МИНИСТЕРСТВО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ИНСТИТУТ РАДИОТЕХНИКИ, ЭЛЕКТРОНИКИ И АВТОМАТИКИ
(ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ)»

ОПЕРАЦИОННЫЕ СИСТЕМЫ

(подраздел "Программирование в операционной среде")

ЯЗЫК СИ

МОСКВА 2004

МИНИСТЕРСТВО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ИНСТИТУТ
РАДИОТЕХНИКИ, ЭЛЕКТРОНИКИ И АВТОМАТИКИ
(ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ)»

ОПЕРАЦИОННЫЕ СИСТЕМЫ

(подраздел "Программирование в операционной среде")

ЯЗЫК СИ

Методические указания
по выполнению лабораторных работ
для студентов, обучающихся
по специальностям 075200 (второй курс),
075600 (третий курс)

МОСКВА 2004

Составители: В.П.Челноков, М.Ж.Акжолов, А.Г.Ветров, И.С. Исаев, И.А. Кузьмин, Д.А. Лысенко, А.Г.Мадера, А.И. Малинин, А.И. Тихонов

Редактор А.Н.Сотников

Методические указания по курсу "ОПЕРАЦИОННЫЕ СИСТЕМЫ" (подраздел "Программирование в операционной среде") "ЯЗЫК СИ" содержат расширенную теоретическую программу по указанному курсу, тематику лекций и семинарских занятий, много примеров программ на языке Си, иллюстрирующих основные возможности этого языка, задания для лабораторных работ, а также список литературы. Предполагается, что студенты освоили язык Паскаль в рамках предыдущих курсов.

Материал предназначен для студентов дневного отделения и может быть использован для самостоятельной работы при освоении базового курса кафедры Прикладного Программного Обеспечения (ППО).

Печатаются по решению редакционно-издательского совета университета.

Рецензенты: к.т.н., профессор Сургуладзе М.Ш.
д.т.н., профессор Торчигин В. П.

Методические указания напечатаны в авторской редакции

Подписано в печать 16.01.2004. Формат 60x84 1/16.
Бумага офсетная. Печать офсетная.
Усл. печ. л.1,63 Усл. кр.-отт. 6,52. Уч.-изд. л. 1,5.
Тираж 200 экз. С 44

Государственное образовательное учреждение
высшего профессионального образования
“Московский государственный институт радиотехники,
электроники и автоматики (технический университет)”
119454, Москва, пр. Вернадского, 78

ВВЕДЕНИЕ

Необходимость знания и умения использовать популярнейший язык программирования Си не вызывает никаких сомнений в век всеобщей компьютеризации. Действительно, все системные вызовы известных операционных систем (ОС) написаны на этом языке. Для более углубленного освоения языка Си необходимо также хорошее знание языка систематического программирования Паскаль, чтобы понимать причины введения в язык Си тех или иных механизмов. По существу этот курс является попыткой обучения студентов систематическому программированию на языке Си.

Целью проводимых лабораторных работ является закрепление знаний по языку Си, приобретению практических навыков в использовании программного обеспечения и освоению управления ОС UNIX.

Пособие включает описание пяти лабораторных работ:

ПОСТРОЕНИЕ ПРОСТОЙ ПРОГРАММЫ НА ЯЗЫКЕ СИ И ИССЛЕДОВАНИЕ ЕЕ РАБОТЫ;

ИСПОЛЬЗОВАНИЕ СТРУКТУРНЫХ ОПЕРАТОРОВ И КОНСТРУКТОРОВ ДАННЫХ ЯЗЫКА СИ;

ИСПОЛЬЗОВАНИЕ СЛАБОЙ ТИПИЗАЦИИ ЯЗЫКА СИ;

СВЯЗЬ МЕЖДУ МАССИВАМИ И УКАЗАТЕЛЯМИ В ЯЗЫКЕ СИ;

ИСПОЛЬЗОВАНИЕ ВОЗМОЖНОСТЕЙ ОПЕРАЦИЙ ЯЗЫКА СИ И ПРЕПРОЦЕССОРА.

Каждая работа рассчитана на 4-8 учебных часов и включает следующие основные этапы:

углубление и закрепление теоретических знаний;

анализ поставленной задачи и подготовка к ее выполнению;

непосредственная работа на компьютере;

оформление и представление отчета.

Теоретическая часть описания лабораторной работы при необходимости может использоваться для самостоятельного изучения постановок задач и методов их реализации.

1. Лабораторная работа "ПОСТРОЕНИЕ ПРОСТОЙ ПРОГРАММЫ НА ЯЗЫКЕ СИ И ИССЛЕДОВАНИЕ ЕЕ РАБОТЫ" (1 занятие)

1.1. Основные теоретические положения

1.1.1. Классификация языков программирования и место языка Си среди этих языков

Языки программирования делятся на три основные группы: *языки высокого уровня* (Паскаль, Модула, Ада и т.д.), *языки низкого уровня* (ассемблеры) и *языки среднего уровня* (Си).

Достоинством языков высокого уровня является предоставление пользователям возможности программирования, не зная архитектуры конкретного компьютера, и, тем самым, обеспечение *мобильности* программ, написанных на этих языках (т.е. программа может быть без переделок перенесена на другой компьютер и там сразу же заработать после перетрансляции). Основным недостатком этих языков, является недостаточная эффективность создаваемых программ.

И, наоборот, достоинством языков *ассемблеров* является возможность составления очень эффективных программ, поскольку ассемблер - это, по существу, тот же язык машинных команд конкретного компьютера. Зная архитектуру конкретного компьютера, всегда можно при должной квалификации составить очень эффективную программу. Однако эта программа не является мобильной, так как в общем случае ее нельзя перенести на другой компьютер, имеющий другую систему команд. Таким образом, основным недостатком ассемблера является зависимость его программ от платформы.

Язык Си является языком среднего уровня. Что это значит? Этот язык, с одной стороны, поддерживает мобильность программ, написанных на этом языке, а, с другой стороны, обеспечивает также достаточную эффективность выполняемых программ.

В настоящее время этот язык получил широкое распространение и в основном вытеснил ассемблеры. Сейчас основным языком реализации операционных систем является именно язык Си. Этот язык стандартизован и уже в течение 10 лет почти не изменяется. Дело в том, что в языке Си все новые возможности реализуются путем составление новых библиотек, а не с помощью расширения самого языка.

Библиотекой языка программирования называется набор оттранслированных функций, написанных на этом языке и предназначенных для выполнения конкретных задач. Пример такой библиотеки: *библиотека системных вызовов* ОС UNIX. В языке Си каждая библиотека дополняется одним или несколькими *include*-файлами, содержащими спецификации функций этой библиотеки или их макрореализации. Если в вашей программе имеются вызовы функций некоторой библиотеки, то, пожалуйста, указывайте в начале своей программы *include*-операторы для подключения *include*-файлов этой библиотеки. Это обеспечит проверку правильности обращения к функциям этой библиотеки (более подробное описание *include*-операторов см. ниже в теоретической части лабораторных работ).

1.1.2. Структура файла с программой на языке среднего уровня Си

Программа на языке Си состоит из одного или нескольких файлов. Простую программу на языке Си проще всего составить в виде одного файла. Этот файл обычно включает в себя:

include-операторы с указанием тех *include-файлов*, которые содержат описания функций стандартной или иной библиотеки, вызываемых в создаваемой программе;

список глобальных переменных, существующих в течение всего времени выполнения данной программы;

список определений функций данной программы.

Среди этих функций обязательно должна быть функция *main()* - стартовая точка программы. Именно с этой функции начинается выполнение программы. Полный интерфейс функции *main()* с операционной системой будет описан ниже, в теоретической части последней лабораторной работы, а пока будем считать, что эта функция не имеет параметров и не возвращает результата (т.е. ее тип результата равен *void*; это означает, что функция *main()* в данном примере рассматривается как процедура).

Далее по тексту будем называть функцию или процедуру, когда не имеет смысла их различать, просто *подпрограммой*.

Приведем пример простой программы на языке Си, которая расписывает значением массив и выдает на экран сообщение. Заметим, что комментарии в программе на языке Си указываются двумя способами: с помощью символов //, за которыми следует текст комментария на одной строке, либо с помощью скобок комментария: /* текст комментария */; в последнем случае текст комментария может занимать несколько строк:

```
#include <stdio.h>
//файл описаний функций стандартного ввода/вывода,
//так как в программе используется вызов printf()

int M[ 100];//глобальный массив Миз 100 целых чисел

int f(int par)
//определение функции/, записывающей во все элементы
//массива M значение параметра par
{ //начало тела функции f
int i;//локальная переменная i типа int
for (i = 0; i < 100; i++)//запись значения par в массив
   M[i] = par;
return 1;//возврат из функции f значения 1 }//end f

void main() //определение стартовой функции main() {
//начало тела функции main()
 int q;//локальная переменная q типа int
q =f(12);//вызов функции f и присваивание ее результата q

printf("Hello world; q= %d \n",q);
//вывод строки:Hello world; q= <знач. q><новая строка>
//символ < новая строка> обозначается как \n
}//end main
```

Заметим, что тело любой функции заключается в фигурные скобки "{}". Сохраним эту программу в файле "<*имя_файла*>.*c*". Собственно, именно имя этого файла и будет являться именем данной программы, а суффикс ".*c*" указывает, что в этом файле

хранится программа на языке Си.

Затем откомпилируем эту программу в выполняемый файл и, запустив его, получим нужный результат.

1.1.3. Отличия языка Си от языка Паскаль (только для начинающих студентов)

Обычно студенты приступают к изучению языка Си после изучения языка Паскаль. Язык Паскаль является языком систематического программирования и специально был создан Н.Виртом для обучения структурному программированию. *Структурное программирование* - это такой стиль написания программ, который обеспечивает следующее: из чтения текста созданной программы (т.е. из *статики* программы) сразу же становится понятной динамика ее выполнения.

Для поддержки структурного программирования Н.Вирт ввел в язык Паскаль все разумные виды структурных операторов и конструкторов данных, а также призвал использовать оператор *GOTO* только в редких случаях (обычно при возникновении ошибок и ни в коем случае для организации циклов). Кроме того, в язык Паскаль были - впервые для языков программирования - введены *типы данных* для повышения надежности программирования (о типах данных мы поговорим более подробно ниже).

Поэтому сначала мы кратко изложим основные отличия языка Си от языка Паскаль, которые будут полезны для студентов, начинающих изучение языка Си и уже освоивших язык Паскаль.

При этом мы не будем углубляться в тонкости языка Си, принципиально отличающие его от языка Паскаль. Пока наша задача другая - помочь студенту написать на языке Си программу, которую он уже создавал на языке Паскаль:

в языке Си нет процедур, а есть только функции; поэтому, если требуется написать процедуру, создавайте функцию, тип результата которой равен *void*, результат вызова такой функции не требуется присваивать никакой переменной;

в языке Си нет оператора присваивания, а есть операция присваивания (обозначается с помощью символа "="); результат этой операции можно использовать в другой операции или при-

своить другой переменной, например: *a = b = 12*13*; заметим, что операция сравнения на равенство (неравенство) обозначается в Си с помощью символов "==" ("!=");

в языке Си запрещена вложенность функций;

в языке Си все параметры передаются функциям только по значению (в частности, нет параметра, передаваемого по ссылке и обозначаемого в языке Паскаль с помощью спецификатора *var*: например, *function f (var c: real): real)*;

если в программе на языке Си используется вызов какой-либо стандартной функции, в начале текста этой программы следует указать *include*-оператор с указанием того *include-файла*, который содержит описание этой функции; например, макрооператор *#include <cstring.h>* подключает описания функций обработки строк; если это не сделать, то компилятор выдаст ошибку, поскольку не сможет проверить правильность обращения к стандартной функции;

в вызове функции стандартного вывода *printf()* сначала указывается строка - формата (в качестве первого параметра); эта

строка содержит обычные символы, которые просто копируются в выходной поток, и спецификации преобразования, каждая из которых вызывает преобразование и печать очередного параметра вызова *printf()*; например, вывод значений целой переменной *i* и вещественной переменной *f* мог бы выглядеть так: *printf(" i = %d; f = %f;\n", i, f)*; спецификация преобразования обозначается с помощью символа "%" и следующих за ним одного или более символов *(%d* и *%f* этом примере);

функция стандартного ввода *scanf()* представляет собой вводной аналог функции *printf()*; в ней предусмотрены многие из только что описанных преобразований, но выполняемые в обратном направлении; вызов этой функции читает символы из стандартного ввода, интерпретирует их в соответствии со спецификациями строки-формата и записывает результаты в соответствующие параметры; каждый из этих параметров должен быть адресом некоторой переменной; например, ввод значений в целую переменную *i* и вещественную переменную *f* мог бы выглядеть так: *scanf("%d %f", &i, &f)*; спецификация преобразования обозначается с помощью символа "%" и следующих за ним одного или более символов *(%d* и *%f* этом примере); символ "*&*" обозначает унарную операцию взятия адреса (*&i* и *&f* в этом примере).

Полное описание языка Си см. в [1].

Заметим также, что далее в теоретических частях лабораторных работ при описании того или иного механизма языка Си мы часто будем сравнивать возможности этого механизма с соответствующими возможностями языка Паскаль. Это требуется для обоснования необходимости введения этого механизма в язык Си.

1.1.4. Общая схема получения выполняемой программы из исходной программы, написанной на языке Си

Сначала исходная программа поступает на вход *препроцессора* (который по существу является *макрогенератором*). Назначение препроцессора - это макробработка текста исходной программы. Поэтому после процессора мы получаем текстовый файл, содержащий только конструкции языка Си и поэтому готовый к компиляции.

Затем полученная программа транслируется с помощью компилятора, чтобы преобразовать операторы и операции этой программы в машинные команды объектной машины, которые выполняют то, что предписано исходной программой, но только на языке объектной машины (т.е. машины, на которой будет работать откомпилированная программа). Результатом компиляции является объектный файл, имеющий двоичный формат. Этот файл еще не готов к работе, так как он в общем случае содержит вызовы подпрограмм, определения которых отсутствуют в исходной программе.

Поэтому следующим этапом является *компоновка* (или линковка) объектного файла. Она заключается в следующем: определения тех подпрограмм, которые используются в исходной программе, но отсутствуют в ней, разыскиваются среди объектных файлов: либо в указанном каталоге (если этот файл был создан самим программистом), либо в каталогах стандартных библиотек. Как только определение подпрограммы в виде объектного файла будет найдено, этот объектный файл объединяется с исходным

объектным файлом, а ссылка на определение функции переключается (т.е. замыкается) на подключаемое определение функции. Работа компоновщика продолжается до тех пор, пока все внешние ссылки не будут замкнуты.

Результатом компоновщика является *загрузочный* файл (двоичного формата), объем которого по только что объясненным причинам больше объема объектного файла. Этот файл почти готов к работе, только иногда нуждается в некоторых настройках. Если требуется запустить загрузочный файл, то он передается загрузчику, который размещает его в оперативной памяти, при необходимости настраивает его на конкретные адреса размещения в памяти, помещает значения инициализирующих выражений в глобальные переменные и запускает его, передавая управление стартовой точке программы. Именно такая программа и является выполняемой (часто выполняемым называют загрузочный файл).

1.1.5. Представление чисел в программе языка Си

1.1.5.1. Системы счисления, используемые в языке Си

В языке Си числовые константы можно представлять в следующих системах счисления:

десятичной (база системы счисления *10*);

шестнадцатеричной (база системы счисления *16*);

восьмеричной (база системы счисления *8*).

Числа в *десятичной системе счисления* обозначаются обычным образом с помощью десятичных цифр, только запрещается указывать нули перед первой значащей (ненулевой) цифрой десятичного числа. Примеры: *189, 100000, 5689*.

Числа в *шестнадцатеричной системе счисления* используются для компактного представления двоичных чисел (чисел с базой системы счисления *2*). Как будет показано ниже, двоичные числовые константы применяются для выполнения побитовых логических операций языка Си.

В шестнадцатеричной системе счисления используются следующие цифры: обычные десятичные цифры от *0* до *9*, а также прописные или строчные первые буквы латинского алфавита: *A, B, C, D, E, F* для обозначения следующих шестнадцатеричных

цифр *10, 11, 12, 13, 14, 15,* соответственно. Перед шестнадцатеричным числом указывается префикс 0x. Пример: шестнадцатеричное число *0xFf* равно двоичному числу *11111111* или десятичному числу *255*. Как можно видеть из этого примера, шестнадцатеричное представление двоичного числа содержит существенно меньше цифр, т.е. более компактно по сравнению с двоичным числом.

Числа в *восьмеричной системе счисления* также используются для компактного представления двоичных чисел. В этой системе счисления при обозначении числа используются обычные десятичные цифры от *0* до *7*. Чтобы отличать восьмеричное число от десятичного, перед восьмеричным числом указывается в качестве префикса цифра 0. Пример: восьмеричное число *0377* равно двоичному числу *11111111* или десятичному числу *255*. Как можно видеть из этого примера, восьмеричное представление двоичного числа содержит существенно меньше цифр, т.е. более компактно по сравнению с двоичным числом.

1.1.5.2. Быстрый перевод чисел из восьмеричной системы счисления в шестнадцатеричную или наоборот

Как только что было сказано выше, двоичное представление чисел в языке Си напрямую не поддерживается, однако - поддерживается косвенно с помощью шестнадцатеричного или восьмеричного представления, так как два последних представления более компактны по сравнению с двоичным представлением.

Если требуется перевести число из двоичной системы счисления в восьмеричную (шестнадцатеричную), то сначала выделите в исходном двоичном числе СПРАВА НАЛЕВО трупы цифр - по три (четыре) двоичных цифры, а затем замените каждую группу двоичных цифр на соответствующую восьмеричную (шестнадцатеричную) цифру. Приведем примеры:

Исходное двоичное число: 1010111110000101

1) перевод двоичного числа в восьмеричную систему
1 010 111 110 000 101 -"тройки" двоичного числа

1 2 7 6 0 5 - восьмеричное число 0127605

2) перевод того же числа в шестнадцатеричную систему
1010 1111 1000 0101 -"четверки" двоичного числа

A F 8 5 - шестнадцатеричное число 0xAF85

Если при выделении групп цифр в двоичном числе в последней группе не хватает слева цифр, дополните последнюю группу слева нулями.

Если требуется перевести число из восьмеричной (шестнадцатеричной) системы счисления в шестнадцатеричную (восьмеричную) систему счисления, то сначала переведите исходное число в двоичное представление, заменяя восьмеричные (шестнадцатеричные) цифры на соответствующие труппы двоичных цифр, а затем переведите полученное двоичное число в нужную систему счисления так, как было показано выше. Приведем пример:

перевод восьмеричного числа в двоичную систему
1 2 7 6 0 5 - восьмеричное число 0127605

001 010 111 110 000 101 -"тройки" двоичного числа

перевод двоичного числа в шестнадцатеричную систему
1010 1111 1000 0101 -"четверки" двоичного числа

A F 8 5 - шестнадцатеричное число 0xAF85

Можно также переводить десятичные числа в двоичные, восьмеричные или шестнадцатеричные числа, однако это делается более сложным способом и обычно не требуется. Это объясняется тем, что для представления логических кодов десятичные числа не удобны (их база системы счисления *10* не является целой степенью числа *2*).

1.1.6. Представление символов в программе языка Си

1.1.6.1. Строки и коды символов

В языке Си можно использовать строки символов и коды символов. Строка символов - это последовательность кодов сим-

волов, заканчивающаяся байтом, все биты которого равны 0 (*NULL-байт*). Строка символов имеет тип *char* *, т.е. является указателем на 0-й символ этой строки. Строка символов обычно размещается в памяти.

Литеральная строка символов заключается в двойные кавычки, например, "*nana*". Эта строка занимает 5 байтов, причем первые байты занимают последовательно коды символов ‘*n*’, ‘*a*’, '*n*', '*a*', а последним байтом является *NULL-байт*. Более подробное описание строк символов см. ниже при описании обработки массивов символов с помощью указателей.

Код символов - это последовательность кодов указанных символов, обычно помещающаяся в формате типа *unsigned int*. Поэтому длина кода символов не должна превышать размер этого типа. В отличие от строки символов, код символов заключается в одинарные кавычки (например, '*q*', '*nana*' или '*0*'- код цифры 0) и имеет тип *unsigned int*. Правда, если код символов состоит из одного символа, то он имеет тип *char*, а длинный код символов может иметь тип *unsigned long*.

Коды символов используются для выполнения целочисленных операций. Например, код одного символа (например, '*a*') может использоваться для сравнения. Приведем пример:

```
char c;//определение переменной типа char
……..
if (c == 'a')//проверка: равна ли с коду 'a'
    c='b';
```

1.1.6.2. Прямое указание кодов символов

Обычно в строке или коде символов: символы указываются литерально, например: *'a'*. Такой способ задания символов очень удобен и не зависит от используемой кодировки.

Иногда требуется указывать специальные управляющие символы или коды символов конкретной кодировки. В этом случае впереди ставится экранирующий символ ‘\’ (обратный слеш), который указывает, что далее стоит не литеральный символ, а либо специальный символ, обозначающий конкретный систем-

ный символ, либо код символа в используемой кодировке. Приведем примеры:

\n - символ новой строки
\t - символ табуляции
\012 - символ, код которого равен 12 в 8-ричной системе

Если требуется указать литеральный символ '\' (обратный слеш), используйте конструкцию '\\'. Более подробное описание кодировок см. в теоретических частях других лабораторных работ.

1.2. Условия и общий порядок выполнения работы

В лабораторной работе можно выделить четыре этапа: подготовка к выполнению работы, контроль преподавателем готовности обучаемого к выполнению работы, непосредственное выполнение работы на компьютере и защита отчета о лабораторной работе.

В процессе подготовки к выполнению работы студенты на основе теоретического раздела описания углубляют и систематизируют свои знания по методам создания приложений. На это также направлена разработка формальной постановки предлагаемого задания.

В процессе собеседования студента с преподавателем проверяется готовность студента к выполнению работы. При необходимости проводятся дополнительные консультации со студентом.

В ходе непосредственного выполнения лабораторной работы студенты используют компьютеры кафедры, а также свои собственные компьютеры для ускорения выполнения работы. Кафедра предоставляет консультации по установке нужного программного обеспечения.

В заключительный период работы студенты демонстрируют преподавателю работу своего приложения, а также отвечают на его вопросы.

По результатам лабораторной работы студент оформляет и предоставляет отчет в произвольной форме, который включает название этой работы, фамилию и инициалы студента, формальную постановку задания, анализ результатов работы и результирующий исходный текст приложения на дискете.

На этапе непосредственного выполнения лабораторной работы студентам рекомендуется работать группами по 8 человек, обеспечивая взаимную помощь друг другу; преподаватель и староста обеспечивают контроль за действиями студентов.

1.3. Примеры заданий

Написать программу перевода чисел из одной системы счисления в другую. Диапазон баз систем счисления: от *2* до *16*. Эта программа может быть легко составлена как на языке Си, так и на языке Паскаль. Советуем создать два варианта такой программы.

Приведем пример интерфейса подобной программы (информация, указываемая пользователем, выделена жирным шрифтом):

Введите исходную базу счисления: ***7***
Введите число: ***16***
Введите базу счисления для результата: ***6***
Результат: ***21***
Завершить работу (Д/Н): ***Д***
Работа программы закончена

2. Лабораторная работа "ИСПОЛЬЗОВАНИЕ СТРУКТУРНЫХ ОПЕРАТОРОВ И КОНСТРУКТОРОВ ДАННЫХ ЯЗЫКА СИ (1 занятие)

2.1. Основные теоретические положения

2.1.1. Определение принципа регулярности

В основу построения структурных операторов и конструкторов данных любого языка программирования (а не только Си) положен *принцип регулярности*. Он гласит: при создании любой программной конструкции на языке программирования вполне достаточно использовать только три операции: *следование*, *альтернатива* и *цикл*.

Именно поэтому в нормальных, самодостаточных языках программирования имеются три вида структурных операторов и три вида конструкторов данных. Рассмотрим их подробнее.

2.1.2. Структурные операторы языка Си

2.1.2.1. Блок операторов (реализует операцию следования)

Блок операторов используется для объединения нескольких операторов в один оператор и указания порядка выполнения этих операторов внутри блока. Этот порядок совпадает с порядком перечисления этих операторов в блоке.

В блоке можно также определять локальные переменные, *область видимости* и *область существования* которых ограничена этим блоком. Эти переменные можно также инициализировать. Приведем пример простого блока:

```
{//начало блока
 int a = 2, b = 3;
//определение/инициализация локальных переменных a и b
f(13);//выполнится первым
b=14;//выполнится вторым
a=a+b;//выполнится третьим
}//конец блока
```

Блок реализуется с помощью программного стека: при входе в блок для размещения его локальных переменных выделяется место в программном стеке, а при выходе из блока это место изымается из стека. Если в блоке нет локальных переменных, то для него не выделяется место в программном стеке.

Допускается произвольная вложенность блоков.

Язык Си допускает определение в блоке переменных, помеченных спецификатором *static*. Эти переменные существуют в течение всего времени работы программы (т.е. их область существования - вся программа), но доступны только внутри блока (т.е. их область видимости - блок, в котором они описаны, а также все вложенные в этот блок другие блоки, объявленные ниже определения этой переменной). Не советуем использовать подобные переменные, так как они противоречат принципам структуризации и не дают какого-либо выигрыша.

2.1.2.2. Операторы if и switch (реализуют операцию альтернативы)

Оператор *if* - самая общая форма оператора альтернативы. Он имеет следующий формат:

```
if (выражение)
  оператор1
else
  оператор2
```

Этот оператор выполняется следующим образом: сначала вычисляется его выражение, а затем его результат интерпретируется логически: если результат этой интерпретации равен *TRUE*, то выполняется *оператор1*; в противном случае выполняется *оператор2* (логическая интерпретация значений языка Си будет описана ниже в теоретической части этой лабораторной работы).

Допускается произвольная вложенность операторов *if*. Альтернатива *else* не является обязательной.

Оператор *switch* (переключатель) - это оптимизированный вариант такого оператора *if*, у которого выражение имеет цело численный тип, а альтернативы выбора можно связать с целочисленными константами. Приведем пример подобного оператора *if* и переделаем его в оператор *switch*:

```
if (выражение == const 1)
      оператор1
else
      if ((выражение == const2) || (выражение == const3))
      оператор2
else
      оператор3
```

В формате *switch* приведенный выше оператор *if* будет выглядеть так:

```
switch (выражение)
{
case const 1://альтернатива const 1
   оператор1
   break;
case const2://альтернатива const2
```

```
case const3://альтернатива const3
   оператор2
   break;
default://альтернатива - остальные случаи
   оператор3
}//end switch
```

В отличие от оператора *if*, в операторе *switch* выражение вычисляется только один раз, а затем сразу же осуществляется переход к выполнению оператора, который соответствует полученному значению выражения (а не последовательный перебор всех вариантов, как в приведенном выше операторе *if*).

Оператор *break* в операторе *switch* используется для прекращения выполнения операторов текущей альтернативы и передачи управления оператору, следующему по тексту за данным оператором *switch*. Если в операторе *switch* не указать оператор *break* в конце альтернативы, то после выполнения операторов этой альтернативы управление будет передано на первый оператор следующей по тексту альтернативы. Иногда это бывает полезно для оптимизации.

2.1.2.3. Операторы цикла (реализуют операцию цикла)

Операторы цикла языка Си подразделяются на *условные* и *итеративный* (*for*). Условные циклы в свою очередь подразделяются на *цикл с предусловием* (*while*) и *цикл постусловием* (*do..while*) (более подробное описание см. в [1]).

Приведем пример цикла с предусловием. В этом цикле сначала производится проверка условия, и если условие выполняется, то выполняется тело цикла; в противном случае управление передается оператору, следующему за оператором этого цикла (т.е. цикл прекращает свою работу). После выполнения тела цикла (т.е. после выполнения очередной итерации цикла) всегда производится проверка условия:

```
while (string[i] != 0)//условный цикл с предусловием
   printf("символ строки = %c\n", string[i]);
   //тело цикла - посимвольный вывод строки
```


Итеративный цикл *for* является аналогом цикла *for* языка Паскаль, предназначенного для реализации пошаговых итераций. Однако он имеет более гибкий формат. В общем случае цикл *for* языка Си эквивалентен приведенной ниже конструкции:

```
for (выражение1; выражение2; выражение3)
  оператор
```

Эквивалентная конструкция

```
выражение1;
while (выражение2)
 {
 оператор
  выражение3;
 }
```

Допускается произвольная вложенность циклов. При проектировании циклов наибольшее внимание уделяйте оптимизации вложенных циклов - помните правило Д.Кнута: *90%* времени выполнения программы происходит в *4%* ее кода (т.е. в самых вложенных циклах).

Для прерывания работы цикла используется оператор *break*, а для прерывания работы очередной итерации цикла и перехода к выполнению следующей итерации этого же цикла используется оператор *continue*. Если циклы вложены, то операторы *break* и *continue* действуют только на самый вложенный цикл (относительно размещения этих операторов внутри операторов циклов). Приведем пример:

```
for (I = 0; I < N; I++) //итеративный цикл
   {if(A[1] < 0) //пропуск отрицательных чисел;
      continue;
     //обработка положительных чисел
   }
```

В этом примере оператор *continue* используется для пропуска обработки отрицательных значений массива.

Если требуется прекратить работу внешнего цикла по отношению к данному, используйте оператор *goto*.

2.1.3. Конструкторы данных языка Си

2.1.3.1. Структура (реализует операцию следования)

2.1.3.1.1. Свойства структуры

Конструктор данных языка Си - *структура* - используется для объединения в одно целое нескольких компонент (в общем случае разных типов). Приведем пример:

```
struct{ s // имя этого типа struct s
{
      int    a;
      float  b;
      char *p;
};
```

Этот конструктор данных содержит три компонента, логический порядок которых задан при определении этого типа, т.е. при перечислении его компонент. Поэтому структура реализует *логическую операцию следования* (логическое размещение компонент).

Для сравнения в языке Паскаль имеется аналогичный конструктор данных *record*. Однако, в отличие от языка Паскаль, язык Си предполагает, что физическое размещение компонент в структуре совпадает с их логическим порядком. Поэтому структура реализует также и *физическую операцию следования* (это еще один пример слабой типизации языка Си, которая будет описана ниже в теоретической части одной из лабораторных работ).

Таким образом, язык Си предоставляет программисту инструмент для управления размещением компонент в памяти. В описанной выше структуре сначала будет располагаться компонент с именем *a* (пусть занимает *2* байта), затем компонент с именем *b* (пусть занимает *4* байта), а потом компонент с именем *p* (пусть занимает *4* байта). Таким образом, переменная этого типа будет занимать (*2+4+4=10*) байт.

Этот тип имеет имя *struct s*, где *s* - это *ярлык типа* (а не имя типа). Далее в программе - после определения этого типа - можно было бы объявить переменную этого типа и указатель на этот тип, а затем выбрать компоненты этих переменных:


```
struct s s, // переменная s, имеющая тип struct s
      *ps;//переменная  ps -указатель на struct s
……..

s. a;//операция, используемая для выборки компонента a
ps->a;
//операция ->, используемая для выборки компонента a
……
```

Вообще говоря, для выборки компонента структуры через указатель надо использовать следующую операцию:

```
(*ps).a;
```

однако для упрощения записи в языке Си введена операция “->”.

2.1.3.1.2. Битовые поля структуры

Для управления побитовым (или поразрядным) размещением компонент внутри структуры используются *битовые поля*. Битовое поле должно иметь целочисленный тип данных, а после указания имени поля через двоеточие должен быть указан размер этого поля в битах. Приведем пример:

```
struct bit //имя типа struct bit
{
      int b1:2;
      int b2:3;
      int b3:1;
      float b;
}b; //переменная b типа struct bit
```

В этой структуре сначала размещается поле с именем *b1*, занимая *2* бита, затем поле с именем *b2* (*3* бита), потом поле с именем *b3* (*1* бит), и, наконец, компонент с именем *b* типа *float*. Так как суммарно первые три поля занимают *6* бит, а компонент *b* должен размещаться в ячейке, адрес которой кратен байту или машинному слову (в разных машинах по разному), то между полем *b3* и компонентом *b* будет дырка в структуре, никак не используемая.

Возможность использования битовых полей - еще один пример слабой типизации языка Си, которая будет описана ниже в

в теоретической части одной из лабораторных работ.

Выборка полей структуры производится так же, как и выборка компонент структуры с помощью операции ".", например:

b.b1;

Учтите, что выборка поля из структуры транслируется не в одну машинную команду, а в несколько команд, т.е. при выборке/записи полей эффективность падает по сравнению с выборкой/записью компонент структуры.

2.1.3.2. Объединение (реализует операцию альтернативы)

Конструктор данных *объединение* реализует операцию альтернативы. Как и структура, объединение содержит описания компонент вообще говоря разных типов. Однако в отличие от структуры, в объединении в каждый момент времени существует только один компонент этого объединения. Приведем пример:

```
union u // имя типа union u
{
      int    a;
      float  b;
      char  *p;
};
```

Этот конструктор данных содержит три компонента; В языке Паскаль имеется аналогичный конструктор данных: *record* с *case*. Приведем пример:

```
Type
rec=record
case t:0..2 of
      0: (a:integer);
      1: (b:real);
      2: (p:^char)
end;
var v:rec;
```

Напомним, что язык Паскаль - строго типизированный язык, поэтому объединение трех его компонент находится под управлением *тега t*: при обращении к любому из компонентов объединения производится проверка, что *тег t* имеет соответствующее значение (в противном случае выдается ошибка). Если, например, выполняется обращение *v.p*, то тег *v.t* должен иметь значение *2*.

В объединении языка Си такого тега нет. Поэтому предполагается, что при получении переменной типа объединение программист либо знает существующий вариант объединения, либо этот вариант каким-либо образом динамически ему указывается (например, с помощью значения дополнительной переменной).

Таким образом, язык Си предоставляет программисту еще один инструмент для управления размещением компонент в памяти, правда, слаботипизированный. В приведенном выше объединении *union u* компонент с именем a занимает *2* байта, компонент с именем *b* - *4* байта, а компонент с именем *p* - *4* байта. Тогда переменная этого типа будет занимать *4* байта: размер наибольшего компонента.

Этот тип имеет имя *union u*, где *u* - это *ярлык типа* (а не имя типа). Далее в программе - после определения этого типа - можно было бы объявить переменную этого типа и указатель на этот тип, а затем выбрать компоненты объединения:

```
union u u,    //переменная u, имеющая тип union u
  *pu;        //переменная —указатель на union u
u.a;//операция, используемая для выборки компонента a
pu->a;
//операция ->, используемая для выборки компонента a
……..
```

Вообще говоря, для выборки компонента объединения через указатель надо использовать следующую операцию:

*(*pu).a;*

но для упрощения записи в языке Си введена операция ->.

Чтобы упростить имена типов структур и объединений, в языке Си используется описатель *typedef* для задания синонима типа. Приведем пример:

typedef struct s ts, union u tu;

В данном примере для типа *struct s* введен синоним *ts*, а для

типа *union u* - синоним *tu*. Эти синонимы можно использовать при объявлении переменных:

```
ts v; //эквивалентно struct s v;
tu vl;//эквивалентно union u vl;
```

2.1.3.3. Массив (реализует операцию цикла)

Конструктор данных языка Си *массив* используется для объединения в одно целое нескольких компонент одинакового типа. Тем самым с помощью массива реализуется операция цикла: *n* - кратное размещение одного и того же компонента. Приведем пример:

```
int m[100];//массив m из 100 целых чисел
```

Выборка компонента массива производится с помощью индексации:

```
m[i];
```

При объявлении массива указывается только верхняя граница диапазона его индексации (точнее говоря, число элементов массива). Нижняя граница массива всегда равна 0.

2.1.4. Логическая интерпретация в языке Си

В отличие от языков высокого уровня, в языке Си нет логического типа. В логических операциях (*!*, *&&* и ||), операциях отношения, а также логических выражениях структурных операторов используется *логическая интерпретация* операндов целочисленных и указательных типов: если операнд не равен *0* (*NULL*), соответственно, то его логическая интерпретация равна *TRUE*, а в противном случае *FALSE*.

Отсутствие логического типа в языке Си является еще одним примером его слабой типизации. Действительно, типы языка Си тесно связаны с машинными типами объектной машины, а среди машинных типов нет чистого логического типа. Обычно в машине при вычислении операнда (или его считывании на регистр) неявно формируются признаки этого операнда (например, в PSW на IBM PC), которые затем неявно используются в командах условных переходов. Поэтому отсутствие логического типа

способствует оптимизации программы. Приведем пример:

```
int a = 3;
while (a)
      a --;
```

Этот оператор *while* выполняется тогда, когда переменная с именем a не равна 0: в этом случае логическая интерпретация выражения равна *TRUE*. Поэтому этот цикл будет выполнен 3 раза, пока переменная *a* не станет равной 0.

Рассмотрим еще один пример:

```
int a = 3
a = a==3;
```

В этом примере сначала выполняется операция отношения ==. Так как ее логическая интерпретация равна *TRUE (a == 3)*, то результат этой операции будет типа *int* и иметь для определенности значение 1 (в противном случае результат этой операции был бы равен 0). Поэтому в переменную с именем *a* будет записано значение 1.

Так как в языке Си нет логического типа, то результаты логических операций имеют тип *int* и равны 1 (0), если результат операции соответствует *TRUE* (*FALSE*) Тоже самое справедливо для операций отношения.

2.1.5. Немного о регулярных выражениях

Регулярные выражения в той или иной форме используются всеми основными текстовыми редакторами и утилитами, выполняющими фильтрацию текста в Shell ОС UNIX.

Ниже перечислены метасимволы и операторы, применяемые в базовых регулярных выражениях:

^ Соответствует началу строки

$ Соответствует концу строки

[ ] Соответствует любому символу из числа заключенных в квадратные скобки; чтобы задать диапазон символов, укажите первый символ диапазона, дефис и последний символ (например, вместо шаблона [12345] можно ввести [1-5])

[^] Соответствует любому символу, кроме тех, которые

указаны в квадратных скобках

\ Отменяет специальное значение следующего за ним символа

. Соответствует любому указанному символу

* Указывает на то, что предыдущий шаблон встречается нуль или более раз; в программах awk или egrep, где используются расширенные регулярные выражения, существует две дополнительных операции: ? (означает, что предыдущий шаблон встречает не более одного раза) и + (означает, что предыдущий шаблон встречается один или более раз)

\{n\} Указывает на то, что предыдущий шаблон встречается ровно n раз

\{n,\} Указывает на то, что предыдущий шаблон встречается не менее n раз

\{,m\} Указывает на то, что предыдущий шаблон встречается не более n раз

\{n,m\} Указывает на то, что предыдущий шаблон встречается не менее n раз и не более m раз.

2.2. Условия и общий порядок выполнения работы

См. раздел 1.2.

2.3. Примеры заданий

1. Написать подпрограмму сортировки одномерного массива.

2. Написать подпрограмму сортировки многомерного массива.

3. Написать подпрограмму слияния двух файлов и сохранения результата в третьем файле.

4. Написать подпрограмму сортировки строк файла в лексикографическом порядке.

5. Написать подпрограмму подсчета числа строк, слов и символов в файле.

6. Предварительно ознакомьтесь с регулярными выражениями UNIX (см. раздел 2.1.5). Написать подпрограмму поиска, которую выполняет команда grep в файле с именем file:

6.1. grep [^0-9\$] file;
6.2. grep ^\.[0-9][0-9] file;
6.3. grep {2\}[0-9]\{4\}[0-9]\{3\}\.[0-9]\{3\}\ file;

6.4. grep [0-9]\{2\}-[0-9]\ file;
6.5. grep De[Vv]ice\. file;
6.6. grep ^.$ file;
6.7. grep \^q file;
6.8. grep De..ce file;
6.9. grep [Dd]evice file;
6.10. grep [123] file;
6.11. grep [^0-9A-Za-z] file;
6.12.. grep ^……$ file;
6.13. grep [iInN] file;
6.14. grep 000* file;
6.15. grep ^d..x..x..x file;
6.16. grep [Ss]igna[Ll] file;
6.17. grep [Ss]igna[lL]\. file;
6.18. grep \. file;
6.19. grep ^[^l] file;
6.20. grep [iI] file;
6.21. grep [0-9]\{2\}-[0-9]\ file;
6.22. grep '5[[:upper:]][[:upper:]]' file.

7. Удалите из файла повторяющиеся строки.

8. Пусть имеются два файла, в каждом из которых информация разбита по полям. Выполните соединение строк из этих файлов по совпадению указанных полей и поместите результат в указанный файл.

9. Пусть имеется файл, в котором информация разбита по пяти полям. Вырежьте 2 и 5 столбцы и поместите результат в указанный файл.

10. Пусть имеется файл, в котором информация разбита по пяти полям. Произведите циклический сдвиг полей: 1 2 3 4 5 -> 5 4 3 2 1.

3. Лабораторная работа "ИСПОЛЬЗОВАНИЕ СЛАБОЙ ТИПИЗАЦИИ ЯЗЫКА СИ" (1 занятие)

3.1. Основные теоретические положения

В данном разделе будет показано, почему язык Си относится к языкам среднего уровня. Возможности этого языка будут иллюстрироваться в сравнении с возможностями известного студентам языка программирования Паскаль.

3.1.1. Назначение типизации в языках программирования

Впервые типы данных были введены Н.Виртом в языке Паскаль. Это было сделано для повышения надежности создаваемых программ и обеспечения АВТОМАТИЧЕСКОГО перехвата основных ошибок в программе на этапе ее компиляции или на этапе ее выполнения. Другими словами, если программист задает тип данных для переменной или другого объекта программы, то тем самым он накладывает на этот объект ограничения, соблюдение которых обеспечивает язык программирования.

Определение типа данных: *тип данных* задает:

множество допустимых значений для объекта этого типа,

множество допустимых операций, разрешенных над объектом этого типа.

Приведем пример на языке Паскаль:

```
program P (input,output);
……..
type range:0..1; (* тип диапазона INTEGER *)

var v: range; (* переменная типа range *)
m: array [range] of integer ;(* массив *)
begin
……..
v:=12; (* ошибка - переменной v присваивается недопустимое значение *)
……..
v := v.s; (* ошибка - недопустимая операция v.s над переменной v *)
……..
m[12] := 1; (* ошибка - выход индексации массива за границы диапазона этого массива *)
……..
```



```
end. (* конец программы *)
```

Язык программирования, в котором поддерживаются типы данных, называется *типизированным*. Если язык всегда поддерживает типизацию согласно приведенному выше определению типа данных, то он называется *строго типизированным*.

Как было сказано выше, проверки нарушения типизации производятся автоматически либо на этапе компиляции программы, либо на этапе ее выполнения. Проверки нарушения типизации на этапе выполнения вызывают накладные расходы, так как в компилируемую программу компилятор неявно вставляет машинные команды, производящие эти проверки. Из-за этого увеличивается объем программы и время ее выполнения.

Так, например, для каждого обращения к элементу массива *m[i]* в программе требуется проверять, что значение индексной переменной *i* принадлежит диапазону индекса массива. При присваивании значения переменной также требуется проверять, что присваиваемое значение принадлежит множеству допустимых значений этой переменной.

Иногда опытные программисты поступают так: после отладки программы отключают все или часть проверок нарушения типизации, что повысить эффективность программы. Но это будет отход от строгой типизации языка, и он чреват последствиями.

3.1.2. Слабая типизация в языке Си

В языке Си также используется механизм типизации данных. В этом языке имеются и предопределенные типы данных, и конструкторы типов данных для создания новых типов. Это обеспечивает достаточную надежность работы программ, написанных на языке Си, по сравнению с работой программ, созданных на языке ассемблера, тем самым облегчая процесс программирования.

Так, например, ассемблер при объявлении переменной просто выделяет для нее ячейку памяти и при этом не контролирует, что будет записано в эту ячейку. Язык Си обеспечивает контроль записи в свою переменную осмысленного значения, правда, как будет показано ниже, в более широком смысле по сравнению с языком строгой типизации.

Для *повышения эффективности программ* в языке Си допускается отход от строгой типизации в тех - предопределенных языком Си - случаях, когда это может привести к существенному повышению эффективности или гибкости программы. Далее эти случаи описываются более подробно.

3.1.2.1. Отсутствие проверок выхода индекса за границу диапазона индексации массива

В языке Си нет полной автоматической проверки правильности выполнения операции индексации массива (*m[i]*) т.е. проверки того, что значение индексной переменной *i* принадлежит диапазону индекса массива.

Это вовсе не означает, что в языке Си программист может с помощью этой операции обращаться к несуществующим элементам массива - просто компилятор возлагает ответственность за правильность использования этой операции на самого программиста. Единственное, что проверяет компилятор, это операции типа *m [константное_выражение]* на этапе компиляции.

Благодаря этому повышается эффективность программ, так как в них неявно не вставляются команды проверки выхода индекса за границы диапазона индексации массива. При необходимости сам программист может вставить в программу проверки индексации.

Приведем пример, иллюстрирующий способ правильной вставки таких проверок:

```
#define TEST//определение макропеременной TEST
...int m[ 100];//массив 100 целых чисел
……..
#ifdef TEST//макрооператор условной макрогенерации
if(i<0||i>=100)
{printf("ошибка в индексе m[%d] \n ", i);exit(-1);}
#endif
……..
i=m[i];
```

В этом примере на этапе отладки программы - т.е. пока определена макропеременная *TEST* - будет производиться проверка правильности обращения к элементу массива *m[i]*. После того, как отладка программы будет закончена, можно убрать определение макропеременной *TEST* (например, закомментировав его или отменив его с помощью последующего микрооператора *#undef TEST*), что отменить включение в выполняемую программу команд проверки правильности обращения *m[i]*.

Кроме того, благодаря отсутствию этих проверок, в языке Си массив можно трактовать как ДИНАМИЧЕСКИЙ. Так, например, в этом языке подпрограмме нельзя передать массив целиком, но можно передать указатель на 0-ой элемент этого массива. Тогда подпрограмма может определить конец этого массива по содержимому его конкретного элемента (это используется при обработке строк, в которых признаком конца является байт, все биты которого равны *0* - *NULL-байт*). Этой подпрограмме можно также передать на обработку указатель на *i*-й элемент этого массива, что обеспечивает возможность обработки подмассивов. И, наконец, как мы увидим далее, массив можно обрабатывать с помощью указателей.

Таким образом, отсутствие проверки индексации повышает эффективность программ и гибкость интерпретации массива, но снижает надежность программ, так как все проверки возлагаются на программиста.

3.1.2.2. Введение явных операций преобразования типов данных

В языке Си можно указывать операции преобразования операнда одного типа к другому типу, например:

```
int * p;//указатель
……..
p = (int *) 2;//операция преобразования типа
```

В этом примере операнд типа *int* (число *2*) преобразуется к указательному типу (*int* *). Эта операция может потребоваться тогда, когда в конкретной машине точно известно, что указатель должен иметь целочисленное значение *2*.

Эта операция обычно не производит никаких преобразова-

ний над значением *2*, а просто изменяет трактовку типа этого значения. Компилятор выдает предупреждение, что эта операция преобразования типов подозрительна и выполнима не на всех компьютерах.

Операции преобразования типов повышают эффективность и гибкость программ, приближая их к ассемблеровским программам, но снижают надежность самих программ, а также их мобильность.

3.1.2.3. Все предопределенные типы языка Си отображаются в соответствующие машинные типы объектной машины

3.1.2.3.1. Представление о форматах машинных типов

Сначала введем несколько определений. Инструментальной машиной называется компьютер, на котором работает компилятор. Объектной машиной называется компьютер, для работы на котором компилируется программа на инструментальной машине. Обычно инструментальная и объектная машины совпадают.

В каждом компьютере существует основной формат операндов, который называется *машинным словом*. Размер машинного слова составляет *n* разрядов (обычно *n* кратно *8*, чтобы в машинном слове помещалось целое число байт). Например, на компьютере CRAY-1 *машинное слово* занимает *64* разряда, а на PC XT - *16* разрядов. Обычно чем больше мощность компьютера, тем больше размер машинного слова. Кроме того, оперативная память разбивается на машинные слова, в которых можно хранить адресуемые по памяти операнды.

Машинным типом называется формат машинных операндов, для которых в конкретной машине определен конкретный набор команд (например, целое число со знаком, вещественное число с плавающей точкой и т.д.). Архитектура компьютера проектируется таким образом, чтобы все основные команды могли обрабатывать операнд соответствующего машинного типа, хранимый в машинном слове. Кроме того, в компьютерах допускаются промежуточные форматы машинных операндов: *полуслова, двойные слова, байты* и т.д. Для этих операндов в компьютере обычно существует неполный набор команд, поэтому часть операций над ними реализуется с помощью подпрограмм.

В отличие от языка Паскаль, в языке Си предполагается, что его предопределенный тип ПРЯМО отображается в соответствующий машинный тип объектной машины. Это означает, что над операндом предопределенного типа можно выполнять не только операции, вытекающие из свойств его типа, но также и многие операции, которые поддерживаются соответствующими машинными командами. Так, например, над целочисленным операндом можно не только выполнять операции +, -, * и / (как в языке Паскаль), но и операции сдвига и побитовые логические операции. Перечислим основные машинные типы большинства известных компьютеров:

1. целое число со знаком;
2. целое число без знака;
3. логический код;
4. вещественное число с плавающей точкой;
5. адрес;
6. строка символов.

3.1.2.3.2. Машинный тип "целое число со знаком"

Пусть машинное слово состоит из *n* разрядов. Значение целого числа со знаком занимает (*n-1*) правых разрядов машинного слова, а в левом *n*-м разряде хранится знак числа: 0 (1) для положительных (отрицательных) чисел. Положительные числа хранятся в прямом двоичном коде (например, десятичное число *+5* представляется в байтовом машинном слове как двоичное число *00000101*). Отрицательные числа хранятся в дополнительном двоичном коде.

Дополнительный двоичный код получается из прямого *двоичного кода* следующим образом: сначала прямой код инвертируется (например, *00000101 -> 11111010*), а затем к полученному *обратному коду* прибавляется 1 (*11111010 + 1 -> 11111011*). Так мы получили дополнительный двоичный код для десятичного числа -5. Заметим, что знаковый разряд отрицательного числа генерируется автоматически при получении дополнительного кода. Также отметим, что сумма кодов чисел (*+5*) + (*-5*) дает двоичный

код *100000000*.

Диапазон представления значений целого числа со знаком: от $-2^{(n-1)}$ до $2^{(n-1)}-1$. Для операндов типа целое число со знаком в машине обычно имеются все команды: сложение, вычитание, умножение, деление и их модификации.

3.1.2.3.3. Машинный тип "целое число без знака"

Значение целого числа без знака занимает все машинное слово. Таким образом, диапазон представления значений целого числа без знака: от *0* до $+2^n-1$. Для операндов этого машинного типа обычно в машине имеется неполный набор арифметических команд.

3.1.2.3.4. Машинный тип "логический код"

Значение *логического кода* занимает все машинное слово. В этом случае машинное слово рассматривается как упорядоченная последовательность логических разрядов, начиная с *0*-го самого правого разряда и заканчивая (*n-1*)-м самым левым разрядом.

Все логические операции над значением логического кода выполняются поразрядно и независимо от значений других разрядов. Если в логической операции участвуют два логических кода, то операция выполняется для разрядов, имеющих одинаковые номера в обоих кодах.

В компьютере обычно имеются следующие логические команды над логическим кодом, хранимым в машинном слове:

инвертирование (~) (все разряды машинного слова меняют свое значение на обратное; например, *~01010101 -> 10101010*);

логическое побитовое умножение (&) (например:

0011&0101 -> 0001);

логическое побитовое сравнение (^) (например:

0011^0101 -> 0110);

логическое побитовое сложение (|) (например:

0011|0101-> 0111};

операции сдвига (влево/вправо; логически, арифметически, циклически).

Приведем примеры:


```
01110101        01110101        01110101
&               ^               |
10001111        10001111        11111101
------------    ------------    ------------
00000101        11111010        11111101
```

Отметим, что логическое сравнение (сравнение по модулю *2*) используется для логического (не путать с арифметическим) сравнения двух операндов.

Операции сдвига бывают трех типов:

логический (значение сдвигается влево/вправо на указанное число разрядов; выдвигаемые из машинного слова разряды исчезают, а с другой стороны в машинное слово вдвигаются нули);

арифметический (сдвиг влево (вправо) на *n* разрядов эквивалентен умножению (делению) числа на 2^n; поэтому при сдвиге вправо для сохранения знака и значения числа производится расширение знака вместо вставки нулей);

циклический (значение сдвигается влево/вправо на указанное число разрядов; выдвигаемые из машинного слова разряды вдвигаются с другой стороны в это же машинное слово).

Приведем примеры:

01010101>> 2 (логически) -> 00010101
10101010>> 2 (арифм.) -> 11101010 (расширение знака)
00001111>> 4 (циклически) -> 11110000

В языке Си не поддерживается только циклический сдвиг (но его можно реализовать с помощью подпрограмм).

3.1.2.3.5. Машинный тип "вещественное число с плавающей точкой"

Теперь исследуем вопрос, как можно хранить операнд машинного типа *вещественное число* в машинном слове. Под вещественным числом понимается такое число, которое содержит целую и дробную части, например, *2.7*.

На первый взгляд, кажется, что для хранения вещественного числа удобнее всего использовать фиксированный формат: для этого разделим машинное слово на две равные половины, в одной

из которых будем хранить целую часть числа, а в другой - его дробную часть. Однако сразу же виден недостаток этого формата: он неудобен для хранения чисел, у которых слишком длинная целая часть и короткая дробная часть либо наоборот.

Теперь рассмотрим пример, когда двоичные числа представляются в *экспоненциальной форме* (т.е. такой форме, когда значение числа представляется в виде произведения самого числа на значение базы системы счисления этого числа, предварительно возведенной в некоторую целую степень; эта степень называется порядком):

$$1.1*2^0 = 0.011*2^2 = 110.0*2^{-2}$$

Как видно из этого примера, значение числа в экспоненциальной форме не изменится, если десятичную точку сдвинуть в числе влево (вправо) на *n* разрядов и одновременно прибавить к порядку (вычесть из него) это число *n*. Такая форма представления вещественного числа называется *вещественное число с плавающей точкой*.

Для хранения вещественного числа с плавающей точкой машинное слово делится на две неравные части: в правой большей части хранится значение (или мантисса) числа, а в левой части - порядок этого числа. И мантисса, и порядок хранятся как целые числа со знаком.

Для экономного хранения мантиссы лучше всего хранить число в *нормализованном виде*, когда самый значащий разряд мантиссы занимает самый левый - перед знаком мантиссы - разряд мантиссы. В этом случае в мантиссе может поместиться максимальное число ее значащих разрядов.

К сожалению, в формате вещественного числа с плавающей точкой не удастся сохранять все вещественные числа, поскольку ограничена по размеру и мантисса, и порядок (так, например, в этом формате не удастся сохранить число *e*). Диапазон представления вещественных чисел определяется размером порядка, а точность (число разрядов числа) - размером мантиссы.

Для увеличения точности часто используют формат удвоенной точности, когда следующее машинное слово используется

для хранения дополнительных младших разрядов мантиссы.

3.1.2.3.6. Машинный тип "адрес"

Машинный тип *адрес* содержит адрес операнда в оперативной памяти. Этому машинному типу соответствует указатель языка Си.

3.1.2.3.7. Машинный тип "строка символов"

Строка символов имеет фиксированный формат в языке Си: в первых байтах хранятся коды символов, а признаком конца строки является байт, все биты которого равны *0* (*NULL-байт*). Например, строка *"мама"* занимает пять байтов, в первых из которых последовательно хранятся коды символов *'м'*, *'а'*, *'м'*, *'а'*, а последний байт является нулевым.

Для кодирования русских букв используется расширенная кодировка *ASCII*. Левую половину этой кодировки (первые *128* кодов) занимают системные непечатаемые символы, используемые для управления ВЗУ и устройствами ввода/вывода (первые *32* байта), затем коды латинских букв, цифр и знаков препинания. В правой половине этой кодировки (коды от *128* до *255*) хранятся коды русских букв.

В настоящее время используются следующие кодировки:
ISO 8859-5 (для UNIX);
CP 1251 (для Windows);
CP 866 (для MS-DOS);
KOI-8 (для E-mail).

3.1.2.3.8. Дополнительные замечания

Обычно предопределенные типы языка Си хранятся так: основные типы (*int*, *unsigned int*, *float*, адрес) в машинном слове, (*long*, *unsigned long*, *double*) в удвоенном машинном слове, (*short*, *unsigned short*) в полуслове, (*signed char*, *char*) в байте. Если же в компьютере нет или недостаточно дополнительных машинных форматов (помимо машинного слова, которое всегда имеется), то язык Си предоставляет разработчику компилятора самому выбрать отображение предопределенных типов языка Си в машинные типы.

Для логического кода не вводится специальный тип в языке Си. Вместо этого предполагается, что логическому коду в языке Си соответствует целое число со знаком или без знака, поскольку оно хранится в двоичном представлении.

Другие аспекты слабой типизации языка Си будут обсуждаться по мере подготовки других лабораторных работ. Некоторые из них уже были отмечены в теоретической части лабораторной работы 2.

3.2. Условия и общий порядок выполнения работы

См. раздел 1.2.

3.3. Примеры заданий

1.Создать базу данных “студенты” в виде файла, содержащего имя и фамилию студента, а также его оценку. Уметь: 1) вводить данные; 2)удалять данные; 3)изменять данные; 4) сортировать данные по любому столбцу; 5) выводить данные на экран.

2. Создать базу данных “служащие” в виде файла, содержащего имя, фамилию и отчество служащего, а также его рейтинг. Уметь: 1) вводить данные; 2)удалять данные; 3)изменять данные; 4) сортировать данные по любому столбцу или их комбинации; 5) выводить данные на экран.

3. Создать подпрограмму исключения комментариев из исходного текста, составленного на языке Си. Программа должна выполняться в режиме фильтра, который получает исходный текст с комментариями из стандартного ввода (stdin) и отображает тот же текст без комментариев через стандартный вывод (stdout).

4. Создать подпрограмму для определения частоты использования латинских букв в данном текстовом файле. При выполнении измерений надо игнорировать регистр букв. Результаты измерений должны быть отображены через стандартный вывод ошибок (stderr). При этом в каждой строке вывода надо перечислять буквы, которые встречаются в данном файле с равной частотой. Строки вывода надо упорядочивать по убыванию частоты букв.

5. Создать подпрограмму для визуальной индикации выполнения длительных процессов путем циклического чередования в начальной позиции стандартного вывода (stdout) отображений

символов '|', '/', '-', '\', '|', '/', '-', '\'. Чередование указанных символов должно продолжаться, пока не исчерпан заданный в секундах временной интервал. Величина временного интервала и скорость чередования этих символов передаются в программу через аргументы командной строки.

6. Написать подпрограмму перевода выражения из обычной формы в польскую обратную запись (a в +).

7. Написать подпрограмму перевода выражения из польской обратной записи (a в +) в обычную форму.

8. Написать подпрограмму перевода выражения из обычной формы в префиксную запись (+ a в).

9. Написать подпрограмму перевода выражения из префиксной записи (+ a в) в обычную форму.

10. Написать подпрограмму перевода выражения из префиксной записи (+ a в) в польскую обратную запись (a b +).

11. Написать подпрограмму перевода выражения из польской обратной записи (a в +) в префиксную запись (+ a b).

12. Написать подпрограмму вычисления константного выражения (использовать операции +, -, * и /).

13. Написать подпрограмму сохранения имен в таблице имен. Использовать для убыстрения поиска хэширование имен, причем хэш-код равен остатку от деления суммы символов имени на число 211. Таблицу имен представлять в виде массива структур: :**struct** s { char NAME[30]; /* имя */ struct s * collision_link; /* ссылка на предыдущее имя, имеющее тот же хэш-код */ struct s * previous_link; /* ссылка на предыдущее имя, ранее объявленное в программе */ }. При попытке записи одинаковых имен фиксировать ошибку.

14. Создать подпрограмму, измеряющую длины всех файлов из заданного списка, фиксируя первые десятичные цифры полученных величин для анализа частоты их повторений. Список имен файлов должен передаваться программе через строки стандартного ввода (stdin), где имя каждого файла указывается в отдельной строке. Число повторений каждой цифры должно быть отображено через стандартный вывод ошибок (stderr).

15. Написать подпрограмму, реализующую комплексное

число в виде структуры: **struct** complex {double R; /* реальная часть */ double I; /* мнимая часть */ }. Реализовать функции инициализации, сложения, вычитания, умножения и деления комплексных чисел, а также обеспечить перевод числа в полярные координаты.

16. Написать подпрограмму, реализующую комплексное число в виде структуры: **struct** complex {double R; /* радиус */ double F; /* угол поворота в полярных координатах*/ }. Реализовать функции инициализации, сложения, вычитания, умножения и деления комплексных чисел, а также обеспечить перевод числа в декартовы координаты.

17. Создать подпрограмму для определения среднего размера строки в данном текстовом файле и количества непустых строк, длина которых не больше средней величины. Результаты измерений должны быть отображены через стандартный вывод ошибок (stderr). Имя файла указывается в командной строке.

18. Создать подпрограмму форматирования данного текста для разбиения его на страницы с фиксированным числом строк. Форматируемый текст может содержать произвольное число абзацев и однострочных заголовков, которые разделяют пустые строки, где нет других символов, кроме необязательных пробелов и табуляций. Процедура форматирования должна исключать разрыв абзаца и размещение заголовка в конце страницы. Дополнение неполной страницы должно обеспечиваться вставкой соответствующих пустых строк в конец страницы. Имя файла с текстом и нужный размер страницы передаются программе через аргументы командной строки. Результаты измерений должны быть отображены через стандартный вывод ошибок (stderr).

19. Создать подпрограмму для определения среднего размера слова в данном текстовом файле и количества различных слов, длина которых не больше средней величины. Программа должна анализировать только слова текста, состоящие из латинских букв. Результаты измерений должны быть отображены через стандартный вывод ошибок (stderr).

20. Написать подпрограмму, реализующую динамический массив (переменная верхняя граница) в виде структуры:

struct array {int * array; /* ссылка на массив */ int size; /* размер массива */ }. Реализовать функции создания, удаления, сложения, вычитания, умножения и деления элементов массива, а также обеспечить выборку элементов массива и контроль индекса.

21. Создать подпрограмму компоновки произвольного списка слов в заданное число колонок. При этом слова списка должны распределяться по колонкам в горизонтальном направлении. Содержимое каждой колонки должно быть выровнено по левой границе. Число слов в колонках не должно различаться больше, чем на единицу. Промежутки между колонками должны состоять из равного числа символов пробел. Программа должна получать список слов через строки стандартного ввода (stdin) и отображать результат его обработки в стандартный вывод (stdout). Требуемое число колонок для размещения слов должно передаваться программе через аргумент командной строки.

22. Создать подпрограмму для поиска абзаца текста, который состоит из наибольшего числа слов. Для разделения абзацев в тексте используются пустые строки, где нет других символов, кроме необязательных пробелов и табуляций. Результат поиска должен отображаться информационным сообщением через стандартный вывод ошибок (stderr). Оно должно указывать номера начальных строк всех абзацев текста, содержащих наибольшее число слов.

23. Написать подпрограмму, заполняющую поля следующего заголовка модуля: имя_модуля (16 байтов), номер_модуля (6 битов), пустое_поле (1 бит), 9 полей битовых полей: r, w, x; r, w, x; r, w, x (по три группы на владельца, группу и остальных); дата_последнего_изменения (6 байтов по два байта на номер месяца, номер дня и 2 последние цифры года); пустое_поле (1 бит); номер_версии (3 бита); бит готовности (1 бит); размер_модуля (31 бит). Используя побитовые логические операции, заполнить поля этого модуля произвольными данными. Считать, что заголовок модуля представляется как: int HEAD[100].

24. Исходные данные такие же, как и в предыдущей задаче, но только оформить их в виде структуры, указав соответствующие поля, например: **struct** head {char name[16];
int num_module:6; :1; ruser:1; wuser:1; xuser:1; и т.д.}. Заполнить

поля этой структуры произвольными данными.

25. Узнать для используемого PC формат числа с плавающей точкой (формат порядка, формат мантиссы и их знаков). Заполнить поле с именем **lng** в объединении таким образом, чтобы получилось число 2.7 в поле **flt**: **union** float {long lng; float flt;}. Используйте логические побитовые операции. Не забывайте, что в объединении все компоненты перекрываются. Не забывайте также о нормализации числа.

26. Узнать для используемого PC формат числа с плавающей точкой (формат порядка, формат мантиссы и их знаков). Присвоить полю **flt** в объединении значение 2.9; затем, зная формат числа с плавающей точкой и используя перекрытие с указанным ниже в объединении полем **lng,** получить целое число 2: **union** float {long lng; float flt;}. Используйте логические побитовые операции. Не забывайте также о нормализации числа.

27. Пусть имеются файлы, состоящие из текстовых столбцов, разделенных указанным преподавателем разделителем. Написать подпрограмму слияния двух файлов по одинаковым значениям указанных столбцов.

4. Лабораторная работа "СВЯЗЬ МЕЖДУ МАССИВАМИ И УКАЗАТЕЛЯМИ В ЯЗЫКЕ СИ"

(3 занятия)

4.1. Основные теоретические положения

4.1.1. Определение массива

Определим массив *m*, состоящий из десяти целочисленных значений:

int m[i];

Этот массив имеет диапазон изменения индекса от 0 до 9. Выборку элемента массива можно производить традиционно, используя конструкцию *m[i]*. Правда, при этом язык Си почти не контролирует выход индекса за границы его диапазона. Поэтому корректность обращения к массиву почти полностью возлагается на программиста.

Если программист ошибется и обратится к несуществующему элементу массива, он может считать или испортить содержимое других переменных.

4.1.2. Определение указателя

Определим два следующих указателя на базовый тип элемента приведенного выше массива (тип *int*):

*int *pm, *pm1;//переменные pm и pm1 типа int **

Выборка значения по указателю осуществляется с помощью следующей операции: **pm*.

4.1.3. Использование указателей для обращения к элементам одного и того же массива

Установим в указатель *pm* адрес 0-ого элемента массива *m*:

pm = &m[0];

В этом выражении унарная операция *&* обозначает взятие адреса переменной (в данном случае переменной *m[0]*). Таким образом, унарная операция & используется для получения значения указателя. В этом выражении полученное значение указателя записывается в переменную *pm*.

Обратной - по отношению к унарной операции & - является унарная операция * (взятие значения по указателю). Поэтому верно следующее тождество для любой переменной a:

a==&a*

В рассматриваемом примере выражение **pm* выдаст значение m[0]. Теперь прибавим к указателю *pm* целое значение 6 и запишем полученный результат в *pm*:

pm = pm + 6;

Теперь указатель *pm* указывает не на 0-ой элемент массива, а на 6-ой. Поэтому выполняется следующее равенство:

**pm = = m[6]*

Затем отнимем от указателя *pm* целое значение 3 и запишем полученный результат в *pm*:

pm = pm - 3;

Теперь указатель *pm* указывает не на 6-ой элемент массива, а на 3-ий. Поэтому выполняется следующее равенство:

**pm = = m[3]*

Таким образом, прибавление или вычитание целого значения (или целой переменной) из указателя на базовый элемент массива позволяет сканировать массив с помощью этого указателя аналогично тому, как это можно было бы сделать с помощью индекса массива. При этом контроль выхода за границы массива полностью возлагается на программиста. Значение элемента массива выбирается с помощью операции **pm*.

Теперь установим в указатель pt1 адрес 9-ого элемента массива:

pm1 = &m[9];

Тогда значение выражения:

pm1 - pm;

будет равно 6, т.е. числу элементов массива *m* между указателями *pm1* и *pm*, а значение выражения:

pm - pm1;

будет равно -6. Благодаря этой операции можно узнать не только число элементов массива между указателями, но также и то, какой из указателей указывает на элемент массива, расположенный ближе к базе массива (в данном примере ближе к базе массива элемент указателя *pm*).

Кроме того, можно применять полный набор операций отношения для указателей, используемых для обработки массива: ==, *!*=, >,>=,< и <=, используя следующее правило: тот указатель, который указывает на элемент массива, расположенный ближе к базе массива, считается меньшим. В данном примере выполняется отношение:

pm1 >pm

Сразу же отметим, что только что описанные операции применимы к указателям, которые используются для обработки *одного и того же* массива. Например, не имеет смысла операция сравнения двух указателей (пусть даже одного и того же типа), которые указывают на элементы разных массивов. Также бессмысленна операция *pm + целое_значение* или *pm - целое_значение*, если указатель *pm* не используется для обработки массива (т.е. указывает на какую-нибудь одиночную переменную типа *int*).

Если в выражении используется идентификатор массива, вслед за которым не указаны квадратные скобки, то он трактуется как указатель на 0-ой элемент этого массива. Так, например, выражение *m* имеет тип *int ** и эквивалентно выражению *&m[0]*. Также выполняется следующее тождество для любого идентификатора массива:

**m==m[0]*

4.1.4. Адресная арифметика языка Си

Пусть определен массив из некоторого числа элементов (например, *m* из предыдущего примера). Пусть также определены указатели на базовые элементы этого массива (например, *pm* и *pm1* из предыдущего примера), которые указывают на некоторые элементы этого массива. Пусть *i* - целочисленная переменная. Предположим также, что мы не выходим за границы массива. Тогда:

1) Всегда выполняется тождество:

*m[i] == * (m + i)*

так как *m* трактуется как указатель на 0-ой элемент массива.

2) Всегда можно выполнить операции:

pm + i

или

pm - i

с помощью которых можно сканировать массив.

3) Всегда можно выполнить операцию:

pm1 -pm;позволяющую определить число элементов массива между двумя указателями (с точностью до знака). Эта операция имеет тип *int*.

4) Всегда можно выполнить любую операцию отношения между двумя указателями:

pm1 > pm;

позволяющую определить, какой из указателей указывает на элемент массива, расположенный ближе к базе массива.

Набор этих операций вместе с унарными операциями *взятие адреса* (&) и *взятие значения* (*) образует адресную арифметику языка Си, которую можно эффективно использовать для обработки массивов.

4.1.5. Сравнение возможностей адресной арифметики языка Си с соответствующими возможностями языка Паскаль

Если сравнить возможности адресной арифметики языка Си с соответствующими возможностями языка Паскаль, то окажется, что язык Паскаль не имеет полной адресной арифметики. В первую очередь это объясняется тем, что при обработке массива с помощью указателей отсутствует контроль выхода за границы массива, что является нарушением строгой типизации. Поэтому в языке Паскаль указатели не используются для обработки массива так, как они используются в языке Си.

В языке Паскаль указатели используются только для получения динамической памяти с помощью механизма куча (операции *new()* и *dispose()*), поэтому их можно сравнивать только с помощью следующих операций: "=" и "<>".

Более того, в языке Паскаль вообще отсутствует операция взятия адреса переменной. Дело в том, что эта операция может приводить к нарушениям работы программы, которые очень трудно контролировать. Приведем пример на языке Си, приводящий к ошибке:

```
int *p;//глобальный указатель
void pr (void)//процедура pr
{int a = 2;//локальная переменная
p = &a;//присваивание глобальному указателю
//адреса локальной переменной
}//end pr
```



```
main()//стартовая функция
{
pr();
printf("*p = %d;\n", *p);•
//печать значения, выбранного по указателю p
}//end main
```

В этом примере в функции *main* сначала производится вызов процедуры *pr()*, которая устанавливает в глобальный указатель *p* адрес локальной переменной *a*, а затем выполняется попытка распечатать значение по этому указателю (т.е. содержимое переменной *a*). Последняя операция недопустима, так как объект *a* уже не существует после вызова процедуры *pr()*. Это некорректное действие очень трудно перехватить.

4.1.6. Примеры обработки массивов с помощью указателей

4.1.6.1. Поиск первого вхождения символа в строке

Приведем пример функции, которая ищет первое вхождение символа в строке и выдает номер его позиции в этой строке (или выдает -1, если в строке нет указанного символа).

В этой функции строка рассматривается как массив символов без явного указания размерности этого массива: вместо этого используется следующий факт - последним символом в строке является однобайтный символ, все биты которого равны 0 (*NULL-байт*). Поэтому с помощью этой функции можно обрабатывать не только массивы символов, но и их подмассивы.

В качестве параметра функции передается указатель на 0-й элемент обрабатываемого массива (или подмассива).

```
//определение функции n_char
static int n_char(char * s, //входная строка символов
                  char c//искомый символ)
{
int i = 0;//индекс массива
while (s[i] != 0) //пока очередной символ != 0
{if(s[i] == c) return i; //символ найден
i = i + 1; //увеличение индекса массива
```

```
}
return -1; //символ не найден
}//end n_char


main()//стартовая функция main
{int n;
char m[] = {'p', 'a', 'm', 'a', '\000'};
//явная инициализация массива отдельными
//символами с учетом NULL-байта
char m1[] = "рама";
//неявная инициализация массива
//строкой символов, уже содержащей NULL-байт
char *pm = "рама";
//инициализация указателя на строку символов
char * pm1;//указатель не проинициализирован
n = n_char(m, 'a');//результат n=1
n = n_char(m1, 'a');//результат n=1
n = n_char(pm, 'a');//результат n=1
n = n_char(pm1, 'a');
//может быть получен любой результат или скорее
//всего будет сбой, т.к. указатель
//не инициализирован
n = n_char(m+1, 'a'); //результат n=0
//передается подмассив, начиная с 1-го элемента
}//end main
```

Теперь перепишем функцию n_char() так, чтобы обработку массива вести только с помощью указателей:

```
//определение функции n_char
static int n_char(char * s, //входная строка символов
                  char c) //искомый символ)
{int * sbeg = s; //запоминание адреса начала строки
       while (*s != 0) //пока очередной символ != 0
{if (*s == c) return (s - sbeg); //символ найден
                                 //(s - sbeg) равно N позиции
s=s+1;//увеличение указателя на 1
```



```
}
return - 1;//символ не найден
} // end n_char
```

В этой реализации функции *n_char* просмотр массива строк производится с помощью указателя *s*.

4.1.6.2. Обработка многомерного массива

Приведем пример программы, используемой для обработки многомерного массива:

```
static int a[3] [3] = //матрица    3 X 3
{//инициализирована 3 строками
       {1,2,3},
       {4,5,6},
       {7,8,9}
 };
//эта матрица состоит из целых чисел
main(){
int i;
//см., что будет распечатано в цикле for
for(i=0;i<=2;i++)
  {
      printf("%d", a[i][2- i]);       //диагональ = 3,5,7
      printf("%d", *(a[2- i]+ i));  // диагональ = 3,5,7
      printf("%d", *a[i]);           //0-ой столбец = 1, 4, 7
      printf("%d", a[i][0]);         //0-ой столбец = 1,4,7
      printf("%d", a[i][i]);         //диагональ = 1, 5, 9
      printf("%d", *(*(a+ i)));    //диагональ = 1,5,9
  }//end for
}//end main
```

4.1.7. Способы передачи параметров функциям языка Си

Как известно, в языках программирования можно передавать параметры подпрограмме следующими способами:

по значению (когда передается значение параметра);

по ссылке (когда НЕЯВНО передается адрес параметра, а не его значение),

по имени (когда НЕЯВНО передается адрес начала выполнения подпрограммы, а также ее контекст, если она вложена в другие подпрограммы).

В языке Си параметры передаются ТОЛЬКО по значению.

Попытаемся создать функцию-процедуру на языке Си, которая обменивает значения двух переменных: a <-> b. Для того чтобы определить в языке Си функцию-процедуру, нужно указать тип результата этой функции как *void*.

```
static void exchange (int a, int b)
{ int  t;//временная переменная
t = a;//обмен значениями
a = b;
b = t;
}//end exchange
main()
{int x, y;
exchange (x,y);//попытка x<->y
}//end main
```

Попытка обменять значения переменных *x<->y* не удастся при вызове функции-процедуры *exchange*, так как этой функции будут переданы только значения этих переменных, которые запишутся в программный стек, затем в этом стеке произойдет обмен полученными значениями, а при выходе из функции exchange эти значения исчезнут в стеке.

Чтобы обменять значения самих переменных *x* и *y*, требуется передавать их адреса, а в качестве формальных параметров использовать указатели, как показано в следующем примере:

```
static void exchange (int * a, int * b)
{
int t;//временная переменная
t = *a;//обмен значениями
*a=*b;
*b = t;
}//end exchange
 main()
```



```
{int x, y;
exchange (&x,&y);//попытка x<->y
 }//end main
```

В этом случае попытка обменять значения переменных *x<->y* удастся при вызове функции-процедуры *exchange*, так как этой функции будут ЯВНО переданы адреса этих переменных, которые запишутся в программный стек, затем в этом стеке эти адреса будут ЯВНО использоваться для обмена значений тех переменных, на которые эти адреса указывают.

На языке Паскаль эта процедура могла бы быть реализована более изящно:

```
program P (input, output);
var x,y: integer;
procedure exchange (var a: integer; var b:integer);
var t: integer
begin
      t:=a;
      a:=b;
      b:=t;
end;(*end exchange*)
begin
      exchange(x,y);
end.//end program
```

Как видно из этого примера, достаточно добавить спецификатор *var* в описание формального параметра, как процедуре *exchange* будет НЕЯВНО передаваться значение фактического параметра по ссылке. При этом больше ничего менять в программе не требуется.

Таким образом, язык Си требует, чтобы при передаче адресов фактических параметров соответствующие формальные параметры имели ЯВНЫЙ указательный тип. Это обеспечивает наглядность использования адресации.

4.1.8. Указатель на функцию языка Си

Пусть требуется написать программу вычисления опреде-

ленного интеграла для заданной функции и заданного интервала значений ее аргумента. Конечно, если используется конкретная функция, эта задача не представляет особой сложности: для этого можно воспользоваться методом, например, прямоугольников, разбив указанный интервал аргумента на маленькие интервалы значений.

Чтобы решить эту задачу для любой функции, требуется ввести *указатель на функцию*. Указатель на функцию - это, по существу, адрес стартовой точки функции плюс спецификация ее формальных параметров и результата. Поэтому если определена функция, например:

```
double f(double i)
{
.....
}
```

то спецификация указателя на эту функцию будет выглядеть следующим образом:

```
double (* pf) (double i);
```

В этой спецификации первой операцией является * (взятие значения), поскольку она заключена в круглые скобки и, следовательно, имеет больший приоритет. Таким образом, переменная *pf* является указателем на любую функцию со следующей спецификацией:

```
double <имя_функции> (double i);
```

Для того чтобы использовать этот указатель, нужно сначала его проинициализировать, например:

```
pf= &sin;
```

В этом выражении указателю на функцию присваивается адрес функции *sin()*, имеющей такую же спецификацию, которая задана для этого указателя.

Вызов функции, адрес которой содержится в указателе на функцию, записывается следующим образом:

```
(*pf)(2.7);
```

Теперь функцию вычисления определенного интеграла можно

определить для любой математической функции и для указанного вместе с ней интервала значений ее аргумента следующим образом:

```
double INTEGRAL (double min, double max,
                   //интервал значений аргумента
                 double (*pf) (double a)
                //указатель на математическую функцию
                 )
{double step = 0.01;//шаг вычисления
double t = 0.0;//временная переменная
for (; min <= max; min = min + step)
t = t + (*pf) (min)//вызов функции
                 * step; //площадь прямоугольника
return t;
}//end
```

В этом примере используется метод прямоугольников.

Сразу же отметим, что указатель на функцию явно указывает, что это не функция, а адрес функции. Поэтому в языке Си не допускается вложенность функций, поскольку в этом случае адреса функции недостаточно для ее передачи в качестве параметра (требуется также передавать и ее контекст).

4.1.9. Указатель на структуру/объединение языка Си

Пусть требуется создать однонаправленный список. Для этого можно определить следующую структуру:

```
struct list
{int i;//значение элемента списка
struct list * next;//указатель на следующий элемент};
```

В этом примере компонент этой структуры *next* имеет тип *struct list* *, т.е. ссылается на тип, который еще не до конца определен. Поэтому язык Си требует до определения такого типа дать его предописание:

```
struct list;//предописание типа struct list

struct list
   {int i;//значение элемента списка
```

```
struct list * next;//указатель на следующий элемент};
```

То же самое справедливо и для подобных объединений.

Напомним также, что, как и в языке Паскаль, запрещено рекурсивное определение структур/объединений, как в следующем примере:

```
struct list
   {int i;//значение элемента списка
    struct list strv//структура того же типа};
```

В этом случае размер структуры будет бесконечен. По той же причине запрещена также косвенная рекурсия структур и объединений (это контролируется языком Си).

4.1.10. Пример программы, реализующей стек значений

Приведем пример программы, используемой для обработки стека. Пусть стек будет реализован как однонаправленный список с помощью следующего типа:

```
struct st {
  int i;//значение, вставляемое в стек
  struct st * prev; //указатель на предыдущий элемент стекам
                    //=NULL, если последний элемент

};
```

Пусть также имеется глобальная переменная (определенная вне функции), которая указывает на последний элемент стека:

```
struct st * LAST;//указатель на последний элемент стека
```

Если эта переменная равна NULL, то стек пуст.

Функции *push* и *pop* будут использоваться для вставки и выборки, соответственно, элемента из стека. Функция *init* используется для инициализации стека, а функция *drop* - для его удаления.

Составим программу:

```
#include <stdio.h>
struct st
{
   int i;//значение, вставляемое в стек
```



```
   struct st * prev; //указатель на предыдущий элемент стека
      //=NULL, если последний элемент
};

static struct st * LAST;
//указатель на последний элемент стека


//определение функции pop
struct st * //выдает указатель на вставленный элемент
            //NULL, если ошибка
       push (int i //вставляемое значение
            )
{struct st * p;

p = (struct st *)malloc(sizeof(struct st));
//выделение места для вставляемого элемента

if(!p) {printf("ошибка при выделении памяти\n");
        return NULL;}

p->i == i;//запись вставляемого значения

if (LAST)//в стеке уже имеется элемент
       p->prev = LAST;//связывание с последним элементом
else //вставка первого элемента в стек
       p->prev = NULL;
LAST = p;//теперь новый элемент - последний

return p;
}//end push

//определение функции pop
int *  //выдает вставленное значение
       //MAXINT, если ошибка или нет элементов

       pop(void)
```

```
{
int i = MAXINT;
if (LAST)//в стеке уже имеется элемент
    {i = LAST->i;//сохранение значения
     LAST = LAST->prev;
     free(LAST);//освобождение памяти, занятой элементом
    }
return i;
}//endpop


//определение функции init
void init(void)
{
LAST=NULL;
//инициализация указателя на последний элемент
}//end init

//определение функции drop
void drop(void)
 {
if (!LAST) return;//стек пуст
while (LAST)//пока в стеке есть элементы
    {struct st*t= LAST;
    LAST = LAST->prev;
    free(t);//освобождение памяти, занятой элементом
    }//end while
}//end drop
main() {
}//end main
```

4.2. Условия и общий порядок выполнения работы

См. раздел 1.2.

4.3. Примеры заданий

1. Написать подпрограмму сортировки одномерного массива, используя только указатели.

2. Написать подпрограмму сортировки многомерного массива, используя только указатели.

3. Написать подпрограмму сортировки 2-мерного массива целых по строкам (использовать сумму значений строки), сохранив эти значения в главной диагонали массива.

4. Написать подпрограмму сортировки 2-мерного массива целых по столбцам (использовать сумму значений столбца), сохранив эти значения во вспомогательной диагонали массива.

5. Написать подпрограмму сортировки 3-мерного массива целых по плоскостям (использовать сумму значений плоскости "строка-столбец"), сохранив эти значения в главной диагонали массива.

6. Написать подпрограмму сложения матриц (матрица – 2-мерный массив double).

7. Написать подпрограмму вычитания матриц (матрица – 2-мерный массив double).

8. Написать подпрограмму умножения матриц (матрица – 2-мерный массив double).

9. Написать подпрограмму получения обратной матрицы (матрица – 2-мерный массив double).

10. Написать подпрограмму транспонирования матрицы (матрица – 2-мерный массив double).

11. Написать подпрограмму, которая выдает длину строки.

12. Написать подпрограмму, которая подсчитывает число вхождений указанного символа во входной строке.

13. Написать подпрограмму, которая подсчитывает число вхождений указанной подстроки во входной строке.

14. Написать подпрограмму сравнения двух строк.

15. Написать подпрограмму, реализующую бинарное дерево.

16. Написать подпрограмму, реализующую очередь значений (в виде массива).

17. Написать программу, реализующую стек значений (в виде массива).

18. Написать подпрограмму, реализующую очередь значений (элемент очереди – **struct s {** int a; struct s * next;**}**).

19. Написать подпрограмму, реализующую стек значений

(элемент очереди – **struct s** { int a; struct s * next;}).

20. Написать подпрограмму, реализующую дек значений (с одной стороны – очередь, а с другой стороны – стек).

21. Написать подпрограмму, реализующую однонаправленный список значений (элемент списка – **struct s** { int a; struct s * next;}).

22. Написать подпрограмму, реализующую двунаправленный список значений (элемент списка – **struct s** { int a; struct s * next; struct s * prev; }).

23. Написать подпрограмму, вычисляющую определенный интеграл методом прямоугольников или трапеций (применять указатель на функцию).

24. Написать подпрограмму, вычисляющую определенный интеграл методом Симсона (применять указатель на функцию).

25. Написать подпрограмму сортировки массива указателей на функции, используя значения этих функций, выдающих значения типа int.

26. Написать переключатель, используя указатели на функции.

27. Написать подпрограмму сортировки 2-мерного массива указателей на функции, которые выдают целые значения, ПО СТРОКАМ (использовать сумму значений строки).

5. Лабораторная работа "ИСПОЛЬЗОВАНИЕ ВОЗМОЖНОСТЕЙ ОПЕРАЦИЙ ЯЗЫКА СИ И ПРЕПРОЦЕССОРА" (3 занятия)

5.1. Основные теоретические положения

5.1.1. Обзор операций языка Си

5.1.1.1. Приоритеты операций

Наивысший приоритет имеют операции *первичного выражения*. В их состав входят операции индексации массива, вызова функции, взятие компонента структуры/объединения, а также идентификаторы и литералы. Эти операции выполняются слева направо.

Следующий уровень приоритетов имеют *унарные операции*. Эти операции выполняются справа налево.

Следующий уровень приоритетов имеют *бинарные операции*, которые в свою очередь разбиваются по приоритетам (более

подробное описание см. в [1]). Порядок выполнения этих операций зависит от самих операций.

Ниже рассматриваются некоторые из *интересных* операций языка Си, которых нет в языке Паскаль.

5.1.1.2. Логические операции

Как было сказано выше, в языке Си нет логического типа, а есть логическая интерпретация целочисленных и указательных, значений. Результат любой логической операции имеет тип *int* и равен 1 или 0, если результат *логической интерпретации* равен *TRUE* или *FALSE*, соответственно.

Логическая операция *!* является унарной, соответствует булевской операции отрицания и выполняется следующим образом: вычисляется значение операнда, а затем оно логически интерпретируется. Если логическая интерпретация операнда равна TRUE, то значение описываемой операции будет 0, а в противном случае 1.

Не путайте логическую операцию *!* с побитовой операцией ~. Операция ~ применяется к целочисленному операнду, трактуемому как логический код: операция ~ просто инвертирует все его разряды. Приведем пример:

~01010101 => 10101010

Оставшиеся логические операции

a && b //логическое умножение

и

a || b //логическое сложение

выполняются следующим образом: сначала вычисляется первый операнд, а затем производится его логическая интерпретация. Если окажется, что результат логической операции уже можно определить по первому операнду, то второй операнд вообще не вычисляется. Приведем пример:

```
int a = 12;
……..
a = a || f(a);
```

В этом примере первый операнд (переменная *a*) не равен 0, поэтому его логическая интерпретация равна *TRUE*. Следовательно, результат логической операции || уже ясен, поэтому вызов функции *f(a)* (второй операнд операции) не будет выполнен.

Результат этой логической операции имеет тип *int*, а в переменную *a* будет записано значение 1.

Не путайте логические операции с побитовыми логическими операциями. Например, результат логической операции:

12 || 13

равен 1, так как логическая интерпретация первого операнда равна *TRUE*, а результат следующей операции:

12 | 13

равен 13. Действительно, 12, представленное здесь в десятичной системе счисления, равно 014 в восьмеричной системе счисления, а 13, представленное здесь в десятичной системе счисления, равно 015 в восьмеричной системе счисления. Затем переводим эти восьмеричные числа в двоичную систему счисления и произведем побитовое логическое сложение:

```
001100
|
001101
--------
001101
```

Полученный результат равен 015 в восьмеричной системе счисления или 13 в десятичной системе счисления.

5.1.1.3. Операции присваивания

Операция присваивания выполняется следующим образом: сначала вычисляется правый операнд, затем его результат преобразуется к типу левого операнда и записывается в левый операнд. Тип результата самой операции присваивания равен типу ее левого операнда.

Если в одном выражении указано несколько операций присваивания, то они выполняются справа налево. Приведем пример:

int a; float b,c;

c = a = b = 2.7;

Сначала в переменную *b* будет записано значение *2.7*, так как эта переменная имеет тип *float*; затем в переменную *a* будет записано значение *2*, так как эта переменная имеет тип *int*, а результат предыдущего присваивания равен вещественному значению *2.7*, у которого при его преобразовании к целому типу будет отброшена дробная часть; потом в переменную с будет записано значение *2.0* (целое значение *2*, преобразованное к типу *float*).

5.1.1.4. Условные операции

Условная операция

a ? b: c

выполняется так: сначала производится логическая интерпретация операнда *a*. Если ее результат равен *TRUE*, то результат условной операции будет *b*, в противном случае *c*. Вложенные условные операции выполняются справа налево.

5.1.1.5. Операции запятые

Операция запятая

a, b, c

выполняется слева направо, а ее результат равен последнему операнду. Она используется тогда, когда требуется произвести дополнительные действия перед выполнением результирующей (самой правой) операции (например, изменить значения некоторых переменных или произвести вызов процедуры с побочным эффектом).

5.1.2. Обзор операторов препроцессора

5.1.2.1. Общая характеристика операторов препроцессора

Все операторы препроцессора начинаются с символа #, чтобы их выделить в тексте программы. С точки зрения препроцессора, вся оставшаяся часть текста программы рассматривается как обычный текст (за исключением макровызовов), которая просто поступает на выход препроцессора. Поэтому препроцессор не использует особенностей синтаксиса языка Си (за исключением вызовов функций, имеющих синтаксис макровызовов) и может

производить обработку любого текста. Однако не злоупотребляйте этим свойством препроцессора языка Си.

5.1.2.2. Оператор include

Приведем пример *include-оператора*:

#include "имя_файла"

Этот оператор обрабатывается препроцессором так: строка с этим оператором исчезает и вместо нее подставляется текст указанного в этом операторе файла. Если имя файла указано в двойных кавычках, то предполагается, что этот файл размещается в текущем каталоге. Если в двойных кавычках указано путевое имя, то файл разыскивается по этому имени. Если имя файла указано в угловых скобках, то предполагается, что это системный файл языка Си, который размещается в специальном системном каталоге языка Си. Если файл не найден, выдается ошибка.

Приведем пример: пусть существует текстовый файл с именем *"qq"*, содержащий одну строку:

Погода

Тогда после обработки препроцессором файла, содержащего следующие строки:

#include "qq"
очень хорошая

будет получен следующий результат:

Погода
очень хорошая

Если сам подключаемый файл содержит *include-оператор*; то этот оператор обрабатывается так же. Не допускается рекурсия *include-операторов*.

В языке Си *include-оператор* используется для инкапсуляции (сокрытия от внешнего пользователя реализации) определений объектов стандартной библиотеки. Обычно подключаемые файлы содержат определения стандартных констант (например, *NULL*), типов (например, *FILE*) и переменных (например, *error*), а также описания стандартных подпрограмм (например, *printf*).

Если требуется использовать ту или иную стандартную библиотеку, то в начале программы указывается *include-оператор* для подключения описаний этой библиотеки. Так, например, если в программе будут использоваться стандартные подпрограммы ввода/вывода, то в начале программы указывается следующий *include-оператор*:

#include <stdio.h>

При этом пользователь не должен знать, как реализован тот или иной объект стандартной библиотеки.

Не *путайте* подключаемые файлы, которые содержат описания стандартной библиотеки и являются текстовыми файлами, с объектными файлами стандартной библиотеки, которые подкомпоновываются к объектному файлу на этапе компоновки и являются двоичными файлами.

5.1.2.3. Макроопределения и макровызовы

Среди текста исходной программы можно задать *макроопределение*, которое будет использоваться далее по тексту программы. Простейшая форма макроопределения имеет вид:

#define A 12

Это макроопределение задает макропеременную с именем *A* и со значением *12*. Все вхождения имени *A* далее по тексту будут заменяться на значение этой макропеременной, т.е. на *12*.

Макропеременные используются для определения констант: вместо того, чтобы вставлять в текст программы значение константы, можно сослаться на имя ее макропеременной; если затем потребуется изменить значение константы, то это нужно будет сделать только в ее макроопределении.

Макроопределение может также содержать параметры. Приведем пример такого макроопределения:

#defnef(a,b) a+b

В этой строке задано макроопределение с именем *f*, которое имя два формальных параметра: *a* и *b*. Затем в строке указано тело макроопределения: *a+b*. Если далее по тексту будет встречен *макровызов* типа *f(текст1, текст2)*, то он будет заменен (или,

как говорят, *расширен*) следующим образом: вместо него будет вставлено тело макроопределения, в котором формальный параметр a будет заменен на фактический параметр текст1, а формальный параметр *b* - на фактический параметр *текст2* (*текст1* и *текст2* могут включать запятые, но только в круглых скобках, поскольку запятые используются как разделители в макровызове). Приведем пример макровызова:

f(12,13)

Этот макровызов будет расширен следующим образом:

10+13

Макроопределения используются как второй способ реализации функций языка Си. Например, стандартную функцию *isdigit()* можно так реализовать с помощью макроопределения:

#define isdigit(ch) (((ch) >= '0') || ((ch) <= '9'))

С помощью макроопределений рекомендуется реализовывать только короткие функции, тела которых можно вставить в выражение.

В подключаемых файлах указывается либо заголовок стандартной функции, если эта функция реализована обычным способом, либо макроопределение функции. И пользователь не должен знать, какой способ использовался для определения функции. Заметим также, что заголовок функции используется для проверки правильности задания вызовов этой функции, а также при необходимости для выполнения операций преобразования ее фактических параметров к типам соответствующих формальных параметров.

5.1.2.4. Операторы условной макрогенерации

Пусть требуется вставить в программу проверку индексации, например:

```
#define TEST//определение макропеременной
...int m[ 100];//массив из 100 целых чисел
...
#ifdef TEST
if(i<0||i>=100)
      {printf("ошибка в индексе m[%d]\n",i);exit(-1);}
#endif
i=m[i];
```


В этом примере на этапе отладки программы - пока определена макропеременная *TEST* - будет производиться проверка правильности обращения к элементу массива *m[i]*. После того, как отладка программы будет закончена, можно убрать определение макропеременной *TEST*, что убрать и команды проверки правильности обращения *m[i]*.

Это типичный пример условной макрогенерации: пока установлена макропеременная *TEST*, оператор условной макрогенерации выдает компилятору его текст. Как только макропеременная будет удалена либо отменено ее определение с помощью макрооператора *#undef TEST*, оператор условной макрогенерации не будет срабатывать.

5.1.3. Интерфейс между ОС и программой на языке Си

Пусть был написан, откомпилирован и скомпонован весь программный проект, состоящий из файлов языка Си, в виде одного выполняемого файла с именем *filename*. При запуске этого файла в ОС с помощью команды:

filename par1 par2 par3

стартовой подпрограмме main() будут АВТОМАТИЧЕСКИ переданы следующие параметры:

явные, т.е. строки параметров "*filename*", "*par1*", "*par2*" и "*par3*" и их число (имя запускаемого файла трактуется как 0-й параметр);

неявные, т.е. значения переменных окружения в формате *"ИМЯ_ПЕРЕМЕННОЙ_ОКРУЖЕНИЯ=ЗНАЧЕНИЕ"*.

Функция *main()* имеет следующий интерфейс:

*int main(int n_par, char * PAR[], char * ENVVAR[])*

где *n_par* - это число явных параметров (в данном примере 4), *PAR* - массив указателей на строки, содержащие значения явных параметров (в данном примере *0*-й параметр ссылается на строку "*filename*", первый - на строку "*par1*" и т.п.) и *ENVPAR* - массив указателей на переменные окружения.

Эти параметры используется для управления программой.

Обычно явные параметры используются для задания часто меняющихся параметров программы. Переменные же окружения используются для задания достаточно редко меняющихся параметров, например: каталогов установки продуктов.

Значение конкретной переменной окружения можно узнать с помощью функции *getenv()*, имеющей следующую спецификацию:

```
char * /*указатель на значение переменной окружения */
     getenv("имя_переменной_окружения");
```

Если указанная переменная не установлена, то функция *getenv()* выдаст NULL.

С помощью функции *putenv()*:

```
int/* 0-успешное выполнение; -1- сбой*/
pu1env("имя_переменной_окружения=значение");
```

можно установить новую переменную окружения или изменить ее.

Функция *main()* выдает операционной системе в качестве результата целое значение. Это значение *(i)* возвращается с помощью либо оператора *return i*; внутри функции *main()*, либо вызова системной функции *exit(i)* в любом месте программы.

Если значение *i* равно 0, то оно трактуется в UNIX как *TRUE*; в противном случае - как *FALSE*.

5.1.4. Раздельная компиляция файлов языка Си

Пусть два пользователя решили составить свой программный продукт из двух файлов. В первом файле определяется функция *f1* и используется функция *f2*, определенная во втором файле;

а во втором файле наоборот - определяется функция *f2*, доступная для первого пользователя, и используется функция *f1*, определенная в первом файле.

Кроме того, в каждом файле имеются внутренние функции, доступные только внутри этого файла и недоступные извне. Эти последние функции образуют инкапсуляционную часть файла, так как в них скрыта реализация алгоритма в этом файле.

Таким образом, каждый файл состоит из экспортной части (предоставляющей ресурсы другим), импортной части (потребляющей ресурсы других) и инкапсуляционной части, отвечающей за реализацию и недоступной для внешнего мира.

Перед определением экспортной функции не ставится никакой спецификатор, перед описанием импортной функции ставится спецификатор *extern*, а внутренняя функция помечается спецификатором *static*. Те же рассуждения справедливы и для переменных, определенных вне функций.

На практике поступают проще: все экспорт/импорт спецификации помещают в отдельный include-файл, который используется в обоих файлах.

5.2. Условия и общий порядок выполнения работы

См. Раздел 1.2.

5.3. Примеры заданий

Преподаватель предложит результирующую курсовую работу, охватывающую весь пройденный материал.

БИБЛИОГРАФИЧЕСКИЕ ОПИСАНИЯ

Керниган Б., Ритчи Д.. Язык программирования Си. М,: Финансы и статистика, 1992. - 280 с.

Уэйт Д., Прата Б. Язык Си. М.: Мир, 1986. - 160 с.

Страуструп Б. Язык программирования С++. М.: Радио и связь, 1991. - 348 с.

Подбельский. Язык С. М.: Мир, 1996. - 404 с.

Улджер Д. С++. Санкт-Петербург: Питер, 1999. - 290 с.

Пол А. Объектно-ориентированное программирование на Си++. М.: BINOM, 1999. - 462 с.

Прата С. Язык программирования С. Лекции и упражнения. М: DiaSoft, 2002.

СОДЕРЖАНИЕ