавать проекты, выполнять сборку из многомодульных программ и проводить их отладку.
- Отладчик СП, который обеспечивает возможности эффективной проверки программ и исправления ошибок в программах.
- Другие сервисные утилиты предназначены для упрощения процесса программирования и обслуживания создаваемых проектов: текстовые редакторы, справочные системы, примеры использования разных технологий и т.д.

В современных системах программирования большинство компонент, которые необходимо часто использовать, объединены в единую программу, которая называется интегрированной оболочкой СП (IDE). Такая оболочка объединяет в себя: текстовый редактор, компилятор, компоновщик, менеджер проектов и отладчик. Примером удобной Интегрированной оболочки СП является MS VS 2005 и выше. В рамках дисциплины ПКШ при выполнении ЛР и ДЗ необходимо создавать проекты в оболочке. Рассмотрим процесс создания проектов.

12.3. Создание консольного проекта в VS 2005 (DZ_XXXXX_XDD).

Для создания консольного проекта *необходимо*:

- Запустить систему программирования VS 2005;
- 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: DZ_XXXXX_XDD (где: XXXXX - часть шифра разработки, X – номер группы, а DD – номер варианта по журналу группы текущего семестра. Например, для студента группы ИУ5-22 с вариантом 15 и темой Street – введем – DZ_Street_215). 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 выключен, **“Add common header files”**-> **“MFC”** и **“ATL”** должны быть включены.

- Далее необходимо нажать кнопку **“Finish”**. Новый проект будет создан.
- Необходимо убрать из главных моделей проекта (DZ_Street_XDD.CPP и DZ_Street_XDD.H) все лишнее. Этого: в файле DZ_XXXXX_XDD.H (у нас в примере DZ_Street_215.H) уберем все, а в файле DZ_XXXXX_XDD.CPP (DZ_Street_215.CPP) оставим следующий текст:

```
#include "stdafx.h"
#include "DZ_Street_215.h"
#include <iostream>
using namespace std;
void main(void)
{ ... }
```

- Для контроля правильности создания пустого проекта, нажмем клавишу **“F7”** для проверки создания программы (build) и **“F5”** для проверки ее выполнения (run/debug). Все перечисленные действия должны быть выполнены безошибочно.

12.4. Обеспечить русификацию консольного ввода и вывода.

Для корректного отображения текстов на русском языке в консольных проектах VS 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 . . .

Обратите внимание на то, что при другом способе локализации (*setlocale(0, "rus");*) не все 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.

13. Формальное описание синтаксиса в БНФ

Во многих учебниках, пособиях, электронных справочниках и документации Вы можете встретить формальное описание синтаксиса языков программирования и правил запуска программ. Такое описание необходимо для однозначного понимания правил записи операторов и команд, исключения ошибок и, в конечном счете, снижения сроков освоения языков и написания программ. Существует много различных языков формального описания других, извините за тавтологию, формальных языков, в частности языков программирования. Они делятся на две группы: текстовые и графические. В графических языках используются специальные диаграммы и правила их построения (пример графического языка для описания Ассемблера Вы найдете в учебнике Юрова).

Более четкое описание можно получить на основе текстовых формальных языков. Одним из примеров распространенных языков формального описания является БНФ. Данная аббревиатура имеет две расшифровки: Бекуса Нормальная Форма (более ранняя) и Бекуса Наура Форма (более поздняя). Разные названия связаны с тем, что второй автор (Наура) внес уточнения в первоначальные правила. В этом пособии мы рассмотрим основные правила применения БНФ для оформления документации ЛР и изучения Ассемблера.

13.1. Назначение и состав языка БНФ

Язык БНФ является распространенным языком описания формализованных конструкций других языков. Такие языки называют также метаязыками. Можно дать математическое описание этого языка на основе теории множеств и теории формальных языков, но здесь мы остановимся на более простом смысловом и текстовом его описании. Язык БНФ включает в себя следующие основные элементы и понятия:

- Понятие терминального символа и множества терминальных символов. Под терминальным символом понимается такой элемент описания, который буквально входит в синтаксические конструкции (предложения) языка. Эти символы не могут быть раскрыты далее в виде правил. Примером терминальных символов могут служить: знаки операций (“+”, “-” и т.д.), ключевые слова языка (for, loop, case и др.) и служебные символы (“=”, “;” и т.д.). Отметим, что любое правило должно быть раскрыто (порождать), в конечном счете, цепочку терминальных символов.
- Понятие нетерминального символа и множества нетерминальных символов. Под нетерминальными символами мы понимаем такой элемент языка, который представляет собой некоторое понятие языка и подлежит раскрытию с помощью специальных правил. Нетерминальные символы называют также переменными языка. При конкретном порождении нетерминальные символы должны, с помощью правил, быть раскрыты в виде терминальных символов. Примером нетерминального символа является понятие идентификатора (или имени переменной). По существующим правилам записи идентификаторов мы можем построить очень большое число различных вариантов имен переменных. Другим примером нетерминального символа является понятие целого числа или числа вообще, которое можно использовать в виде констант при написании операторов.
- Понятие правил грамматики языка и множества этих взаимосвязанных правил, составляющих, по сути, саму грамматику формального языка. Правило записывается с помощью метасимволов и предназначено для раскрытия нетерминальных символов с помощью комбинации других нетерминальных символов и терминальных символов. Предполагается, что существует некоторое первоначальное правило, с которого должно проводиться описание грамматики и, что более важно, грамматический разбор конкретной конструкции.

- Множество **метасимволов** языка БНФ, которые используются для записи правил грамматики описываемого формального языка. Это символы: “:=”, “<”, “>”, “[”, “]”, “{”, “}”, “...”. Более подробно о назначении метасимволов мы поясним ниже. Отметим также, что некоторые символы (а именно, “[”, “]”, “{”, “}”, “...”) входят в расширенный вариант БНФ и не являются обязательными.

13.2. Правила, нетерминальные переменные и метасимволы

Нетерминальный символ на языке БНФ записывается следующим образом: в угловые кавычки (<...>) помещается произвольный текст, который используется для обозначения понятия языка. Нетерминальный символ и его обозначение должны быть уникальными в пределах описываемой грамматики, то есть не повторяться. Примерами нетерминальных символов могут служить следующие понятия:

<Переменная> - понятие, описывающее переменную в программе,

<Оператор> - понятие, описывающее оператор в языке программирования.

Правило грамматики записывается в следующем виде (правила похожи на операторы присваивания, но нельзя их путать):

<Нетерминальный символ>:= <Выражение на основе терминальных символов, метасимволов и нетерминальных символов>

Для пояснения самого языка БНФ нам приходится использовать также язык БНФ. В левой части правила должен быть расположен только один нетерминальный символ. Выражение является комбинацией терминальных и нетерминальных символов, а также метасимвола “|”, который позволяет упростить написание правил и, в конечном счете, не является обязательным. Так, например, два правила вида (где, НС – нетерминальный символ):

<НС1>:= <НС2> и

<НС1>:= <НС3> можно объединить в правило вида:

<НС1>:= <НС2> / <НС3> , где НС1 ÷ НС3 нетерминальные символы.

Правила на языке БДН могут быть рекурсивными, это означает, что один и тот же нетерминальный символ, задаваемый правилом, может присутствовать в правом выражении этого же правила. Например:

<НС1>:= <НС2> | <НС2> <НС1>

В выражении правил могут располагаться терминальные символы (ТС). Они могут располагаться в любом порядке. Например:

<НС1>:= ТС1 <НС2> ТС2 ТС3 <НС3> ТС4

При описании конструкций на языке Ассемблера возникает проблема с использованием угловых скобок, так как они являются также терминальными символами языка. В этом случае, либо делается словесное пояснение, либо эти символы выделяются жирным шрифтом, либо заключаются в круглые кавычки “(“ - ”)”. Например:

<Описание записи>:= <Имя записи> <Шаблон записи> (<) <Инициализация записи> (>)

Заключенные в круглые скобки символы рассматриваются как терминальные символы языка. Для упрощения записи правил могут дополнительно использоваться квадратные скобки - “[”, “]”. Применение их означает, что заключенное в них подвыражение может отсутствовать. Например, правил вида:

<НС1>:= <НС2> [<НС1>]

Заменяет более сложное правило вида:

<НС1>:= <НС2> | <НС2> <НС1>

При перечислении повторов можно также использовать и фигурные скобки метасимволы - “{”, “}”, если это не усложняет запись правила. Так, например, правило вида:

<НС1>:= {<НС2> , } ...

Означает сокращенную форму записи правила вида:

<НС1>:= <НС2> | <НС2> , <НС2> | <НС2> , <НС2> , <НС2> и так далее.

Если обратиться к строгой записи на языке БНФ, то использование круглых и фигурных скобок можно исключить. В документациях на многие программные продукты Вы найдете описания на этом языке, включая и расширенное его толкование.

При формальном описании на языке БНФ необходимо помимо формальных правил давать текстовые пояснения основных понятий – нетерминальных символов. Это выполняется в текстовом режиме.

13.3. Примеры описания на БНФ

Рассмотрим несколько примеров описаний на языке БНФ.

Пример 1. Описание в БНФ синтаксиса целого числа со знаком:

```
<целое число со знаком>:= <целое число> | + <целое число> | - <целое
число>
<целое число>:= <цифра> | <цифра> <целое число>
<цифра>:= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Таким синтаксическим правилам соответствуют числа: +15, -35, 01, 12345678788 и т.д. Отметим, что для написания программ, недостаточно одних синтаксических правил, нужно еще дополнительное семантическое (понятийное или смысловое) описание языка (семантические правила). Даже в этом простом примере нужно указать максимальное число знаков целой константы, которое допустимо в конкретной системе программирования. Если данная константа соответствует типу int, то для записи ее можно указать только 5 значащих цифр ($2^{16} = 65536$).

В нашем примере нетерминальными символами являются: <целое число со знаком>, <целое число> и <цифра>. Терминальными символами являются: “+”, “-”, “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”.

Отметим также, что один и тот же синтаксис может быть описан разными способами. Так первое правило может быть описано так:

```
<целое число со знаком>:= <знак> <целое число>
<знак>:= <пусто> / + / -
```

Где <пусто> часто используемый для описания правил нетерминальный (фактически терминальный) символ, означающий пустую строку (отметим – не пробел, который является значащим терминальным символом. Если при описании правил требуется указать терминальный символ – пробел, то его обозначают специальным символом, например, – “_”).

Пример 2. Инструкция командной строки для запуска программ.

```
<запуск программы>:= <имя программы> _ <список параметров программы>
<имя программы>:= progr.exe | progr
<список параметров программы>:= <Параметр 1>[_ <Параметр 2 >] [_ <Параметр
3 >]
<Параметр 1>:= <Имя файла>
<Параметр 2>:= /L
<Параметр 3>:= /P:<Пароль>
```

Отметим, что и здесь необходимо пояснить смысловое (семантика правил) значение правил синтаксиса. Так <Имя файла> - например, стандартный текстовый файл операционной системы (в некоторых случаях не нужно далее раскрывать значение нетерминального символа, сославшись на общепринятые правила в операционной системе, но можно и раскрыть). Нужно раскрыть также, что означает использование параметра 2 (/L), например установку специального режима работы программы. Для параметра 3 нужно пояснить число символов, которые нужно ввести для пароля. Символ “_” в нашем описании обозначает пробел, а параметры 2 и 3 могут быть опущены при запуске. Если допускается изменять последовательность параметров (пусть 2 и 3), то нужно в описание грамматики запуска добавить дополнительное правило вида:

<список параметров программы>:= <Параметр 1>[_<Параметр 3>] [_<Параметр 2>]

Для описания инструкций командной строки во 2-й, 6-й лабораторных работах и в курсовой работе необходимо использовать подобное описание.

Пример 3. Рассмотрим также для примера описание команды IF в формате БНФ. Во-первых, не будем учитывать возможности расширенного режима работы CMD, и, кроме того, сократим при этом немного синтаксис и семантику описания.

```

<Оператор IF>:= IF_ [NOT] <Варианты конструкции оператора IF>
<Варианты конструкции оператора IF>:= <конструкции ERRORLEVEL> | <кон-
струкции со строками> | <конструкции EXIST>
<конструкции ERRORLEVEL>:= ERRORLEVEL_<число>_<команда>
<конструкции со строками>:= <строка>==< строка >_<команда>
<конструкции EXIST>:= EXIST_<имя файла>_<команда>
<строка>:=<любая последовательность символов до пробела > | <переменная>
| %<переменная>% | <параметр командной строки>
<параметр командной строки>:=%<десятичная цифра>
<десятичная цифра>:=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
9

```

Где:

<команда> - любая команда языка командных файлов, исключая IF. Данная команда выполняется в случае, если условие, заданное в операторе IF, является истинным.

<число> - целое число без знака, которое сравнивается со значением переменной ERRORLEVEL, определяющий код возврата последней выполненной в командном файле программы или встроенной команды. Между этими значениями подразумевается знак отношения (>=).

<имя файла> - стандартное имя файла операционной системы, существование которого проверяется в условии EXIST. Условие считается истинным, если файл найден в пределах текущей директории или по заданному пути (PATH).

<переменная> - любая текстовая переменная командного файла, включая и системные переменные окружения установленные командой SET. Если не указаны символы процентов (%), то имя переменной рассматривается как строка, а противном случае рассматривается значение этой переменной.

<любая последовательность символов до пробела> - любая последовательность символов, которая ограничивается пробелом. Необходимо следить, чтобы в группу символов случайно не попадали служебные слова.

Примеры использования оператора командных файлов IF:

1. IF ERRORLEVEL 3 GOTO **MET1** – переход на метку MET1, если ERRORLEVEL >= 3
2. IF (%1) == () SET VAR=**NODOSTUP** – переменной VAR присваивается значение NODOSTUP если первый параметр не задан.
3. IF NOT EXIST **FIRST.EXE** GOTO **FINISH** – переход на метку FINISH, если файла FIRST.EXE не существует.

Практика.

1. Опишите в БНФ синтаксис вещественного числа с порядком (Пример числа: 12.234 E +10).
2. Опишите в БНФ синтаксис запуска командного файла для 2-й ЛР (обязательное требование к ЛР).
3. Опишите в БНФ синтаксис запуска программы для 6-й ЛР (обязательное требование к ЛР).
4. Опишите в БНФ синтаксис запуска резидента для КР (обязательное требование к КР).
5. Опишите в БНФ синтаксис оператора IF для расширенного режима CMD.

14. Работа с интегрированными файловыми менеджерами

Вы наверно уже поняли, что при работе в режиме командной строки приходится вводить много команд и выполнять операций переключения между каталогами, дисками. При работе в среде Windows также приходится выполнять аналогичные действия, но в этом случае Вы можете воспользоваться стандартным приложением – Windows Explorer. И в первом и во втором случае работать не очень удобно, трудоемко и возможны ошибки при выполнении операций над файлами. Для большего удобства предусмотрены специальные программы, которые называются файловыми менеджерами. Эти программы обеспечивают выполнение множества полезных операций и очень удобны в работе. Надеюсь, что даже при первом знакомстве Вы в этом сможете убедиться.

Разновидностей файловых менеджеров много. Они обеспечивают работу как в режиме эмуляции ДОС, так и в среде WINDOWS. Перечислим основные возможности файловых менеджеров:

- Управление файлами (копирование, перемещение, поиск и многие другие);
- Управление программами (запуск, настройка и т.д);
- Редактирование текстовых файлов;
- Сортировка файлов;
- Поиск файлов и информации в файлах;
- И многие другие возможности.

В данном разделе мы кратко познакомимся с тремя такими программами: *DosNavigator*, *FarManager* и. Кроме этих вариантов Вы можете встретить и другие файловые менеджеры: *Norton Commander*, *Volkov Commander* и другие. Функционально и по интерфейсу эти программы очень похожи друг на друга. Самое существенное отличие – это набор функциональных клавиш, которые используются для управления.

Общим для всех файловых менеджеров является то, что в окне программы обычно представлены две панели (два подокна), одна из которых является активной. Активность обычно выделяется цветом. Переключение между панелями осуществляется клавишей **TAB**. В активном окне выделен блочный курсор (выделяется цветом или рамкой). Не нужно путать этот курсор с курсором мыши. Блочный курсор выделяет объект (файл), над которым могут быть выполнены операции (копирования, перемещения, удаления и т.д.). Подсказка о возможных операциях (функциональных клавишах) обычно помещается в нижней строке окна файлового менеджера.

Второй особенностью файлового менеджера, является наличие командной строки, в которой можно вводить команды и запускать программы. Кстати, обычно запуск программы может быть выполнен простым нажатием клавиши ENTER, после выделения нужной программы с помощью блочного курсора. Командная строка аналогична командной строке в режиме эмуляции ДОС, под управлением командного процессора. Если файловый менеджер работает в среде WINDOWS, командная строка соответствует режиму ввода команд в меню пуск: ПУСК/Start=> Выполнить/Run => Ввод команды.

14.1. Dos Navigator

Окно файлового менеджера *Dos Navigator* (есть на сайте) представлено ниже. На левой панели представлен каталог (директория, папка) диска (c:)- c:\QC25\BIN. На правой панели каталог диска (i:).



Для выхода в меню можно использовать клавише **F10**. Для завершения программы необходимо воспользоваться клавишами **Alt+X**. Назначение функциональных клавиш для выполнения основных операций показано в нижней строке. При выполнении операций копирования и перемещения файлы будут копироваться из активной панели менеджера в пассивную панель. Вторая строка снизу представляет собой командную строку для ввода команд. Программа позволяет делать множество специальных настроек, которые значительно упрощают работу пользователя. Например, можно создать пользовательское меню (**F2**) для быстрого вызова нужных программ и выполнения команд.

14.2. Far manager

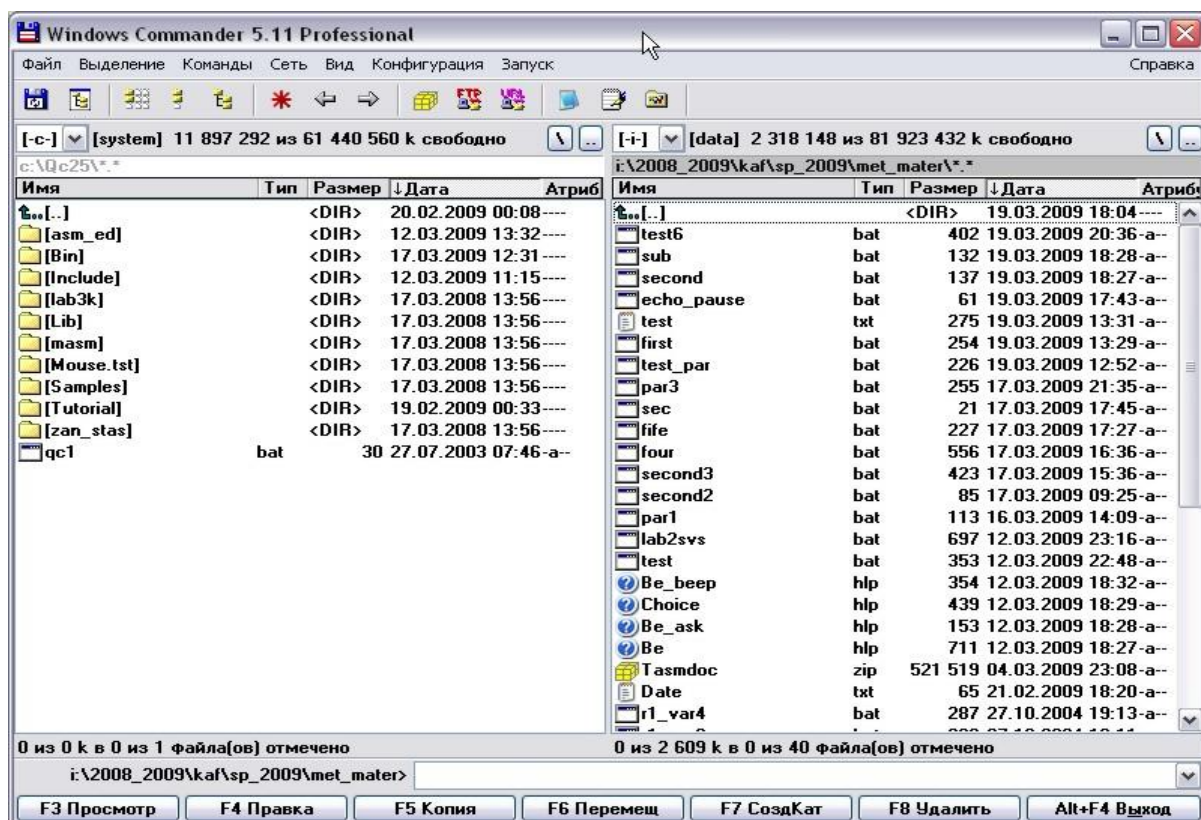
Окно файлового менеджера *Far manager* (есть на сайте) представлено ниже. На левой панели представлен каталог (директория, папка) диска (c:)- c:\QC25. На правой панели каталог диска (i:).



Завершение работы FAR выполняется клавишей **F10**. Для запуска меню используется **F9**. После нескольких сеансов работы в файл менеджером FAR Вы, несомненно, освоитесь и будете его использовать в своей работе.

14.3. Windows/Total Commander

Для работы в среде WINDOWS можно использовать *Windows Commander* или *Total Commander* (синонимы). Окно этой программы показано на рисунке ниже.



Достаточно внимательно посмотреть, чтобы увидеть сходство с другими файловыми менеджерами. Особенностью Windows Commander по сравнению с другими рассмотренными менеджерами является более активное использование мыши. Кроме того здесь соблюдены все стандарты Windows программ (меню и клавиатура).

15. Разработка блок-схем программ

Важнейшим этапом разработки программ и программных систем является формализация процесса выполнения операций (шагов, действий, вложенных процессов и т.д.). Этот этап также называют разработкой алгоритма программы. Для такой формализации (формализованного описания) должны быть использованы формализованные языки, не содержащие неоднозначностей. Существует много подобных графических языков, но, на мой взгляд, самым удачным является формализованный язык блок-схем.

Нужно иметь в виду, что этот язык подходит не для всех описаний используемых при разработке программного обеспечения, а в первую очередь используется для описания последовательности действий и команд. Например, для описания систем классов применяется язык диаграмм классов, для описания структур баз данных графический язык сущностей и связей. Однако, если Вам необходимо разработать метод класса (процедуру, функцию и т.д.) даже средней сложности, то при разработке ее и ее отладки обойтись без блок-схемы. Я уже не говорю о документировании программного обеспечения, сохранения информации о проекте для последующего сопровождения и модернизации.

Для лабораторных работ по дисциплине Системное программирование, при разработке командных файлов и программ на языке Ассемблер, целесообразно (и требуется) использование блок-схем как основного языка для описания алгоритмов. Поэтому, я в данное пособие включил этот раздел.

Язык блок-схем является графическим языком описания алгоритмов. В общем случае, блок-схема это ориентированный граф из специальных элементов - блоков (вершины) и связей (стрелки и линии).

15.1. Назначение блок-схем программ

Блок-схемы (я буду использовать название с дефисом, хотя допустимо и другое) позволяют обеспечить следующее:

- Создать формализованное описание алгоритма работы разрабатываемой программы.
- На смысловом уровне и на логическом уровне проверить правильность описанного алгоритма.
- Вручную (по шагам) выполнить создаваемую программу и проверить корректность выполняемых действий и получаемого результата.
- Служить основой для создания программы на конкретном языке программирования. Обращаю Ваше внимание, что только в этот момент может быть принято решение о выборе языка программирования, причем именно характер блок-схемы, или ее вид, позволяет сделать такой выбор более точно.
- Служить подсказкой при отладке программ с помощью универсальных отладчиков.
- Не исключено, что по написанной программе может быть повторно построена блок-схема, а ее можно сравнить с блок-схемой, построенной на этапе проектирования. Это сравнение может привести к следующим выводам: программа построена неверно, блок схема построена неверно или, что не учтены важные особенности разработанного алгоритма.

Думаю, что данный перечень может быть продолжен и далее, а здесь я отмечу, что блок-схема программы позволяет представить программу и алгоритм в целом, в виде единого образа, по которому можно дать ее оценку: плохая программа или хорошая. Например, если в блок-схеме много связей (в том числе и избыточных), то, возможно, следует подумать о том, чтобы использовать процедуры и, тем самым, упростить понимание алгоритма и написанной программы. Считаю, что программист, который не умеет правильно

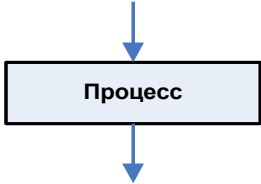
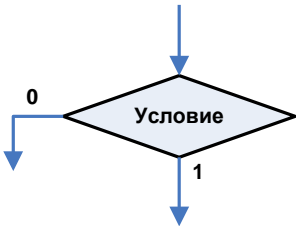
оформлять блок-схемы и не использует их в своей работе, не может претендовать на звание профессионального программиста!

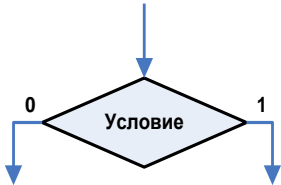
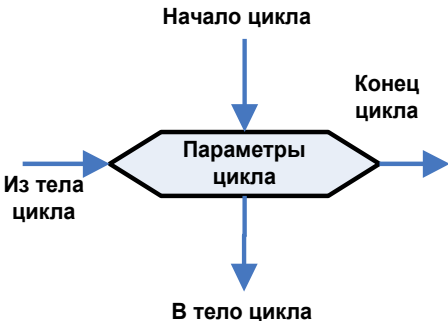
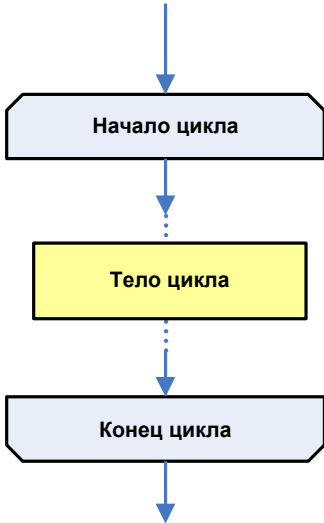
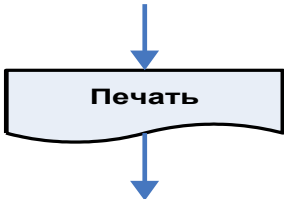
В зависимости от этапа разработки алгоритмов программ, блок-схемы могут иметь разную степень детальности. Более того, технология разработки блок-схем должна предусматривать последовательное раскрытие деталей алгоритма, постепенное уточнение действий и операций, выполняемых для решения задачи. Самый детальный способ описания программы – это описание на уровне команд или операций языка. Хотя этого не всегда нужно, тем не менее, это возможный и самый низкий уровень описания, к которому нужно стремиться, в частности при документировании программ.

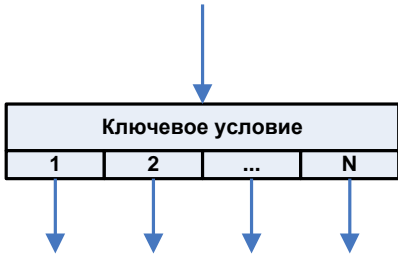
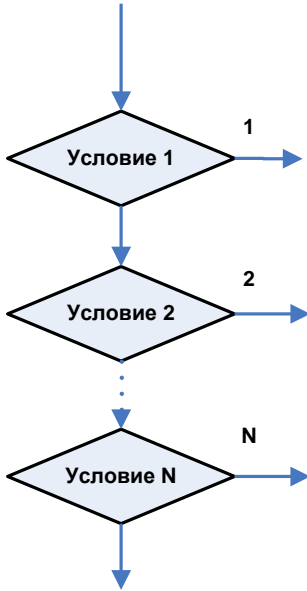
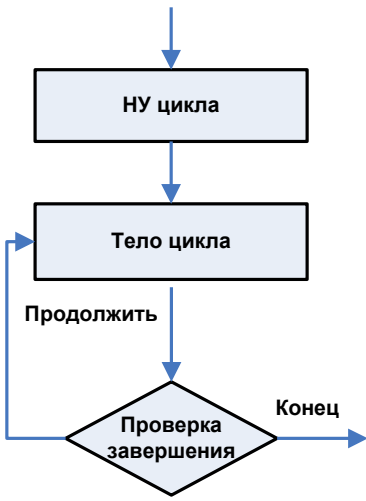
Отмечу еще одну особенность блок-схем, которая иногда приводит к недоумению.

15.2. Элементы блок-схем программ

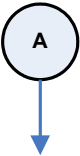
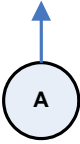
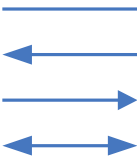
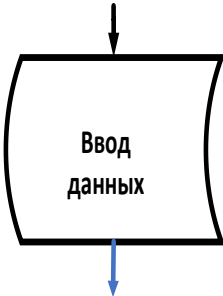
Блок-схемы, как было уже отмечено, представляют собой ориентированный граф, в вершинах которого размещаются специфические элементы. Каждый элемент имеет особую структуру, свое назначение и обозначение. Самое существенное отличие элементов – число входящих и выходящих стрелок. Логика работы элемента блок-схемы в основном и определяется их числом. Основные элементы (их обозначения и назначение) представлены в таблице расположенной ниже.

№	Обозначение элемента	Назначение	Примечание
1.		Элемент типа процесс описывает последовательность действий и операций. Операции выполняются одна за другой без условий. Имеет одну входящую стрелку и одну выходящую.	Используется как элемент последовательной детализации.
2.		Элемент ветвления, типа условие. Если условие при проверке получает значение истина (1, ДА, TRUE), то выполняется переход по стрелке помеченной 1. Если условие получает на конкретном шаге значение ложь (0, НЕТ, FALSE), то выполняется переход по стрелке помеченной 0.	Можно использовать обозначения стрелок (1, ДА, Истина, TRUE) или (0, НЕТ, ЛОЖЬ, FALSE) соответственно.

№	Обозначение элемента	Назначение	Примечание
3.		<p>Это вариант конструкции условие. Описание аналогично. Возможны и другие варианты стрелок. Самое главное: одна входит, а две выходят.</p>	
4.		<p>Данный элемент, элемент описания циклов, имеет две входящие стрелки и две выходящие стрелки. Назначение стрелок показано на рисунке. В поле данного блока записываются условия продолжения цикла и изменяемые на каждом его шаге параметры.</p>	<p>Данный элемент рекомендуется для оформления блок-схем лабораторных работ.</p>
5.		<p>Другой элемент для организации циклических фрагментов программ. Он соответствует ГОСТу. Этот элемент состоит из двух частей, в первой записываются начальные условия цикла, а во второй условия его продолжения.</p>	<p>При организации вложенных циклов не всегда является наглядным. Элемент не является обязательным для ЛР.</p>
6.		<p>Элемент для обозначения вывода информации, в частности на печать. Данный элемент похож на элемент типа процесс и во многих случаях может быть им заменен.</p>	<p>В ЛР лучше использовать процесс.</p>

№	Обозначение элемента	Назначение	Примечание
7.		<p>Элемент типа переключатель. Имеет один вход и множество выходов. По ключевому условию определяется (вычисляется) номер выхода, по которому производится передача управления.</p>	<p>Рекомендуется для выполнения ЛР по курсу.</p>
8.		<p>Это другая форма блока переключателя. Но есть и отличия, например, условия, могут быть разными и число выходов больше на единицу, так как предусмотрен вариант, когда ни одно условие не сработало (по умолчанию - default).</p>	<p>Применять разновидности блоков переключателя желательно на основе логики работы программы.</p>
9.		<p>Здесь представлена конструкция, которая не является отдельным элементом блок-схемы, а состоит из трех важных блоков, которые должны присутствовать в любом цикле: задание начальных условий цикла (НУ), тело цикла, проверка условия завершения/продолжения цикла.</p>	<p>Эту простейшую, но обобщенную блок-схему я включил для более четкого понимания циклов. Такую конструкцию можно использовать с оператором IF, if.</p>
10.		<p>Здесь также представлена конструкция, которая не является отдельным</p>	<p>Эту простейшую, но обобщенную блок-схему я</p>

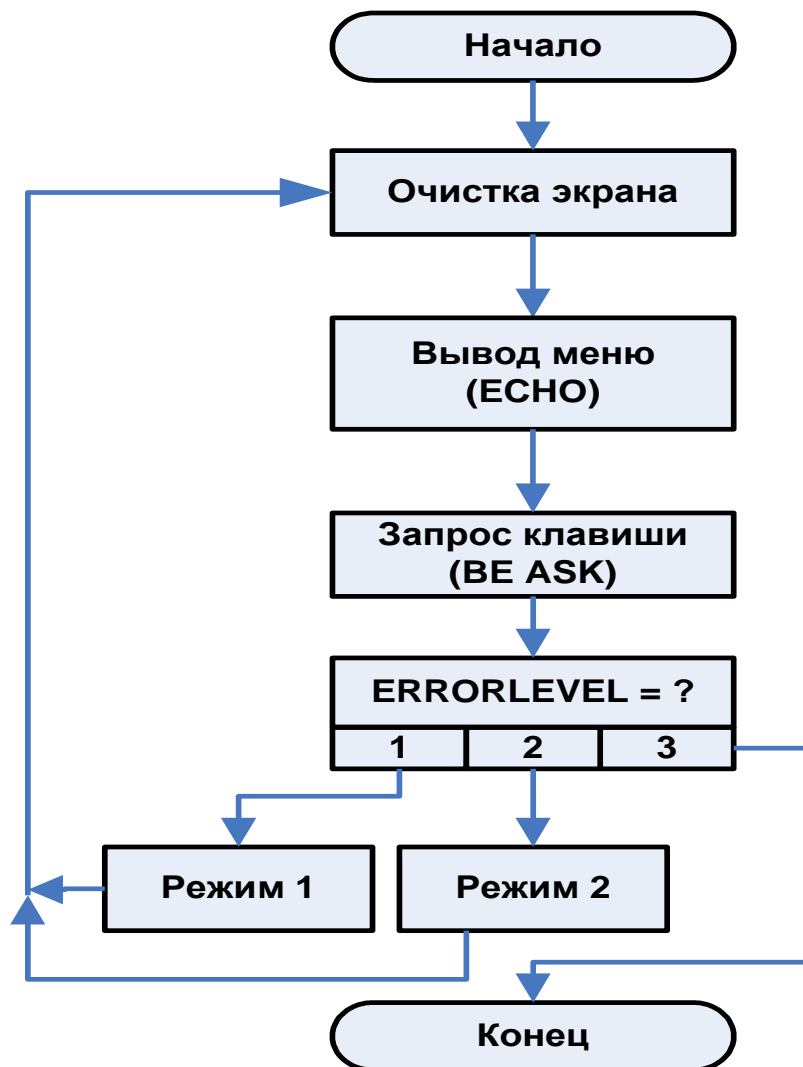
<i>№</i>	<i>Обозначение элемента</i>	<i>Назначение</i>	<i>Примечание</i>
		<p>элементом блок-схемы, а состоит из трех важных блоков, аналогичных предыдущей конструкции. Отличие заключается в том, что теоретически возможен цикл без единого выполнения тела цикла (Сравни конструкции DO и WHILE).</p>	<p>включил для более четкого понимания циклов. Такую конструкцию можно использовать с оператором IF, if.</p>
11.		<p>Такой элемент используется для выделения в блок-схемах участков вызова процедур (функций, методов, подпрограмм и т.д.). Двойная стрелка показывает то, что выполняется возврат из процедуры.</p>	<p>Необязательный элемент блок-схем. В поля блока записывается название процедуры.</p>
12.		<p>Данный элемент определяет начало алгоритма, начало выполнения программы. Чаще всего в одной блок-схеме такой элемент один. Имеет одну входящую стрелку.</p>	<p>Для программ с несколькими входами начальные элементы должны быть помечены.</p>
13.		<p>Данный элемент определяет конец алгоритма, конец выполнения программы. Таких элементов может быть несколько, в зависимости от логики работы программы. Имеет одну входящую стрелку.</p>	<p>Элемент окончания может иметь один или несколько входов.</p>

№	Обозначение элемента	Назначение	Примечание
14.		Данный (и следующий) элемент используется для условного разрыва соединяющих линий. В поле элемента записывается обозначение (чаще буква или цифра). Это обозначение должно соответствовать обозначению другой точки разрыва.	
15.		Этот элемент является парным к предыдущему. Они позволяют организовать разрыв линий на одном листе или на разных листах.	
16.		Элементы в виде простых линий и линий со стрелками задают связи между блочными элементами блок-схем. Для линий без стрелок подразумевается стрелка: сверху - вниз и слева - направо. Линии с двойными стрелками используются для обозначения вызова процедур.	Линии могут быть логическими, маневровыми, только под прямым углом (ГОСТ). Предпочтительнее использовать линии со стрелками.
17.		Элемент для ввода данных извне (например с клавиатуры)	

15.3. Примеры блок-схем программ

В данном разделе мы рассмотрим несколько примеров блок-схем для иллюстрации использования элементов и приемов построения алгоритмов. Примеры простые и соответствуют программам, которые рассмотрены в других разделах настоящего пособия. Перед блок-схемой дана ссылка на раздел, в котором описана программа ей соответствующая. Если щелкнуть мышкой на ссылке (с клавишей Ctrl) можно оперативно перейти к тесту программы.

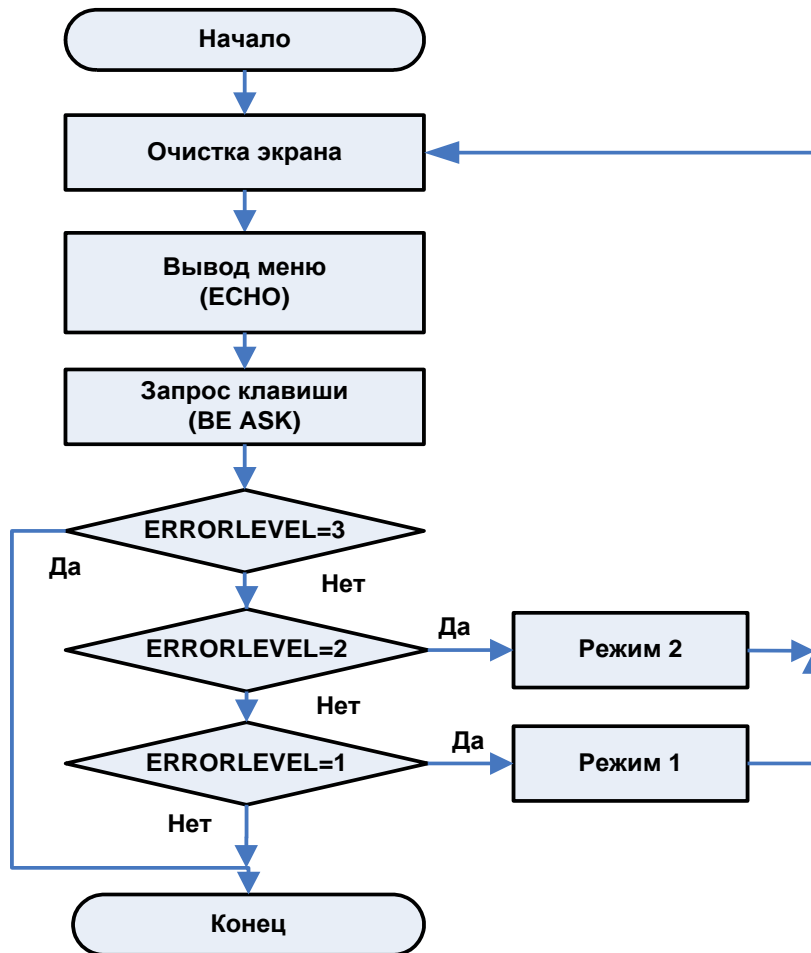
Пример блок-схемы программы для командного файла, рассмотренного в разделе **Ошибка! Источник ссылки не найден..** Данный командный файл соответствует программе, которую нужно разработать в ЛР номер 2. В блок-схеме этого примера продемонстрировано использование переключателя и элементов типа процесс. Отметим, что переключатель в



командном файле реализован с помощью директив IF, поэтому, строго говоря, блок-схема не совсем точна, хотя полностью отражает алгоритм программы. Причина неточности данной блок-схемы заключается в том, не ясно куда мы будем двигаться в том случае, когда будет получен результат $ERRORLEVEL < 1$. Вспомните, что сравнение выполняется по условию \geq . При использовании BE ASK допустимо назначить ключ возврата по умолчанию, например 3. Тогда программа завершится даже при невозможности задания нужного номера с клавиатуры (отличными от 1, 2, 3). При использовании CHOICE такое невозможно, так как ключа по умолчанию задать нельзя.

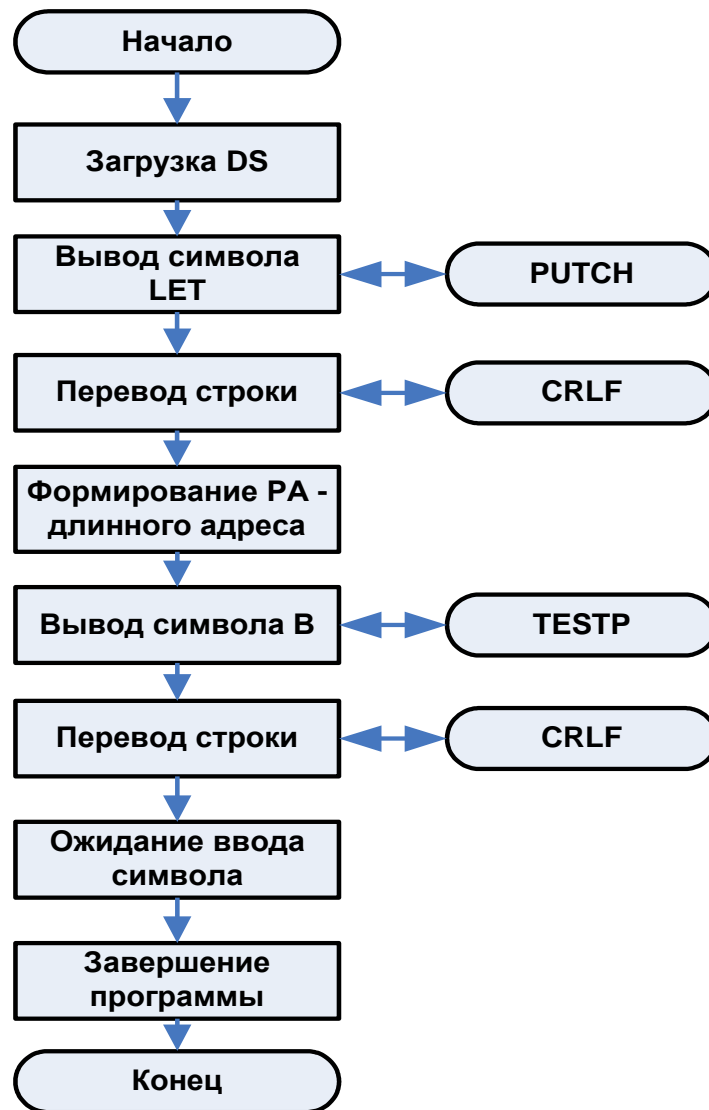
Когда выбран метод реализации алгоритма можно приступить к написанию программы. Если блок-схему для нашего командного файла построить после написания программы, то она может выглядеть так, как показано на следующем рисунке.

Эта блок-схема является более точной, так как может правильно работать и в том случае когда код возврата < 1 .



В этой схеме переключатель построен на основе второй конструкции.

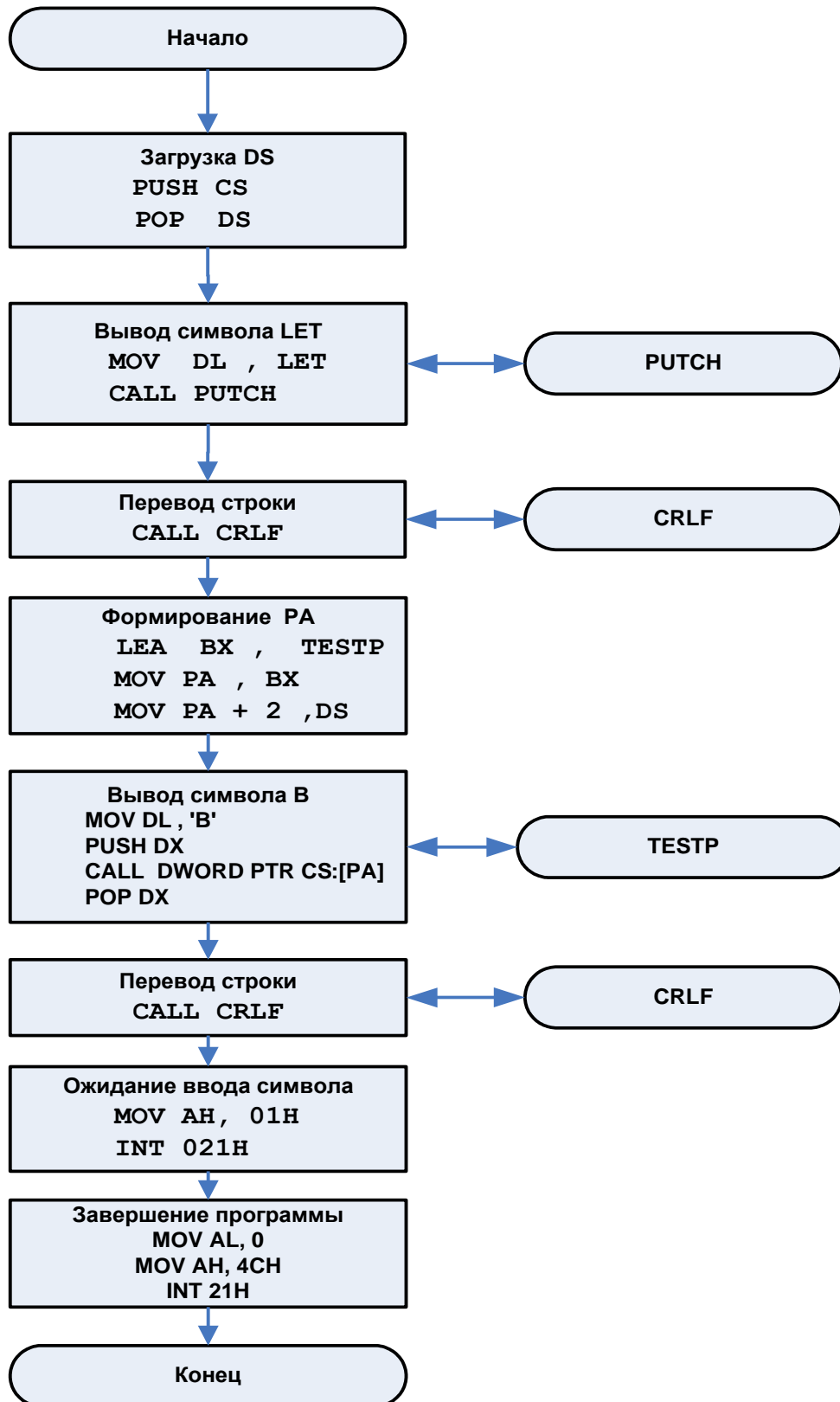
Пример блок-схемы алгоритма программы на языке Ассемблер с процедурами (см. раздел **Ошибка! Источник ссылки не найден.**). Следующая блок-схема показывает как и спользовать блоки вызова процедур.



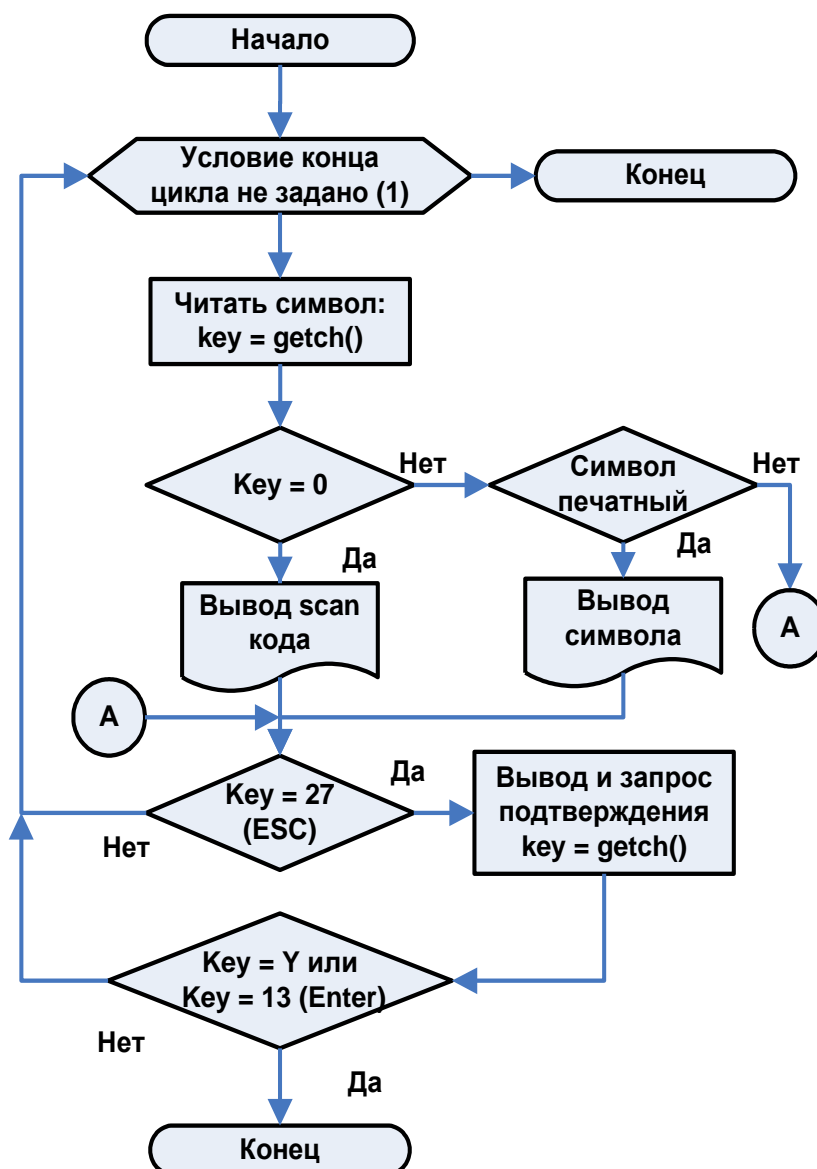
Для отдельных процедур нужно разработать отдельные блок-схемы и представить в отчете. В методических указаниях отмечено, что если Вы используете повторно одни и те же процедуры, то допускается не повторять в новых отчетах блок-схемы этих процедур. Программа для этой блок-схемы очень проста. Однако, блок-схема нужна для детального описания шагов, которые нужно выполнить для решения задачи.

В детальной блок-схеме мы можем расписать алгоритм работы программы вплоть до отдельной команды. Это позволяет процедуру кодирования сделать простой и формализованной. Возможно в группах разработчиков и такое: один программист-аналитик разрабатывает алгоритмы и оформляет детальные блок-схемы, а другие, причем менее квалифицированные выполняют кодирование и отладку программ. Такое разделение возможно, так как процесс создания алгоритмов решения задач более трудоемок, требует большего опыта и квалификации. Часто бывает так, что начинающие программисты занимаются кодированием программ до тех пор пока не получают достаточного опыта и знаний для самостоятельной разработки сложных алгоритмов программ.

Детальная блок-схема программы на Ассемблере для вывода 2-х символов приведена ниже.



Ниже представлена блок-схема процедуры для программы печати символов и скан кодов клавиш (см. раздел 16.7. Программы для получения списка кодов).



В этой программе в интерактивном режиме считываются коды нажатых клавиш, и производится их распечатка. Программа написана на языке СИ. В данном примере проиллюстрировано: использование циклической конструкции. Использование условных элементов и применение элементов для разрыва линий связи в блок-схеме.

В программе организован бесконечный цикл. Выход из цикла выполняется в теле цикла при проверке нажатия клавиши ESC (код 27) и подтверждения клавишами Enter или клавишей “Y”. При считывании кода клавиши проверяется первый байт. Значение первого байта указывает на ввод печатного символа или ввод скан кода ($key = 0$). Подробно о скан-кодах смотрите в разделе пособия 16.5. SCAN – коды.

Проверки в программе выполняются оператором *if* и условным выражением. Условное выражение (*isgraph(key) ? key : ' '*) вставлено в вызов функции печати – *printf*. Функция *isgraph* проверяет признак печатного символа и отображает символ (*key*) или пробел. Отмечу, что в блок-схеме эта логика отображается условным элементом, в программе вставлена в вызов функции. Кстати, это одна из причин того, почему не всегда программы сгенерированные автоматически являются эффективными и наглядными. После проверки завершения по ESC в программе выполняется запрос символа подтверждения и проверка его ввода. Для этого используются условный элемент блок-схемы.

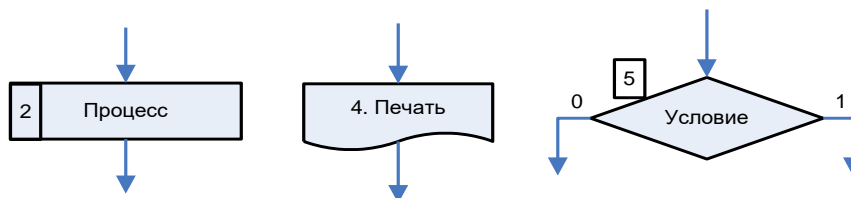
И последнее, с помощью элемента в кружочке (“А”) мы показываем, как сделать разрыв линии связи. Это позволяет не тянуть связующую линию через всю страницу и избежать пересечения линий. После выяснения того, является ли символ печатным или нет, мы должны перейти на проверку завершения цикла. Это делается с помощью добавления элемента с символом “А” в место проверки клавиши завершения.

15.4. Оформление блок-схемы программы

При оформлении блок-схем нужно соблюдать формальные, визуальные и семантические правила. Формальные правила задаются языком блок-схем и изображением его элементов. Визуальные правила позволяют сделать блок-схему более наглядной. Смысловые правила определяются алгоритмом конкретной задачи разработки программы. Перечислим основные правила:

- Все элементы должны иметь точно такое число входов и выходов, которое определяется изображением элемента.
- Должны существовать пути из начального элемента хотя бы в один конечный элемент.
- Не должно быть элементов типа процесс с одним входом и без выхода.
- Данные в блок-схемах не описываются.
- Из элемента типа процесс не может быть более одного выхода.
- Элементы блок-схемы должны занимать равномерно все пространство страницы или листа, на котором они располагаются.
- Элементы должны иметь приблизительно равные размеры и одинаковое форматирование.
- Расстояние между элементами должно быть по возможности одинаковым.
- Элементы и линии связи не должны располагаться очень близко друг от друга.
- Линии связи не должны пересекаться (только в порядке исключения).
- Если линии связей не имеют стрелок, то направление стрелки подразумевается так: слева направо и снизу вверх. Если возникают неоднозначности, то желательно использовать концевые стрелки.
- Блоки в блок-схеме желательно пронумеровать, в этом случае удобнее создавать описание блок-схем программ.
- Толщина линий связи должна быть одинакова.
- Блок-схема должна оформляться с помощью стандартных программ рисования, например MS VISIO или MS WORD.

Нумерация блоков может быть выполнена различными способами. На рисунке ниже показаны варианты такой нумерации.



Для описания блок схем очень удобно использовать программный продукт MS VISIO, причем можно воспользоваться любой версией программного продукта. Рисунки сделанные в VISIO можно легко вставить в документы MS, в частности MS WORD.

Примечание. После вставки рисунка желательно выполнить две операции:

- Выполнить форматирование объекта рисунка так, чтобы он раздвигал текст по горизонтали (выделить рисунок, формат объект =>Закладка Положение =>Кнопка дополнительно => Закладка Обтекание => Выбор Снизу и сверху => ОК =>ОК).

- Выполнить привязку рисунка к абзацу. Абзац лучше выбрать над рисунком. (Включить режим просмотра непечатных знаков - п, выделить рисунок, переместить якорь перед абзацем слева, формат объект =>Закладка Положение =>Кнопка дополнительно => Закладка Положение рисунка => Галочка установить привязку => ОК =>ОК). После этого на якоря появятся замочки. Рисунок будет перемещаться вместе с абзацем. Нужно рассчитать так, чтобы и рисунок и абзац помещались на одной странице документа.

Для рисования в VISIO нужно выбрать метрический шаблон объектов с названием “Basic Flowchart Shapes”. В этом шаблоне содержатся все необходимые объекты для рисования блок-схем.

15.5. Блок-схемы и описания данных

В блок-схемах не предусмотрено элементов для описания данных. Это объясняется тем, что блок-схема используется для описания действий (операторов), а описания переменных никаких действий в программе обычно не выполняют. Поэтому описание переменных, так называемая спецификация переменных программы производится отдельно. Лучше всего это сделать в отдельной таблице, в которой отображается: название и тип переменной, ее назначение и использование. Примером такой таблицы для программы вывода символов может быть следующая таблица:

<i>№</i>	<i>Название</i>	<i>Тип</i>	<i>Назначение</i>	<i>Где используется</i>
1.	LET	DB	Для вывода символа на экран	В программе и процедуре PUTCH

Попытка описания данных в лабораторных работах и рейтингах будет расцениваться как ошибка, и свидетельствовать о незнании правил написания блок схем и логики программ.

16. Коды их назначение и виды

Кодировка русских символов различается в разных режимах работы: работа в среде WINDOWS и работы в среде эмуляции ДОС. Так исторически сложилось, что эти кодировки не идентичны, а для нормального представления символов на экране приходится выполнять операции перекодирования. Кодировка в среде ДОС носит название ASCII (American Standard Code for Information Interchange – американский стандартный код для обмена информацией) в первоначальном варианте содержала всего 128 кодов (см. первую таблицу ниже). Затем появилась расширенная кодировка ASCII, которая была рассчитана уже на 256 символов (см. вторую таблицу ниже). В этой кодировке уже можно было представить символы других языков и символы псевдографики. В современных русифицированных программах эту кодировку Вы можете встретить также под названием “Кириллица ДОС/ Кириллица ДОС 2”. Кодировка в среде WINDOWS, в первую очередь вторая часть таблицы (128 - 255), имеет другое содержание и название - ANSI. Другое название ANSI – кодировка WINDOWS. В этой системе кодирования не используются символы псевдографики, так как вывод информации на экран выполняется в графическом режиме. Позиции русских символов находятся во второй половине таблицы (см. таблицы ниже) и имеют отличную кодировку от кода ASCII. Это создает много проблем при совместной работе в WINDOWS и режиме командной строки.

Состояние кодировки влияет на следующие операции:

- Вывод информации на экран дисплея, если информация не печатается по-русски, это означает, что кодировка вводимого символа или файла не совпадает с кодировкой вывода информации.
- Ввод информации с клавиатуры, вводимые символы отображаются не в том виде, который ожидается программистом или пользователем.
- Взаимный перевод информации из одной кодировки в другую (чаще всего это файлы). Исходная и результирующая кодировка символов должны быть заданы правильно.

Примечание. Рекомендую Вам детально разобраться с кодировкой русских символов при разработке отчетов по лабораторным работам. Если Вы пойдете по “простому пути” и ограничитесь только латинскими буквами, то это будет неверно, и такой отчет по ЛР не будет считаться правильным.

Таким образом, для корректной работы в среде двух кодировок необходимо обеспечить корректный ввод и отображение информации. Это обеспечивается специальными программами – русификаторами, которые должны быть предварительно запущены перед основной работой. Русификатор представляет собой драйвер (резидентную программу), который будет работать при каждом нажатии клавиши на клавиатуре и при выводе символов на экран дисплея. Чаще всего такие драйверы совмещают две основные функции: обслуживание клавиатуры и обеспечение корректного вывода на экран дисплея. При выполнении ввода русских символов нужно переключиться в режим русской раскладки (обычно с помощью “горячей” клавиши). Никаких дополнительных действий после запуска драйвера, для корректного вывода на экран, предпринимать не нужно, он сработает автоматически. На сайте представлен драйвер RKM, который работает устойчиво и, по умолчанию, обеспечивает переключение раскладок клавиатуры с помощью клавиши “правый Shift”. Могут быть заданы и другие настройки.

Ниже мы кратко рассмотрим кодировки символов для разных режимов.

16.1. ASCII

Таблица кодировки ASCII, первая ее половина показана на рисунке расположенном ниже. Эта табличка получена в среде QC25 в ее справочной системе. Эта первая часть таблицы (0-127) является общей для всех кодировок и для разных кодовых страниц.

ASCII коды в диапазоне 0 – 127											
0	<nul>	16	<dle>	32	<sp>	48	0	64	@	80	P
1	<soh>	17	<dc1>	33	!	49	1	65	A	81	Q
2	<stx>	18	<dc2>	34	"	50	2	66	B	82	R
3	<etx>	19	<dc3>	35	#	51	3	67	C	83	S
4	<eot>	20	<dc4>	36	\$	52	4	68	D	84	T
5	<eng>	21	<nak>	37	%	53	5	69	E	85	U
6	<ack>	22	<syn>	38	&	54	6	70	F	86	V
7	<bel>	23	<eth>	39	'	55	7	71	G	87	W
8	<bs>	24	<can>	40	<	56	8	72	H	88	X
9	<tab>	25		41	>	57	9	73	I	89	Y
10	<lf>	26	<eof>	42	*	58	:	74	J	90	Z
11	<vt>	27	<esc>	43	+	59	;	75	K	91	[
12	<np>	28	<fs>	44	,	60	<	76	L	92	\
13	<cr>	29	<gs>	45	-	61	=	77	M	93]
14	<so>	30	<rs>	46	.	62	>	78	N	94	^
15	<si>	31	<us>	47	/	63	?	79	O	95	_
										111	o
										112	p
										113	q
										114	r
										115	s
										116	t
										117	u
										118	v
										119	w
										120	x
										121	y
										122	z
										123	{
										124	
										125	}
										126	~
										127	Δ

Первая группа символов в этой таблице (0-32 или 0h – 020h) является группой служебных символов, которые используются для управления. Они обычно не отображаются на экране (например, символ ESC – код 27). Несмотря на это, в программе они могут быть

использованы. Например, символ с кодом 7 (bel) может быть использован для выдачи звукового сигнала при его передачи на дисплей в текстовом режиме – при его выводе срабатывает стандартный динамик компьютера (если ООП не отключен!). Символ с кодом 13 (cr) является символом возврата каретки или конца строки, а символ с кодом 26 (eof) стандартным символом конца файла. В литературе и справочниках Вы найдете детальную расшифровку служебных символов и способов их использования.

Вторая часть таблицы ASCII показана ниже. Обратите внимание на кодировку русских букв и символов псевдографики.

ASCII коды в диапазоне 128 – 255							
128 А	144 Р	160 а	176 █	192 Ь	208 Ш	224 р	240 Ё
129 Б	145 С	161 б	177 █	193 Ь	209 Ш	225 с	241 ё
130 В	146 Т	162 в	178 █	194 Ь	210 Ш	226 т	242 ё
131 Г	147 У	163 г	179 █	195 Ь	211 Ш	227 у	243 ё
132 Д	148 Ф	164 д	180 █	196 Ь	212 Ш	228 ф	244 ё
133 Е	149 Х	165 е	181 █	197 Ь	213 Ш	229 х	245 ё
134 Ж	150 Ц	166 ж	182 █	198 Ь	214 Ш	230 ц	246 ё
135 З	151 Ч	167 з	183 █	199 Ь	215 Ш	231 ч	247 ё
136 И	152 Ш	168 и	184 █	200 Ь	216 Ш	232 ш	248 ё
137 Й	153 Щ	169 й	185 █	201 Ь	217 Ш	233 щ	249 ё
138 К	154 Ъ	170 к	186 █	202 Ь	218 Ш	234 ъ	250 ё
139 Л	155 Ы	171 л	187 █	203 Ь	219 Ш	235 ы	251 ё
140 М	156 Ь	172 м	188 █	204 Ь	220 Ш	236 ь	252 ё
141 Н	157 Э	173 н	189 █	205 Ь	221 Ш	237 э	253 ё
142 О	158 Ю	174 о	190 █	206 Ь	222 Ш	238 ю	254 ё
143 П	159 Я	175 п	191 █	207 Ь	223 Ш	239 я	255 ё

Символы псевдографики могут быть использованы в программах лабораторных работ для выполнения основных и дополнительных требований. Так в 4-й ЛР в качестве дополнительных требований предлагается поместить таблицу символов в рамку. Эту рамку можно сделать с помощью символов с кодами: 179, 191, 192, 217 и 218. Эти символы нужно будет вывести на экран в определенной последовательности.

16.2. Кодировка ANSI

Первая часть таблицы кодировки WINDOWS (ANSI) показана ниже (0-127). Вы можете убедиться в том, что она полностью совпадает с таблицей кодировки ASCII. Можете сами в этом убедиться. В этой таблице три колонки: символ (SYM), десятичный (DEC) и шестнадцатеричный (HEX) коды. В каждой строке приводятся значения 4-х символов.

SYM	DEC	HEX	SYM	DEC	HEX	SYM	DEC	HEX	SYM	DEC	HEX
	32	20	!	33	21	"	34	22	#	35	23
\$	36	24	%	37	25	&	38	26	'	39	27
(40	28)	41	29	*	42	2A	+	43	2B
,	44	2C	-	45	2D	.	46	2E	/	47	2F
0	48	30	1	49	31	2	50	32	3	51	33
4	52	34	5	53	35	6	54	36	7	55	37
8	56	38	9	57	39	:	58	3A	;	59	3B
<	60	3C	=	61	3D	>	62	3E	?	63	3F
@	64	40	A	65	41	B	66	42	C	67	43
D	68	44	E	69	45	F	70	46	G	71	47
H	72	48	I	73	49	J	74	4A	K	75	4B
L	76	4C	M	77	4D	N	78	4E	O	79	4F
P	80	50	Q	81	51	R	82	52	S	83	53
T	84	54	U	85	55	V	86	56	W	87	57
X	88	58	Y	89	59	Z	90	5A	[91	5B

\	92	5C	J	93	5D	^	94	5E	95	5F
`	96	60	a	97	61	b	98	62	c	63
d	100	64	e	101	65	f	102	66	g	103
h	104	68	i	105	69	j	106	6A	k	107
l	108	6C	m	109	6D	n	110	6E	o	111
p	112	70	q	113	71	r	114	72	s	115
t	116	74	u	117	75	v	118	76	w	119
x	120	78	y	121	79	z	122	7A	{	123
	124	7C	}	125	7D	~	126	7E	•	127
										7F

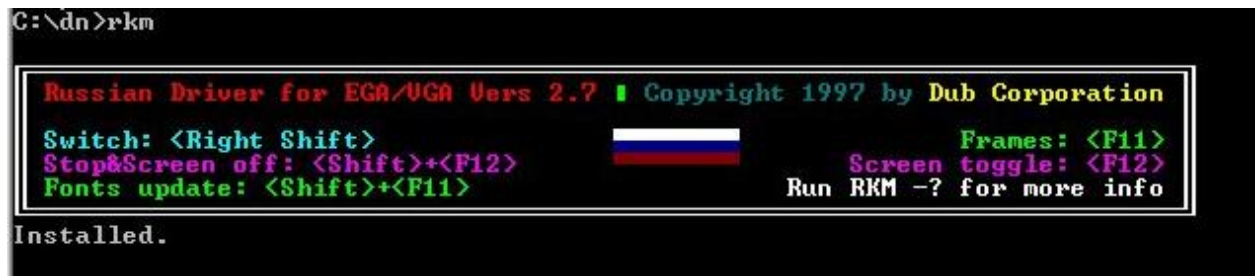
Вторая часть таблицы кодировки WINDOWS (ANSI) показана ниже (128-255). Структура таблицы аналогична. Обратите внимание на кодировку русских букв, она существенно отличается от кодировки ASCII.

Ъ	128	80	Ѓ	129	81	/	130	82	ѓ	131	83
„	132	84	...	133	85	†	134	86	‡	135	87
€	136	88	%	137	89	љ	138	8A	<	139	8B
Њ	140	8C	ќ	141	8D	џ	142	8E	џ	143	8F
ђ	144	90	`	145	91	'	146	92	“	147	93
”	148	94	•	149	95	—	150	96	—	151	97
□	152	98	™	153	99	љ	154	9A	>	155	9B
њ	156	9C	ќ	157	9D	ћ	158	9E	џ	159	9F
	160	A0	Ў	161	A1	ў	162	A2	Ј	163	A3
џ	164	A4	Ѓ	165	A5	ђ	166	A6	Ѕ	167	A7
Ѕ	168	A8	©	169	A9	€	170	AA	«	171	AB
¬	172	AC	–	173	AD	®	174	AE	ї	175	AF
°	176	B0	±	177	B1	І	178	B2	і	179	B3
ѓ	180	B4	µ	181	B5	Љ	182	B6	·	183	B7
ё	184	B8	№	185	B9	е	186	BA	»	187	BB
ј	188	BC	Ѕ	189	BD	ѕ	190	BE	ї	191	BF
А	192	C0	Б	193	C1	В	194	C2	Г	195	C3
Д	196	C4	Е	197	C5	Ж	198	C6	З	199	C7
И	200	C8	Й	201	C9	К	202	CA	Л	203	CB
М	204	CC	Н	205	CD	О	206	CE	П	207	CF
Р	208	D0	С	209	D1	Т	210	D2	У	211	D3
Ф	212	D4	Х	213	D5	Ц	214	D6	Ч	215	D7
Ш	216	D8	Щ	217	D9	Ъ	218	DA	Ы	219	DB
Ь	220	DC	Э	221	DD	Ю	222	DE	Я	223	DF
а	224	E0	б	225	E1	в	226	E2	г	227	E3
д	228	E4	е	229	E5	ж	230	E6	з	231	E7
и	232	E8	й	233	E9	к	234	EA	л	235	EB
м	236	EC	н	237	ED	о	238	EE	п	239	EF
р	240	EO	с	241	F1	т	242	F2	у	243	F3
ф	244	F4	х	245	F5	ц	246	F6	ч	247	F7
ш	248	F8	щ	249	F9	ъ	250	FA	ы	251	FB
ь	252	FC	э	253	FD	ю	254	FE	я	255	FF

16.3. Русификаторы

После запуска командной строки необходимо запустить русификатор, в нашем случае это RKM. Запуск выполняется так:

```
>RKM.COM, /
```



ниже:

Для получения более подробной информации и настройки его можно вызвать в режиме справки:

```
>RKM.COM -? ↵
```

Снятие русификатора выполняется вместе с завершением работы в режиме командной строки. Кроме этого выгрузка может быть выполнена так:

```
>RKM.COM -U ↵
```

Для русификации Вы можете использовать и другие русификаторы, однако перед их использованием желательно детально разобраться в возможностях, в параметрах программы и ограничениях применения.

16.4. Перекодировка символов

Лабораторные работы по курсу выполняется в режиме командной строки, в файловом менеджере и в операционной системе MS DOS. В этом случае используется кодировка ASCII. Текстовые константы разрабатываемых программ и комментарии в них должны вводиться в этой кодировке. При оформлении отчетов использовать среду WINDOWS и текстовые редакторы в ней (MS WORD). Эти текстовые редакторы работают в кодировке ANSI. Поэтому в процессе работ необходимо выполнять перекодировки типа:

- ANSI => ASCII и
- ASCII => ANSI.

Такие операции могут быть выполнены следующим образом:

- С использованием специального текстового редактора – ASN_ED.EXE, в котором предусмотрены эти операции преобразования (он есть на сайте). Для выполнения перекодировки нужно: загрузить файл и с помощью меню выполнить операции: “Edit” => “Convert ANSI =>ASCII ” или “Edit ” => “Convert ASCII =>ANSI ”.

- Воспользоваться специальной простой программой перекодировки – TRANS.EXE (есть на сайте).

- Разработать самостоятельно для себя программу перекодировки и использовать ее для работы. Этот вариант предлагаю сделать самостоятельно для практики программирования. Кодировка символов для этого была уже рассмотрена.

При работе с командной TRANS.EXE нужно руководствоваться следующим синтаксисом командной строки:

<Запуск TRANS >:= TRANS.EXE_</?> / TRANS.EXE_<Режим>_<Исходный файл>_<Результирующий файл>

Где:

TRANS.EXE – имя программы перекодировки.

“_” - символ пробела,

</?> - параметр выдачи справки,

<Режим>:= DW / WD – способ перекодировки: DW - ASCII =>ANSI, а WD - ANSI =>ASCII.

<Исходный файл> и <Результирующий файл> - стандартные текстовые файлы операционной системы, имена файлов могут совпадать.

Пример запуска перекодировки файла test.txt из ДОС в WINDOWS (ASCII =>ANSI):

```
>TRANS.EXE DW test.txt test.win ↵
```

В результате получим перекодированный файл test.win. для обратной перекодировки нужно вызвать программу так:

```
>TRANS.EXE WD firstw.asm firstd.asm ↵
```

Для вызова справки о работе программы нужно задать:

```
>TRANS.EXE /? ↵
```

16.5. SCAN – коды

Кодировка вводимых символов, в первую очередь отображаемых на экране, представлена во множествах ASCII и ANSI. Однако этого недостаточно для обработки в программе сигналов от клавиатуры. Например, необходимо знать была ли нажата клавиша CTRL совместно с другой клавишей, или проверить была ли нажата клавиша NUMLOCK или любая клавиша на дополнительной части клавиатуры. Фактически нужно знать номер (!!!) нажатой клавиши. Термин скан-код (Scan Code) соответствует номеру нажатой клавиши, причем этот номер может соответствовать нескольким разным символам. Так, например, скан-код 08 (см. колонку Code в таблице, приведенной ниже) соответствует символам “*” и “8” (если учесть русификацию то символов может быть больше). С помощью клавиатуры в 101 символ приходится кодировать 256 символов и служебных управляющих сигналов, определяющих управление вводом с клавиатуры. Скан-коды могут быть прочитаны в программу и использованы для работы.

Таблица основных скан-кодов клавиатуры представлена ниже:

Key	Code	Key	Code	Key	Code	Key	Code
ESC	01	U	16	or \	2B	F6	40
! or 1	02	I	17	Z	2C	F7	41
@ or 2	03	O	18	X	2D	F8	42
# or 3	04	P	19	C	2E	F9	43
\$ or 4	05	{ or [1A	V	2F	F10	44
% or 5	06	} or]	1B	B	30	NUMLOCK	45
^ or 6	07	ENTER	1C	N	31	SCROLL LOCK	46
& or 7	08	CTRL	1D	M	32	HOME or 7	47
* or 8	09	A	1E	< or ,	33	UP or 8	48
(or 9	0A	S	1F	> or .	34	PGUP or 9	49
) or 0	0B	D	20	? or /	35	-	4A
_ or -	0C	F	21	RIGHT SHIFT	36	LEFT or 4	4B
+ or =	0D	G	22	PRTSC or *	37	5	4C
LEFT	0E	H	23	ALT	38	RIGHT or 6	4D
TAB	0F	J	24	SPACEBAR	39	+	4E
Q	10	K	25	CAPSLOCK	3A	END or 1	4F
W	11	L	26	F1	3B	DOWN or 2	50
E	12	: or ;	27	F2	3C	PGDN or 3	51
R	13	" or ' or `	28	F3	3D	INS or 0	52
T	14		29	F4	3E	DEL or .	53
Y	15	LEFT SHIFT	2A	F5	3F		

Хотя для кодировки вводимых символов достаточно одного байта, при вводе с клавиатуры в программу может быть прочитано 2 байта информации. Логика ввода информации с клавиатуры такова:

- При выполнении специальных операций ввода (например, функции getch() в СИ) первый байт либо содержит код вводимого с клавиатуры (например, “А” – код 65), либо специальное значение, обозначающее использование режима ввода скан-кода.
- Специальное значение может быть либо 0, либо 0xE0h. Величина 0 означает, что получен скан-код и его значение находится во втором байте, который должен быть

считан дополнительно. Скан-код позволяет однозначно определить комбинацию клавиш нажатых одновременно (Например, для ALT+V – скан-код = 47). Значение 0xE0h также сигнализирует о режиме ввода скан-кодов, и дополнительно указывает, что используется клавиши 101 символьной клавиатуры дублирующие основные клавиши.

Список распространенных расширенных скан-кодов приведен в фрагменте программы на СИ, описывающей перечисление (enum EXTENDED) с константами скан – кодов.

```
enum EXTENDED
{
    extINTRO = 0,          // Для всех клавиатур
    extINTRO2 = 0xE0,      // For keypad and other keys unique to
                          // 101-key keyboard

    /* Второй байт содержит расширенные скан коды : */

    /* Цифровая клавиатура */
    extUP = 72,  extDOWN = 80,  extLEFT = 75,  extRIGHT = 77,
    extPGUP = 73,  extPGDN = 81,  extHOME = 71,  extEND = 79,
    extINS = 82,  extDEL = 83,

    extCTRL_PRTSC = 114,
    extCTRL_LEFT = 115,  extCTRL_RIGHT = 116,
    extCTRL_PGUP = 132,  extCTRL_PGDN = 118,
    extCTRL_HOME = 119,  extCTRL_END = 117,

    NullKey = 3,  extSH_Tab = 15,

    /* ALT+буква */
    extALT_A = 30,  extALT_B = 48,  extALT_C = 46,  extALT_D = 32,
    extALT_E = 18,  extALT_F = 33,  extALT_G = 34,  extALT_H = 35,
    extALT_I = 23,  extALT_J = 36,  extALT_K = 37,  extALT_L = 38,
    extALT_M = 50,  extALT_N = 49,  extALT_O = 24,  extALT_P = 25,
    extALT_Q = 16,  extALT_R = 19,  extALT_S = 31,  extALT_T = 20,
    extALT_U = 22,  extALT_V = 47,  extALT_W = 17,  extALT_X = 45,
    extALT_Y = 21,  extALT_Z = 44,

    /* extALT+цифровая клавиатура */
    extALT_1 = 120,  extALT_2,  extALT_3,  extALT_4,  extALT_5,
    extALT_6,          extALT_7,  extALT_8,  extALT_9,  extALT_0,

    extALT_minus = 130,  extALT_equals,

    /* функциональная клавиша */
    extF1 = 59,  extF2,  extF3,  extF4,  extF5,
    extF6,          extF7,  extF8,  extF9,  extF10,
    extF11 = 133,  extF12,

    /* SHIFT+функциональная клавиша */
    extSH_F1 = 84,  extSH_F2,  extSH_F3,  extSH_F4,  extSH_F5,
    extSH_F6,          extSH_F7,  extSH_F8,  extSH_F9,  extSH_F10,
    extSH_F11 = 137,  extSH_F12,

    /* CTRL+ функциональная клавиша */
    extCTRL_F1 = 94,  extCTRL_F2,  extCTRL_F3,  extCTRL_F4,
    extCTRL_F5,          extCTRL_F6,  extCTRL_F7,  extCTRL_F8,
    extCTRL_F9,          extCTRL_F10,  extCTRL_F11 = 137,
    extCTRL_F12,
```

```

/* ALT+ функциональная клавиша */
extALT_F1 = 104,  extALT_F2, extALT_F3, extALT_F4, extALT_F5,
extALT_F6,        extALT_F7, extALT_F8, extALT_F9, extALT_F10,
extALT_F11 = 139, extALT_F12,
};

```

В справочниках и в литературе вы найдете полный перечень расширенных скан – кодов. Кроме того, вы можете воспользоваться программкой, которая приведена в конце данного раздела для получения таблиц кодов самостоятельно.

16.6. Кодировка UNICODE

Большинство современных программ позволяет работать в специальной кодировке UNICODE. В этой кодировке символы кодируются 2-мя байтами, поэтому возможно закодировать 65536 символов. Такой способ кодирования позволяет таблицы кодов для разных стран и для разных иностранных языков. Это в свою очередь позволяет хранить информацию в файлах и БД на разных языках, что делает программные продукты более универсальными.

Здесь мы не будем приводить особенности работы с кодировкой UNICODE, так как для выполнения ЛР и курсовой работы в этом нет необходимости. Таблицы UNICODE очень большие, поэтому для получения информации о кодах символов удобнее воспользоваться специальной программой *charmap.exe*, которую Вы легко найдете в операционной системе. Эту программу достаточно вызвать из главного меню WINDOWS: “Пуск/Start” => “Выполнить/Run” => “*charmap.exe*”.

16.7. Программы для получения списка кодов

Небольшая программка на СИ, сделанная в среде QC25, позволяет получить коды разных символов в кодировке ASCII. Текст ее приведен ниже. Ее можно скопировать в NOTEPAD, сохранить с расширением *.c, перекодировать в ДООС формат и выполнить в QC25.

```

#include <stdio.h>
main()
{
    int key;  // Переменная для ввода кодов символов

    /* Программа читает и выводит символы пока не нажата
    клавиша ESC */
    while( 1 )
    {
        // Ввод первого символа
        key = getch();
        if( (key == 0) || (key == 0xE0) )
        {
            // Если скан код(первый символ = 0) или расширенная клавиатура (101 клавиш)
            // В этом случае первый символ равен 0xE0, то выводим так:
            key = getch();
            printf( "ASCII: скан\tChar: Нет\t" );
            printf( "Десятичное: %d\tШестнадц.: %X\n", key, key );
        }
        else
        {
            // Первый символ не является скан кодом, а обычным символом, то выводим так:
            if ( key <=128 )
            // Можно ли напечатать символ
            printf( "ASCII: Да\tСимвол: %c \t", isgraph( key ) ? key : ' ' );
        }
    }
    printf( "ASCII: Да\tСимвол: %c \t", key );
}

```

```

printf( "Десятичное: %d\tШестнад.: %X\n", key, key );

    }
    /* Проверка ESC - 27 или 0x1b */
    if( key == 27)
    {
        printf( "Вы хотите завершить программу? (Y/n) " );
        key = getche();
        printf( "\n" );
    }
    // Завершаем программу по Y/y или Enter
    if( (toupper( key ) == 'Y') || (key == 13) )
        break;
}
}
}

```

Блок-схема данной программы представлена в разделе 15.3. Примеры блок-схем программ. Достаточно щелкнуть мышкой на номере раздела для перехода в раздел описания блок-схем.

Программка для записи в файл кодов ANSI. Данную программу можно выполнить в среде BORLANDC и получить файл справки с ANSI кодировкой (ANSI.txt).

```

#include <stdio.h>
void main()
{
    int key , i ; // Переменная для ввода кодов символов
    FILE *fp;
    // Открываем файл
    fp = fopen("ANSI.txt", "w");
    fprintf( fp, "SYM    DEC    HEX SYM    DEC    HEX SYM    DEC    HEX
SYM DEC    HEX\n" );
    for( i = 32 ; i < 256; i++ )
    {
        fprintf( fp, " %c    %3d    %X  ", i , i , i );
        i++;
        fprintf( fp, " %c    %3d    %X  ", i , i , i );
        i++;
        fprintf( fp, " %c    %3d    %X  ", i , i , i );
        i++;
        fprintf( fp, " %c    %3d    %X  \n", i , i , i );
    }

    fclose (fp);
}

```

Практика.

1. Запустите под управлением VS 2005 программу для считывания расширенных кодов.
2. Распечатайте список кодов и сохраните в отдельный текстовый файл.

17. Литература

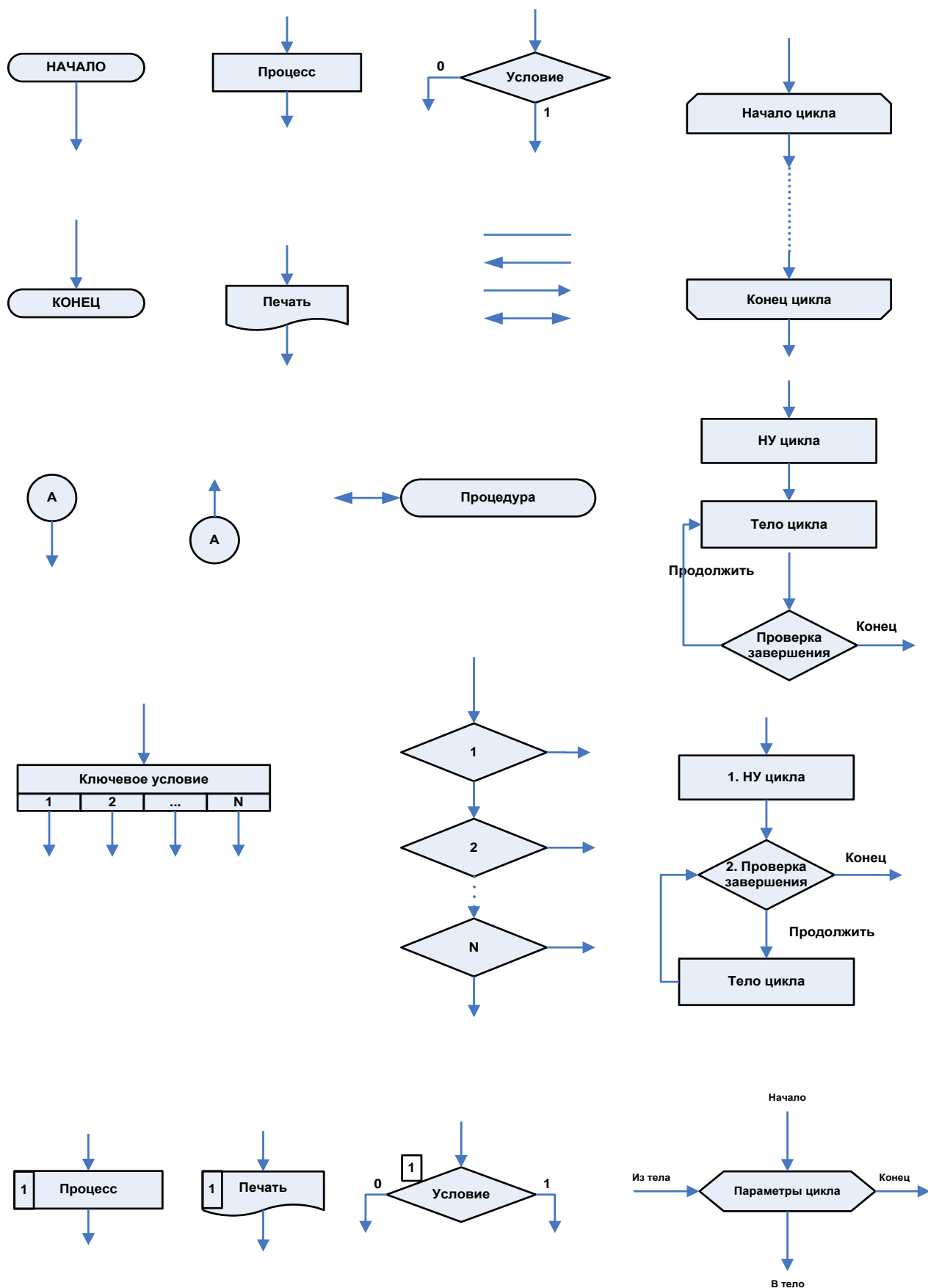
Основная литература

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. К36 Язык программирования Си.\Пер. с англ., 3-е изд., испр. - СПб.: "Невский Диалект", 2001. - 352 с.: ил.
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 (Microsoft Document Explorer – входит в состав дистрибутива VS. Нужно обязательно развернуть при установке VS VS или настроить доступ через Интернет.)

Дополнительная литература

5. 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!!!
6. Другие методические материалы по дисциплине с сайта www.sergebolshakov.ru.
7. Конспекты лекций по дисциплине “Основы программирования”.
8. Подбельский В.В. Язык Си++: Учебное пособие. – М.: Финансы и статистика, 2003.
9. 5. Подбельский В.В. Стандартный СИ++: Учебное пособие. – М.: Финансы и статистика, 2008.
10. Г. Шилдт “С++ Базовый курс”: Пер. с англ.- М., Издательский дом “Вильямс”, 2011 г. – 672с
11. Фридланд А.Я. Информатика и компьютерные технологии. Основные термины: толковый слов. : 3-е изд. Испр. и доп./ А.Я. Фридланд, Л.С. Хааамирова, И.А. Фридланд. – М.:ООО «Издательство Астрель»: ООО «Издательство АСТ». 2003 – 272с.
12. Г. Шилдт “С++ Руководство для начинающих” : Пер. с англ. - М., Издательский дом “Вильямс”, 2005 г. – 672с
13. Г. Шилдт “Полный справочник по С++”: Пер. с англ.- М., Издательский дом “Вильямс”, 2006 г. – 800с
14. Бьерн Страуструп "Язык программирования С++"- М., Бином, 2010 г.

Приложение 1 Блок-схемы



Приложение 2 . Примеры программной реализации КЛР/ДЗ

17.1. Главный модуль File_P3.cpp**File_P3.cpp – ГЛАВНЫЙ МОДУЛЬ**

```

// Комплексная лабораторная работа
#include "stdafx.h"
#include "FILE_P3.h"
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
// #include <afx.h>
#include <stdio.h>
// #include <iostream>
#include <conio.h>
#include <string>
#include <string.h>
#include <memory.h>
//
#include <errno.h>

//using namespace std;
///
////////////////////
// Задание для работы с файлами (уровень А)
////////////////////
// Прототипы функций работы с файлами
// МУ п.5
void PrintStudent(Student * pS);
// МУ п.8
void StudPrintMas( Student * pMas , int Razm);
// МУ п.12
void StudFileToMas( const char * FileName , Student * pMas , int * pRazm);
// МУ п.13
void StudMasToFile( const char * FileName , Student * pMas , int Razm);
// МУ п.14
void StudPrintFile( const char * FileName );
// МУ п.15
int StudFileCount( const char * FileName , int * pRazm);
// МУ п.16
void StudClearFile(const char * FileName);
// МУ п.17
void ClearReadOnlyFile(const char * FileName);
// МУ п.18
void SwapStudent ( Student * pA , Student * pB );
// МУ п.20
void StudSortMasNum( Student * pMas , int Razm );
// МУ п.21
void StudSortOklad(const char * FileName );
// МУ п.22
double StudSumm( const char * FileName , double * Sum);
////////////////////
// Задание для работы с файлами (уровень В и С)
////////////////////
// МУ п.21
void StudSortName(const char * FileName );
// МУ п.21
void StudSortNum(const char * FileName );
// МУ п.23.1/2
void StudAdd(const char * FileName , Student S , TypeAddDel TAD );

```

```

// МУ п.23.4/5
void StudDel(const char * FileName , TypeAddDel TAD );
// МУ п.23.3
void StudAddNum(const char * FileName , Student S , int Numb );
// МУ п.23.6
void StudDelNum(const char * FileName , int Numb );
// МУ п.23.7
void FindStudNum( const char * FileName , Student * pS , int Numb);
// МУ п.23.8
void ChangeStudNum( const char * FileName , Student pS , int Numb);
// МУ п.23.9
int FindStudName( const char * FileName , Student * pS , const char * FindName );
// МУ п.23.10
void SwapStudFile( const char * FileName , int NumA ,int NumB);
// МУ п.17/23ю11
void StudRemoveFile(const char * FileName);

////////////////////
void main(void)
{
system(" chcp 1251 > nul");
//////////////////// СИ начало
    int pFBin = 0;
    //////////////////////
    //////////////////////
    //StudRemoveFile("BDStud.bin"); // Тестирование БД
    //////////
    // МУ п.3
    //////////
    Student Stud1 = { "Лаптева" , 1 , 1000.0};
    //////////
    // МУ п.4
    //////////
    Student Stud2;
    Stud2.Num = 2;
    Stud2.Oklad = 2000.00;
    strcpy(Stud2.Name , "Аксенова");
    printf ("Пункт МУ 4:\n");

    printf( "Stud2: Имя = %-15s Номер = %2d Стипендия = %8.2lf \n\n",
        Stud2.Name , Stud2.Num, Stud2.Oklad );

    //////////
    // МУ п.5
    //////////
    printf ("Пункт МУ 5:\n");
    PrintStudent(&Stud1) ;
    PrintStudent(&Stud2) ;

    //////////
    // МУ п.6
    //////////
    // Student * pStud = new Student;
    Student * pStud = (Student *) malloc (sizeof(Student));
    pStud->Num = 3;
    pStud->Oklad = 3000.00;
    strcpy(pStud->Name , "Большаков");
    printf ("Пункт МУ 6:\n");
    PrintStudent( pStud ) ;
    // delete pStud;
    free( pStud );

```

```

pStud= NULL;
//////////
// МУ п.7
//////////
Student SMas[] ={{"Первый" , 1 , 1000.0}, {"Второй" , 2 , 2000.0}, {"Третий" , 3 , 3000.0},
{"Четвертый" , 4 , 4000.0}};
int RazmS = sizeof(SMas)/ sizeof(Student); // Вычисление размерности массива
printf ("Пункт МУ 7:\n");
for (int i = 0 ; i < RazmS ; i++ )
    PrintStudent( &SMas[i] ) ;

//////////
// МУ п.8
//////////
printf ("Пункт МУ 8:\n");
StudPrintMas( SMas , RazmS);

//////////
// МУ п.9
//////////
printf ("Пункт МУ 9:\n");
const int Rzm = 6;
Student * pPotok = new Student[Rzm];
// Заполнение
srand( (unsigned)time( NULL ) ); // Настройка датчика случайных чисел
rand(); // первое число
for (int i=0 ; i < Rzm ; i++ )
{
    char Buf[20];
    char Num[10];

    strcpy(Buf , "Stud № - ");
    pPotok[i].Num = (rand()*9)/ RAND_MAX ; // диапазон 0 - 9
    pPotok[i].Oklad = 1000.0 * 10.0 * rand() / RAND_MAX ;// диапазон 0 - 10000.0
    int n = (rand()*30)/ RAND_MAX ;// диапазон 0 - 30
    strcat(Buf , itoa (n + 1 ,Num, 10 ));
    strcpy( pPotok[i].Name , Buf);
    // PrintStudent( &pPotok[i] );
};
StudPrintMas( pPotok , 6);

//////////
// МУ п.10 BDStud.bin
//////////
//////////
// МУ п.11 Синий
//////////
Student ZMas[] ={{"Запись 1" , 0 , 1000.0}, {"Запись 2" , 0 , 2000.0}, {"Запись 3" , 0 ,
3000.0},
{"Запись 4" , 0 , 4000.0}};
int RazmZ = sizeof(ZMas)/ sizeof(Student); // Вычисление размерности массива

printf ("Пункт МУ 11(синий):\n");
system ( "attrib -R BDStud.bin " ); // Снятие атрибута защиты записи, нужно после первого
создания
pFBin = _open( "BDStud.bin" , _O_RDWR | _O_BINARY | _O_CREAT | _O_TRUNC);
system("attrib -R BDStud.bin ");
if ( pFBin != -1 )
{
    for ( int i = 0 ; i < RazmZ ; i++ )
    {
        ZMas[i].Num = i + 1;
        _write (pFBin , &ZMas[i] , sizeof(Student));
    }
}

```

```

    };
    _close( pFBin );
};
StudPrintMas( ZMas , RazmZ);

    system("attrib -R BDStud.bin ");
/////////
// МУ п.11 Рас
Student MASStud[100];
Student SRBuf;
printf ("Пункт МУ 11(Печать файла):\n");

pFBin = _open( "BDStud.bin", _O_RDWR | _O_BINARY );
if ( pFBin != -1 )
{
    for ( int i = 0 ; _eof(pFBin) == NULL ; i++ )
    {
        /* int nByte = _read( pFBin , &SRBuf , sizeof(Student));
        PrintStudent( &SRBuf ); */
        int nByte = _read( pFBin , &MASStud[i] , sizeof(Student));
        PrintStudent( &MASStud[i] );
        RazmZ = i;
    }
    _close( pFBin );
};
printf ("Пункт МУ 11(Печать всего массива):\n");
StudPrintMas( MASStud , RazmZ + 1 );

/////////
// МУ п.11 Зеленый
/////////
printf ("Пункт МУ 11:\n");
system ( "attrib -R BDStud.bin "); // Снятие атрибута защиты записи, нужно после первого
создания

pFBin = _open( "BDStud.bin", _O_RDWR | _O_BINARY | _O_CREAT | _O_TRUNC);
system("attrib -R BDStud.bin ");

if ( pFBin != -1 )
{
    for ( int i = 0 ; i < 4 ; i++ )
    {
        Student StudFile;
        char Buf[20];
        char Num[10];
        strcpy(Buf , "Запись № - ");
        StudFile.Num = (rand()*9)/ RAND_MAX ;
        StudFile.Oklad = 1000.0 * 10.0 * rand() / RAND_MAX ;
        strcat(Buf , itoa (i + 1 ,Num, 10 ));
        strcpy( StudFile.Name , Buf);
    }
    //
    _write (pFBin , &StudFile , sizeof(Student));
    PrintStudent( &StudFile );
};
//
_close( pFBin );
system("attrib -R BDStud.bin ");

};

```

```

// Специальные функции:
// - из файла в массив ( имя файла , указатель на массив структурных переменных)
///////////////////////////////////////////////////////////////////
//
// МУ п.12
//
Student * pStMas;
int StudCount;
//
printf ("Пункт МУ 12:\n");
//printf ("Работа с функциями:\n");
StudFileToMas( "BDStud.bin" , &pStMas , &StudCount );
StudPrintMas( pStMas , StudCount);
// - из массива в файл ( имя файла , указатель на массив структурных переменных)
strcpy( (char *)((pStMas + 1)->Name) , "Новая");
///////////////////////////////////////////////////////////////////
//
// МУ п.13
//
Student NZMas[] ={{"Запись 1" , 1 , 1000.0}, {"Запись 2" , 2 , 5000.0}, {"Запись 3" , 3 ,
3000.0},
{"Запись 4" , 4 , 4000.0}};
RazmZ = sizeof(NZMas)/ sizeof(Student); // Вычисление размерности массива
// Изменение второй (индекс = 1) записи
NZMas[1].Num =5;
strcpy( NZMas[1].Name , "Изменение 2-й");
printf ("Пункт МУ 13:\n");
StudMasToFile( "BDStud.bin" , NZMas , RazmZ );
printf ("После редактирования массива и запоминания в файле\n");
StudPrintMas( NZMas , StudCount);
//
// МУ п.14
//
printf ("Пункт МУ 14:\n");
printf ("После редактирования массива и запоминания - распечатка из файла
функцией:\n");
// Распечатка файла записей
StudPrintFile( "BDStud.bin" );
// Число записей в БД
//
// МУ п.15
//
printf ("Пункт МУ 15:\n");
//
printf ("Число записей в БД = %d \n",StudFileCount( "BDStud.bin" , &StudCount ));
// Заполнение файла на основе массива
//
// МУ п.13 (второй вариант на основе инициализированного массива)
//
printf ("Пункт МУ 13:\n");
Student StudMas[] ={{"Иванов" , 3 , 7000.0}, {"Петров" , 2 , 2000.0}, {"Сидоров" , 1 ,
3000.0}, {"Печкин" , 4 , 4000.0}};
StudMasToFile( "BDStud.bin" , StudMas , StudCount );
// Распечатка файла записей
StudPrintFile( "BDStud.bin" );
//
//
// МУ п.16
//
printf ("Пункт МУ 16:\n");

```

```
StudClearFile("BDStud.bin");
```

```
StudPrintFile( "BDStud.bin" );
```

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

```
// МУ п.17
```

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

```
//
```

```
printf ("Пункт МУ 17:\n");
```

```
// StudRemoveFile("BDStud.bin"); // Временно за комментировано
```

```
ClearReadOnlyFile("BDStud.bin");
```

```
StudClearFile("BDStud.bin");
```

```
StudPrintFile( "BDStud.bin" );
```

```
//
```

```
printf ("Пункт МУ 17 снова добавим записи из массива:\n");
```

```
StudMasToFile( "BDStud.bin" , pStMas , StudCount );
```

```
StudPrintFile( "BDStud.bin" );
```

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

```
// МУ п.18
```

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

```
printf ("Пункт МУ 18 (Sawp - до):\n");
```

```
Student StudA = { "Лаптева" , 1 , 1000.0};
```

```
Student StudB = { "Иванова" , 2 , 2000.0};
```

```
PrintStudent(&StudA);
```

```
PrintStudent(&StudB);
```

```
printf ("Пункт МУ 18 (Sawp - после):\n");
```

```
SwapStudent( &StudA , &StudB); // Обмен двух отдельных записей
```

```
PrintStudent(&StudA);
```

```
PrintStudent(&StudB);
```

```
// В массиве
```

```
printf ("Пункт МУ 18 (Sawp - pStMas - до):\n");
```

```
StudPrintMas( pStMas , StudCount);
```

```
printf ("Пункт МУ 18 (Sawp - pStMas - после):\n");
```

```
SwapStudent( &pStMas[0] , &pStMas[2]); // Обмен двух элементов массива
```

```
StudPrintMas( pStMas , StudCount);
```

```
// Сортировка массива по одному полю записи
```

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

```
// МУ п.19
```

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

```
printf ("Пункт МУ 19 (SORT - pStMas - до):\n");
```

```
StudPrintMas( pStMas , StudCount);
```

```
// Сортировка по окладу
```

```
for( int k = 0 ; k < StudCount - 1 ; k++)
```

```
for ( int i = 0 ; i < StudCount - 1 ; i++)
```

```
if ( (pStMas + i)->Oklad > (pStMas + i + 1)->Oklad ) // Возрастание
```

```
SwapStudent( pStMas + i , pStMas + i + 1 );
```

```
printf ("Пункт МУ 19 (SORT - pStMas - после Oklad):\n");
```

```
StudPrintMas( pStMas , StudCount);
```

```
// Сортировка по имени
```

```
for( int k = 0 ; k < StudCount - 1 ; k++)
```

```
for ( int i = 0 ; i < StudCount - 1 ; i++)
```

```
if ( strcmp((pStMas + i)->Name , (pStMas + i + 1)->Name ) > 0 ) //
```

Возрастание

```
SwapStudent( pStMas + i , pStMas + i + 1 );
```

```
printf ("Пункт МУ 19 (SORT - pStMas - после Name):\n");
```

```
StudPrintMas( pStMas , StudCount);
```

```
// Сортировка по номеру
```

```
for( int k = 0 ; k < StudCount - 1 ; k++)
```

```
for ( int i = 0 ; i < StudCount - 1 ; i++)
```

```
if ( (pStMas + i)->Num > (pStMas + i + 1)->Num ) // Возрастание
```

```
SwapStudent( pStMas + i , pStMas + i + 1 );
```

```

printf ("Пункт МУ 19 (SORT - pStMas - после Num):\n");
StudPrintMas( pStMas , StudCount);
////////
// МУ п.20
////////
printf ("Пункт МУ 20 (SORT - функция - до):\n");
StudPrintMas( pStMas , StudCount);
StudSortMasNum( pStMas , StudCount );
printf ("Пункт МУ 20 (SORT - функция - после):\n");
StudPrintMas( pStMas , StudCount);
//
////////
// МУ п.21
////////
// Сортировка файла (!!!) по одному полю записи
////////////////////////////////////
printf ("Пункт МУ 21 (SORT - с разными функциями):\n");

printf ("СОТИРОВКА: Работа с функциями Name:\n");
StudSortName("BDStud.bin" );
StudPrintFile( "BDStud.bin" );
////////////////////////////////////
printf ("СОТИРОВКА: Работа с функциями Num (убывание):\n");
StudSortNum("BDStud.bin");
StudPrintFile( "BDStud.bin" );
////////////////////////////////////
printf ("СОТИРОВКА: Работа с функциями Oklad(возрастание ):\n");
StudSortOklad("BDStud.bin" );
StudPrintFile( "BDStud.bin" );
////////////////////////////////////
////////
// МУ п.22 чтение одной записи 3-й (0 - 4 )
////////
Student SFind ;
pFBin = _open( "BDStud.bin" , _S_IREAD |_O_BINARY );
if ( pFBin != -1 )
{
    long posF = _lseek( pFBin, sizeof(Student) * 2, SEEK_SET ); // Установить новую текущую
    позицию
    int nByte = _read( pFBin , &SFind , sizeof(Student));
};
printf ("Пункт МУ 22:\n");
printf ("Чтение из программы (номер = 3):\n");
PrintStudent( &SFind );
// Закрытие файла
    _close( pFBin );

//
////////
// МУ п.22.1 чтение из функции
////////
///// - Прочитать запись по номеру
printf ("Чтение из функции (номер = 1):\n");
// Student SFind ;
FindStudNum( "BDStud.bin" , &SFind , 1);
PrintStudent( &SFind );
////////
// МУ п.22.2 интегрированные расчеты в программе
////////
double SumStip = 0;
Student SIntg;

```

```

printf ("Пункт МУ 22.2(Интегральные расчеты):\n");
pFBin = _open( "BDStud.bin" , _S_IREAD | _O_BINARY );
if ( pFBin != -1 )
{
    for ( int i = 0 ; _eof(pFBin) == NULL ; i++ )
    {
        int nByte = _read( pFBin , &SIntg , sizeof(Student));
        SumStip = SumStip + SIntg.Oklad;
    }
    _close( pFBin );
};
printf ( "Фонд зарплаты в цикле = %8.2lf \n" , SumStip );

////////
// МУ п.22
////////
double SumOklad;
printf ( "Фонд зарплаты из функции = %8.2lf \n" , StudSumm( "BDStud.bin" , &SumOklad));
////////
////////
// Очистка файла и его удаление
ClearReadOnlyFile("BDStud.bin");
// StudClearFile("BDStud.bin");
// StudRemoveFile("BDStud.bin");
///// УРОВЕНЬ С
////////
// МУ п.23
////////
// МУ п.23.1
////////
// Добавить запись ( начало )
printf ("Добавление (начало):\n");
Student SFirst = {"Первый" , 22 , 3400.00};
StudAdd("BDStud.bin" , SFirst , First );
StudPrintFile( "BDStud.bin" );
////////
// МУ п.23.2
////////
// - Добавить запись ( конец)
printf ("Добавление (конец):\n");
Student SLast = {"Последний" , 22 , 3400.00};
StudAdd("BDStud.bin" , SLast );
StudPrintFile( "BDStud.bin" );
////////
////////
// МУ п.23.3
////////
// Добавить запись (номер)
////////
printf ("Добавление (номер):\n");
Student SNumb = {"По номеру - 2" , 33 , 5500.00};
StudAddNum("BDStud.bin" , SNumb , 3);
StudPrintFile( "BDStud.bin" );
///// - Удалить запись по номеру (? другие режимы)
////////
////////
// МУ п.23.4
////////
printf ("Удаление (Начало):\n");
StudDel("BDStud.bin" , First );

```

```

StudPrintFile( "BDStud.bin" );
//////////
//////////
// МУ п.23.5
//////////
printf ("Удаление (конец):\n");
StudDel("BDStud.bin", Last );
//StudDel("BDStud.bin" );
StudPrintFile( "BDStud.bin" );
//////////
//////////
// МУ п.23.6
//////////
printf ("Удаление (номер):\n");
StudDelNum("BDStud.bin", 4 );
StudPrintFile( "BDStud.bin" );
//////////
//////////
// МУ п.23.7
//////////
///// - Прочитать запись по номеру
printf ("Чтение (номер):\n");
Student * pFind = new Student;
FindStudNum( "BDStud.bin" , pFind , 2);
PrintStudent( pFind );
//////////
// - Изменить запись по номеру
//////////
// МУ п.23.8
//////////
printf ("Изменение записи (номер):\n");
Student SChang ={"Замена" , 11, 111.00};
ChangeStudNum( "BDStud.bin" , SChang , 2);
StudPrintFile( "BDStud.bin" );
//////////
// - Найти запись по признаку равенства поля (имени - первую в файле) по любому полю
//////////
// МУ п.23.9
//////////
int n;
printf ("Поиск по имени:\n");
if ( ( n =FindStudName( "BDStud.bin" , pFind , "Замена" )) != -1)
{ printf ("Найдена по номеру = %d \n", n );
printf ("Найдена запись: ");
PrintStudent( pFind );
}
else
printf ("Не найдена запись! ");
//////////
//////////
// МУ п.23.10
//////////
// Swap замена записей в файле БД
printf ("Замена SWAP в файле по номеру: 1<->3\n");
SwapStudFile( "BDStud.bin" , 1 ,3 );
StudPrintFile( "BDStud.bin" );
//////////
delete [] pStMas;
//////////
system("PAUSE");

```

////////////////////////////////////

}

17.2. Заголовочный файл проекта File_P3.h**File_P3.h– Заголовочный МОДУЛЬ**

```

//
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <memory.h>
#include <malloc.h>

//

#include <errno.h>
//
//using namespace std;
////////////////////
////////
// МУ п.2
////////
struct Student {
public:
    char Name[20];
    int Num;
    double Oklad;
};
// Константы для управления функциями
enum TypeAddDel{ First = 1 , Last = 2 , Default = Last};
// Прототипы
////////
////////////////////
// МУ п.5
// 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 );

};
////////////////////
// МУ п.8
void StudPrintMas( Student * pMas , int Razm)
{
    for (int i = 0 ; i < Razm ; i++ )
    {
        PrintStudent(pMas + i); // Используется готовая функция
    }
};
////////////////////
// Выборка из файла в динамический массив
// МУ п.12
////////////////////
void StudFileToMas( const char * FileName , Student ** pMas , int * pRazm)
{
    long posF;
    int pFBin = _open( FileName, _S_IREAD | _O_BINARY );

```

```

if ( pFBin == -1 ) { *pRazm = NULL; return;};
posF = _lseek( pFBin, 0, SEEK_END ); // Установить новую текущую позицию
*pRazm = posF / sizeof(Student);
// *pMas = new Student[ *pRazm];
*pMas = (Student *) calloc( *pRazm , sizeof(Student) );
posF = _lseek( pFBin, 0, SEEK_SET ); // На начало
// Буфер для чтения
Student SBuf;
    for (int i = 0 ; i < *pRazm ; i++ )
    {
        int nByte = _read( pFBin , &SBuf , sizeof(Student));
        if ( nByte == NULL) break;
        memcpy( *pMas + i , &SBuf , sizeof(Student));
    };
// Заккрытие файла
    _close( pFBin );
};
//
// //////////////////////////////////
//Перезапись массива структур в файл (Файл очищается!)
// МУ п.13
// //////////////////////////////////
void StudMasToFile( const char * FileName , Student * pMas , int Razm)
{
    char Comand[40];
    // Снятие защиты с файла - отключение Readonly
    strcpy (Comand , "attrib -R ");
    strcat (Comand , FileName);
    system( Comand );
    int pFBin = _open( FileName , _O_RDWR | _O_BINARY | _O_CREAT | _O_TRUNC);
    if ( pFBin == -1 ) return;
    // цикл записи
    for ( int i = 0 ; i < Razm ; i++ )
    {
        _write (pFBin , pMas + i , sizeof(Student));
    };

    _close( pFBin );
//
};
// //////////////////////////////////
// Печать файла БД Студентов
// МУ п.14
// //////////////////////////////////
void StudPrintFile( const char * FileName )
{
    long posF;
    int pFBin = _open( FileName, _S_IREAD | _O_BINARY );
    if ( pFBin != -1 )
    {
        posF = _lseek( pFBin, 0, SEEK_END ); // Установить новую текущую позицию
        int Razm = posF / sizeof(Student);
        if ( Razm == 0 ) { _close( pFBin ); printf( "Записей в файле нет! \n" ); return;};
        posF = _lseek( pFBin, 0, SEEK_SET ); // На начало
        // Буфер для чтения
        Student SBuf;
            for (int i = 0 ; i < Razm ; i++ )
            {
                int nByte = _read( pFBin , &SBuf , sizeof(Student));

```

```

        if ( nByte == NULL) break;
        // вывод записи
        printf( "%d - ", i + 1);
        PrintStudent(&SBuf);
    };
    // Закрытие файла
    _close( pFBin );
};
};

////////////////////
// Получить число записей в БД
// МУ п.15
int StudFileCount( const char * FileName , int * pRazm)
{
    long posF;
    int pFBin = _open( FileName, _S_IREAD | _O_BINARY );
    if ( pFBin == -1 ) { *pRazm = NULL; return NULL;};
    posF = _lseek( pFBin, 0, SEEK_END ); // Установить новую текущую позицию
    *pRazm = posF / sizeof(Student);
    _close( pFBin );
    return *pRazm;
};
////////////////////
// Очистка файла студентов
// МУ п.16
////////////////////
void StudClearFile(const char * FileName)
{
    char Comand[40];
    // Временное снятие защиты с файла - отключение Readonly
    strcpy (Comand , "attrib -R ");
    strcat (Comand , FileName);
    system( Comand );
    int pFBin = _open( FileName , _O_RDWR | _O_BINARY | _O_CREAT | _O_TRUNC);
    if ( pFBin == -1 ) return;
    _close( pFBin );
};
////////////////////
// Сброс флага файла
// МУ п.17
////////////////////
void ClearReadonlyFile(const char * FileName)
{
    char Comand[40];
    // Снятие защиты с файла - отключение Readonly
    strcpy (Comand , "attrib -R ");
    strcat (Comand , FileName);
    system( Comand );
};
////////////////////
// МУ п.18
////////////////////
// обмен на основе адресов структурных переменных (возможен только без динамики!!)
void SwapStudent ( Student * pA , Student * pB )
{
    Student Temp;
    /* можно и так
    Temp = *pA;
    *pA = *pB;

```

```

*пВ = Temp;
*/

memcpy( &Temp , pA, sizeof(Student) );
memcpy( pA , pB, sizeof(Student) );
memcpy( pB , &Temp, sizeof(Student) );
};
//////////
// МУ п.20
//////////
void StudSortMasNum( Student * pStMas , int StudCount )
{
    for( int k=0 ; k< StudCount - 1 ; k++)
        for ( int i=0 ; i< StudCount - 1 ; i++)
            if ( (pStMas + i)->Num > (pStMas + i + 1)->Num ) // Возрастание
                SwapStudent( pStMas + i , pStMas + i + 1 );
};
//////////
// Сортировка массива по окладу по убыванию
// МУ п.21
//////////
void StudSortOklad(const char * FileName )
{
    // Чтение из файла в массив
    Student * pStudMas;
    int Razm;
    StudFileToMas( FileName , &pStudMas , &Razm );
    // Сортировка массива по окладу по убыванию
    for ( int i = 0 ; i < Razm - 1 ; i++)
        for ( int k = 0 ; k < Razm - 1 ; k++ )
            // Сортировка разные режимы!!!!!!!!!!!!!!!!!!!!!!
            if ( ((Student *) (pStudMas + k))->Oklad > ((Student *) (pStudMas + k + 1))->Oklad ) //
                SwapStudent ( pStudMas + k , pStudMas + k+1 );
};
//
StudMasToFile( FileName , pStudMas , Razm );
};
//////////
// МУ п.22
//////////
double StudSumm( const char * FileName , double * Sum)
{
    Student * pStudMas;
    int Razm;
    *Sum = 0.0;
    StudFileToMas( FileName , &pStudMas , &Razm );
    for ( int i = 0 ; i < Razm ; i++)
        *Sum+= (pStudMas + i)->Oklad;
    delete [] pStudMas;
    return *Sum;
};
//////////
// Сортировка по имени (дополнительные примеры) МУ п.21
//////////
void StudSortName(const char * FileName )
{
    // Чтение из файла в массив
    Student * pStudMas;
    int Razm;
    StudFileToMas( FileName , &pStudMas , &Razm );
}

```

```

//
//////////
for (int i = 0 ; i < Razm - 1 ; i++)
    for (int k = 0 ; k < Razm - 1 ; k++)
// Сортировка по имени
    if ( strcmp(((Student *) (pStudMas + k)) -> Name , ((Student *) (pStudMas + k + 1)) -
> Name) > 0 ) // Убывание имя
        { SwapStudent ( pStudMas + k , pStudMas + k + 1 ); };
//
StudMasToFile( FileName , pStudMas , Razm );
};
//////////
// Сортировка по номеру (дополнительные примеры) МУ п.21
//////////
void StudSortNum(const char * FileName )
{
// Чтение из файла в массив
Student * pStudMas;
int Razm;
StudFileToMas( FileName , &pStudMas , &Razm );
StudSortMasNum( pStudMas , Razm ); // Использование сортировки в массиве!!!
StudMasToFile( FileName , pStudMas , Razm );
};
//////////
//////////
//////////
// Удаление из файла начало или конец файла
//////////
// МУ п.23.1/2
//////////
void StudAdd(const char * FileName , Student S , TypeAddDel TAD = Default ){
//
if ( TAD == Last)
{
ClearReadOnlyFile(FileName);
int pFBin = _open( FileName, _O_RDWR | _O_BINARY | _O_APPEND );
long posF = _lseek( pFBin, 0, SEEK_END ); // Установить новую текущую позицию
        _write (pFBin , &S, sizeof(Student));
// Закрытие файла
        _close( pFBin );
        return;};
if( TAD == First )
{
Student * pStudMas;
int Razm;
StudFileToMas( FileName , &pStudMas , &Razm );
Student * pTemp = (Student *) new Student [ Razm + 1];
        memcpy( pTemp , &S, sizeof(Student) );
for ( int i = 1 ; i <= Razm ; i++)
            memcpy( pTemp + i , pStudMas + (i - 1), sizeof(Student) );
//
StudMasToFile( FileName , pTemp , Razm + 1 );
delete [] pTemp;
delete [] pStudMas;
return;};
return;
};
//////////
// Добавления по номеру в файл
// МУ п.23.3

```

```

////////////////////////////////////
void StudAddNum(const char * FileName , Student S , int Numb )
{
    Student * pStudMas;
    int Razm;
    if ( Numb < 0) return;
    // Формирование массива
    StudFileToMas( FileName , &pStudMas , &Razm );
    if (Numb >= Razm )
    {
        StudAdd(FileName, S , Last );
        delete [] pStudMas;
        return;
    };
    //
    Student * pTemp = (Student *) new Student [ Razm + 1];

    for ( int i = 0 , k=0 ; i <= Razm ; i++, k++)
    {
        if ( ( i - 1) == Numb)
        {
            memcpy( pTemp + i , &S, sizeof(Student) );
            k--;
        }
        else
        {
            memcpy( pTemp + i , pStudMas + k, sizeof(Student) );
        };
    };
    // Запоминание в файл нового
    StudMasToFile( FileName , pTemp , Razm + 1 );
    delete [] pTemp;
    delete [] pStudMas;
};
////////////////////////////////////
// Удаление первого или последнего из файла
// МУ п.23.4/5
////////////////////////////////////
void StudDel(const char * FileName , TypeAddDel TAD = Last )
{
    Student * pStudMas;
    int Razm;
    if ( TAD == Last)
    {
        StudFileToMas( FileName , &pStudMas , &Razm );
        StudMasToFile( FileName , pStudMas , Razm - 1 );
        delete [] pStudMas;
        return;
    };
    if ( TAD == First )
    {
        StudFileToMas( FileName , &pStudMas , &Razm );
        StudMasToFile( FileName , pStudMas + 1 , Razm - 1 );
        delete [] pStudMas;
    }

};
////////////////////////////////////
// Удаление по номеру из файла
// МУ п.23.6

```

```

////////////////////////////////////
void StudDelNum(const char * FileName , int Numb )
{
    Student * pStudMas;
    int Razm;
    StudFileToMas( FileName , &pStudMas , &Razm );
    if ( Numb < 0 || Numb > Razm ) return;
    if ( Numb == 0 ) {
        StudDel( FileName , First );
        delete [] pStudMas;
        return;};
    if ( Numb == Razm -1 ) {
        StudDel( FileName , Last );
        delete [] pStudMas;
        return;
    };
    // Удаление из середины
    Student * pTemp = (Student *) new Student [ Razm - 1];

    for (int i =0, k =0 ; i < Razm ; i++ , k++)
    {
        if ( i == Numb )
            k--;
        else
            memcpy( pTemp + k , pStudMas + i , sizeof(Student) );
    };
    StudMasToFile( FileName , pTemp , Razm - 1 );
    delete [] pTemp;
    delete [] pStudMas;
    };
    //
    //////////////////////////////////////
    // Поиск и выборка одной записи по номеру
    // МУ п.23.7
    //////////////////////////////////////
    void FindStudNum( const char * FileName , Student * pS , int Numb){
        Student * pStudMas;
        int Razm;
        StudFileToMas( FileName , &pStudMas , &Razm );
        if ( Numb < 0 || Numb >= Razm )
        {
            pS->Name[0] = '\0';
            pS->Num = NULL;
            pS->Oklad = NULL;
        }
        else
            memcpy( pS , pStudMas + Numb , sizeof(Student) );
        delete [] pStudMas;
    };
    //
    //////////////////////////////////////
    // МУ п.23.8
    //////////////////////////////////////
    void ChangeStudNum( const char * FileName , Student * pS , int Numb){
        Student * pStudMas;
        int Razm;
        StudFileToMas( FileName , &pStudMas , &Razm );
        if ( Numb < 0 || Numb >= Razm )
        {
            delete [] pStudMas;
            return;
        }
    };
}

```

```

}
else
    memcpy( pStudMas + Numb , &pS, sizeof(Student) );
StudMasToFile( FileName , pStudMas , Razm );
delete [] pStudMas;
};
//////////
// Поиск по имени содержимого записей в файле в файле (первая запись)
// МУ п.23.9
//////////
int FindStudName( const char * FileName , Student * pS , const char * FindName )
{
    Student * pStudMas;
    int Razm;
    int i ;
    StudFileToMas( FileName , &pStudMas , &Razm );
    for ( i = 0 ; i < Razm ; i++)
        if ( strcmp ( (pStudMas + i)->Name , FindName ) == 0) break;
    if ( i == Razm )
    {
        // Не найдено
        pS->Name[0] = '\0';
        pS->Num = NULL;
        pS->Oklad = NULL;
        delete [] pStudMas;
        return -1 ;
    }
    // Найдено
    else
        memcpy( pS , pStudMas + i , sizeof(Student) );

        delete [] pStudMas;
        return i ;
};
//////////
// Замена по номерам записей в файле
//////////
// МУ п.23.10
//////////
void SwapStudFile( const char * FileName , int NumA ,int NumB)
{
    Student * pStudMas;
    int Razm;
    int i ;
    StudFileToMas( FileName , &pStudMas , &Razm );
    if ( NumA < 0 || NumA >= Razm || NumB < 0 || NumB >= Razm )
    { }
    else
    {SwapStudent ( pStudMas + NumA , pStudMas + NumB );
    StudMasToFile( FileName , pStudMas , Razm );};
    //
    delete [] pStudMas;
};
//////////
//////////
// МУ п.17/23.11
//////////
void StudRemoveFile(const char * FileName)
{
    char Comand[40];

```

// Снятие защиты с файла - отключение Readonly

```
strcpy (Comand , "attrib -R ");
strcat (Comand , FileName);
system( Comand );
strcpy (Comand , "attrib -A ");
strcat (Comand , FileName);
system( Comand );
```

// Удаление файла все дескрипторы файла должны быть закрыты

```
strcpy (Comand , "del ");
strcat (Comand , FileName);
system( Comand );
};
```

17.3. Результаты работы примера (текст)

Пункт МУ 4:

Stud2: Имя = Аксенова Номер = 2 Стипендия = 2000.00

Пункт МУ 5:

Запись: Имя = Лаптева Номер = 1 Стипендия = 1000.00

Запись: Имя = Аксенова Номер = 2 Стипендия = 2000.00

Пункт МУ 6:

Запись: Имя = Большаков Номер = 3 Стипендия = 3000.00

Пункт МУ 7:

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

Пункт МУ 8:

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

Пункт МУ 9:

Запись: Имя = Stud № - 12 Номер = 3 Стипендия = 3348.19

Запись: Имя = Stud № - 16 Номер = 7 Стипендия = 8209.78

Запись: Имя = Stud № - 20 Номер = 3 Стипендия = 7145.30

Запись: Имя = Stud № - 3 Номер = 1 Стипендия = 720.85

Запись: Имя = Stud № - 22 Номер = 3 Стипендия = 1234.78

Запись: Имя = Stud № - 15 Номер = 7 Стипендия = 9540.09

Пункт МУ 11 (синий) :

Запись: Имя = Запись 1 Номер = 1 Стипендия = 1000.00

Запись: Имя = Запись 2 Номер = 2 Стипендия = 2000.00

Запись: Имя = Запись 3 Номер = 3 Стипендия = 3000.00

Запись: Имя = Запись 4 Номер = 4 Стипендия = 4000.00

Пункт МУ 11 (Печать файла) :

Запись: Имя = Запись 1 Номер = 1 Стипендия = 1000.00

Запись: Имя = Запись 2 Номер = 2 Стипендия = 2000.00

Запись: Имя = Запись 3 Номер = 3 Стипендия = 3000.00

Запись: Имя = Запись 4 Номер = 4 Стипендия = 4000.00

Пункт МУ 11 (Печать всего массива) :

Запись: Имя = Запись 1 Номер = 1 Стипендия = 1000.00

Запись: Имя = Запись 2 Номер = 2 Стипендия = 2000.00

Запись: Имя = Запись 3 Номер = 3 Стипендия = 3000.00

Запись: Имя = Запись 4 Номер = 4 Стипендия = 4000.00

Пункт МУ 11:

Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

Запись: Имя = Запись № - 2 Номер = 6 Стипендия = 7522.20

Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

Пункт МУ 12:

Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

Запись: Имя = Запись № - 2 Номер = 6 Стипендия = 7522.20

Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

Пункт МУ 13:

После редактирования массива и запоминания в файле

Запись: Имя = Запись 1 Номер = 1 Стипендия = 1000.00

Запись: Имя = Изменение 2-й Номер = 5 Стипендия = 5000.00

Запись: Имя = Запись 3 Номер = 3 Стипендия = 3000.00

Запись: Имя = Запись 4 Номер = 4 Стипендия = 4000.00

Пункт МУ 14:

После редактирования массива и запоминания - распечатка из файла функцией:

1 - Запись: Имя = Запись 1 Номер = 1 Стипендия = 1000.00

2 - Запись: Имя = Изменение 2-й Номер = 5 Стипендия = 5000.00

3 - Запись: Имя = Запись 3 Номер = 3 Стипендия = 3000.00

4 - Запись: Имя = Запись 4 Номер = 4 Стипендия = 4000.00

Пункт МУ 15:

Число записей в БД = 4

Пункт МУ 13:

1 - Запись: Имя = Иванов Номер = 3 Стипендия = 7000.00

2 - Запись: Имя = Петров Номер = 2 Стипендия = 2000.00

3 - Запись: Имя = Сидоров Номер = 1 Стипендия = 3000.00

4 - Запись: Имя = Печкин Номер = 4 Стипендия = 4000.00

Пункт МУ 16:

Записей в файле нет!

Пункт МУ 17:

Записей в файле нет!

Пункт МУ 17 снова добавим записи из массива:

1 - Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

2 - Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

3 - Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

4 - Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

Пункт МУ 18 (Sawp - до):

Запись: Имя = Лаптева Номер = 1 Стипендия = 1000.00

Запись: Имя = Иванова Номер = 2 Стипендия = 2000.00

Пункт МУ 18 (Sawp - после):

Запись: Имя = Иванова Номер = 2 Стипендия = 2000.00

Запись: Имя = Лаптева Номер = 1 Стипендия = 1000.00

Пункт МУ 18 (Sawp - pStMas - до):

Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

Пункт МУ 18 (Sawp - pStMas - после):

Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

Пункт МУ 19 (SORT - pStMas - до):

Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

Пункт МУ 19 (SORT - pStMas - после Oklad):

Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

Пункт МУ 19 (SORT - pStMas - после Name):

Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

Пункт МУ 19 (SORT - pStMas - после Num):

Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

Пункт МУ 20 (SORT - функция - до):

Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

Пункт МУ 20 (SORT - функция - после):

Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

Пункт МУ 21 (SORT - с разными функциями):

СОРТИРОВКА: Работа с функциями Name:

1 - Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

2 - Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

3 - Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

4 - Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

СОРТИРОВКА: Работа с функциями Num (убывание):

1 - Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

2 - Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

3 - Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

4 - Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

СОРТИРОВКА: Работа с функциями Oklad (возрастание):

1 - Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

2 - Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

3 - Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

4 - Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

Пункт МУ 22:

Чтение из программы (номер = 3):

Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

Чтение из функции (номер = 1):

Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

Пункт МУ 22.2 (Интегральные расчеты):

Фонд зарплаты в цикле = 27480.70

Фонд зарплаты из функции = 27480.70

Добавление (начало):

1 - Запись: Имя = Первый Номер = 22 Стипендия = 3400.00

2 - Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

3 - Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

4 - Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

5 - Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

Добавление (конец):

1 - Запись: Имя = Первый Номер = 22 Стипендия = 3400.00

2 - Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

3 - Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

4 - Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

5 - Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

6 - Запись: Имя = Последний Номер = 22 Стипендия = 3400.00

Добавление (номер):

1 - Запись: Имя = Первый Номер = 22 Стипендия = 3400.00

2 - Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

3 - Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

4 - Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

5 - Запись: Имя = По номеру - 2 Номер = 33 Стипендия = 5500.00

6 - Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

7 - Запись: Имя = Последний Номер = 22 Стипендия = 3400.00

1 - Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

2 - Запись: Имя = Запись № - 4 Номер = 2 Стипендия = 7055.57

3 - Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

4 - Запись: Имя = По номеру - 2 Номер = 33 Стипендия = 5500.00

5 - Запись: Имя = Запись № - 3 Номер = 2 Стипендия = 9526.05

6 - Запись: Имя = Последний Номер = 22 Стипендия = 3400.00

Удаление (конец):

1 - Запись: Имя = Запись № - 1 Номер = 0 Стипендия = 3376.87

2 - Запись: Имя = Запись № - 4	Номер = 2	Стипендия = 7055.57
3 - Запись: Имя = Новая	Номер = 6	Стипендия = 7522.20
4 - Запись: Имя = По номеру - 2	Номер = 33	Стипендия = 5500.00
5 - Запись: Имя = Запись № - 3	Номер = 2	Стипендия = 9526.05

Удаление (номер):

1 - Запись: Имя = Запись № - 1	Номер = 0	Стипендия = 3376.87
2 - Запись: Имя = Запись № - 4	Номер = 2	Стипендия = 7055.57
3 - Запись: Имя = Новая	Номер = 6	Стипендия = 7522.20
4 - Запись: Имя = По номеру - 2	Номер = 33	Стипендия = 5500.00

Чтение (номер):

Запись: Имя = Новая Номер = 6 Стипендия = 7522.20

Изменение записи (номер):

1 - Запись: Имя = Запись № - 1	Номер = 0	Стипендия = 3376.87
2 - Запись: Имя = Запись № - 4	Номер = 2	Стипендия = 7055.57
3 - Запись: Имя = Замена	Номер = 11	Стипендия = 111.00
4 - Запись: Имя = По номеру - 2	Номер = 33	Стипендия = 5500.00

Поиск по имени:

Найдена по номеру = 2

Найдена запись: Запись: Имя = Замена Номер = 11 Стипендия =

111.00

Замена SWAP в файле по номеру: 1<->3

1 - Запись: Имя = Запись № - 1	Номер = 0	Стипендия = 3376.87
2 - Запись: Имя = По номеру - 2	Номер = 33	Стипендия = 5500.00
3 - Запись: Имя = Замена	Номер = 11	Стипендия = 111.00
4 - Запись: Имя = Запись № - 4	Номер = 2	Стипендия = 7055.57

Для продолжения нажмите любую клавишу . . .

17.4. Шаблон отчета по ЛР10

Смотрите на сайте файлы шаблона отчета и примера (Шаблон отчета ЛР_ОП.doc ,
Пример отчета ЛР_ОП.doc).