МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра «Интеллектуальные информационные технологии»

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ СИ

Методические указания к выполнению
лабораторных работ по дисциплине

«Основы алгоритмизации и программирования в традиционных и интеллектуальных компьютерах»

для студентов специальности
1-40 03 01 «Искусственный интеллект»

Брест 2013

УДК 347 77/681.3
ББК 67.403.3 73/32.97

В методических указаниях приведены необходимые теоретические сведения по основам программирования на языке Си. Методические указания содержат информацию о структуре языка Си, структуре программы, основных и составных типах данных языка Си, функциях, основных операторах языка Си, принципах работы с файлами, динамических структурах. В методических указаниях приведены конкретные примеры реализации программ, для закрепления теоретического материала.

Методические указания предназначены для использования студентами специальностей: 1 – 40 03 01 «Искусственный интеллект», 1 – 53 01 02 «Автоматизированные системы обработки информации» в ходе выполнения лабораторных, практических и контрольных работ по дисциплине «Основы алгоритмизации и программирования», «Основы алгоритмизации и программирования в традиционных и интеллектуальных компьютерах», «Общеинженерная практика».

Составитель: Хацкевич М.В., старший преподаватель

СОДЕРЖАНИЕ

Лабораторная работа №1

Тема: «Форматный ввод /вывод»

Элементы языка С. ***Основные символы***

Множество символов, используемых в языке С, можно разделить на пять групп: символы (прописные и строчные буквы английского алфавита и подчеркивание (_)); арабские цифры; разделители(, . ; : ? ' " ! | / \ ~ * () {} [] <> + - & # % = _); пробельные символы; ESC-символы.

\a – звуковой сигнал	\b – возврат на шаг	\f – перевод страницы
\n – новая строка	\r – возврат каретки	\t – гориз. табуляция
\v – верт. Табуляция	\\ – наклонная черта	\" – двойная кавычка
\' – одиночная кавычка	\ooo – 8-ричный код	\xhh – 16-ричный код.

Ключевые слова

Зарезервированные языком С служебные слова, имеющие определенный смысл для компилятора, приведены в таблице 1.1

Таблица 1.1

auto	Do	for	Return	switch	const
break	Double	goto	Short	typedef	restrict
case	Else	if	signed	union	volatile
char	Enum	int	Sizeof	unsigned	_Bool
continue	Extern	long	Static	void	_Complex
default	Float	register	Struct	while	_Imaginary

Идентификаторы

Идентификатором называется имя какого-либо объекта в программе (переменной, функции и т. д.).

Для образования идентификаторов могут быть использованы строчные или прописные буквы латинского алфавита, цифры и символ подчеркивание '_'. Первым символом в идентификаторе должна быть буква. Допускается использование в качестве первого символа знака подчеркивания '_'. Идентификатор не должен совпадать с ключевыми и зарезервированными словами, именами библиотечных функций.

Константы

Константами называются неизменяемые в программе величины. В языке С определены следующие типы констант: целые; с плавающей точкой; символьные; строковые (литералы).

Целая константа – целое положительное число, представленное в десятичной, восьмеричной или шестнадцатеричной системах счисления.

Десятичная константа состоит из одной или нескольких десятичных цифр, причем первая цифра не должна быть нулем (в противном случае число будет воспринято как восьмеричное), например, 385.

Восьмиричная константа состоит из обязательного нуля и одной или нескольких восьмиричных цифр (0,1,2,3,4,5,6,7), например, 017.

Шестнадцатиричная константа начинается с обязательной последовательности 0x или 0X и содержит одну или несколько шестнадцатеричных цифр (0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F), например, 0x6E или 0x6e.

Для формирования отрицательной целой константы используют знак «минус» перед записью константы, называемый унарным минусом:–42, –073, –0x5F.
Тип константы определяется автоматически в зависимости от значения константы, при этом выбирается самый компактный тип.
Для того чтобы любой целой константе был присвоен тип long, достаточно в конце константы поставить букву l или L, например, 5l,6l, 128L,0105L,0X2A1L.
Константа с плавающей точкой представляется в виде десятичного числа в формате с фиксированной точкой ([цифры] . [цифры]) или с плавающей точкой ([цифры]. [цифры] [E/e [+|–] цифры]), т. е. в экспоненциальном (полулогарифмическом) виде.
Константы с плавающей точкой состоят из целой и дробной части и (или) экспоненты. Константы с плавающей точкой по умолчанию представляют собой положительные величины удвоенной точности, т. е. имеют тип double. Например, 3.14159; 3E8; 1.; .2; 3.73e–19.
Если константа заканчивается символом F или f, то она имеет тип float, а если L или l, то long double (42.F; 1.602e –19L).
Символьная константа представляется символом, заключенным в апострофы:'m','7',';' . Значением символьной константы является числовой код символа таблицы ASCII. Символьные константы имеют тип char.
Строковая константа (литерал) – это последовательность символов, включающая строчные и прописные буквы русского и латинского алфавитов, цифры и знаки, заключенные в кавычки, например, "C++","Hello, Mary","2005 год". В конец каждого строкового литерала компилятором добавляется нулевой символ, представляемый управляющей последовательностью '\0'.Строковый литерал имеет тип static char (статический массив символов).

Лексемы

Лексемой называется единица текста программы, которая имеет определенный смысл для компилятора и не может быть в дальнейшем разбита на составные части. В языке С лексемами являются знаки пунктуации, все виды скобок, знаки операций, константы, идентификаторы, ключевые слова.

Комментарии

Комментарий является некоторым пояснительным текстом и представляет собой последовательность символов, заключенных в ограничительную конструкцию /* и */. Символы могут быть любые, включая символ новой строки, но исключая */ (конец комментария). Комментарии могут занимать более одной строки, но не могут быть вложенными. Формат комментария: */*-символ*/*, например, */*Это комментарий */*.
Комментарий может начинаться значком //, а заканчиваться символом новой строки.

Типы данных языка

Иерархию типов данных можно представить следующей схемой.
Простые (скалярные) типы: целые; вещественные; символьные; указатели; перечислимый тип.
Составные (структурированные) типы: массив; структура; объединение.
Переменная простого (скалярного) типа в любой момент времени хранит только одно значение. В отличие от простых переменных, переменные составного (структурированного) типа одновременно хранят несколько значений. Целые и вещественные переменные предназначены для хранения чисел, символьные переменные – это также числовые переменные, они хранят ASCII коды символов.

Числовые типы данных

Числовые типы данных языка С представлены в таблице 1.2

Таблица 1.2

Тип данных	Размер памяти, бит	Диапазон значений
char (символьный)	8	от–128 до 127
signed char (знаковый символьный)	8	от–128 до 127
unsigned char (беззнаковый символьный)	8	от 0 до 255
short int (короткое целое)	16	от–32768 до 32767
unsigned int (беззнаковое целое)	16	от 0 до 65535 (16-битная платформа) от 0 до 4294967295 (32-битная платформа)
int (целое)	16 32	от–32768 до 32767 (16-битная платформа) от –2147483648 до 2147483647 (32-битная платформа)
long (длинное целое)	32	от–2147483648 до 2147483647
unsigned long (длинное целое без знака)	32	от 0 до 4294967295
float (вещественное)	32	от 3.4E–38 до 3.4E38
double (двойное вещественное)	64	от 1.7E–308 до 1.7E308
long double (длинное вещественное)	80	от 3.4E–4932 до 3.4E4932

В 32-разрядной ОС Windows тип int занимает в памяти 32 бита, и диапазон допустимых значений для знакового int в этом случае от минус 2147483648 до 2147483647. Такое различие в размере памяти, выделяемой под переменную типа int, объясняется тем, что тип int – машинно-зависимый, и для него выделяется одно машинное слово, длина которого в 16-разрядных процессорах – 16 бит, в 32-разрядных – 32 бита.

Объявление переменных

Одним из отличий языка С от ряда других языков программирования является отсутствие принципа умолчания, что приводит к необходимости явного объявления всех переменных, используемых в программе, вместе с указанием соответствующих им типов. Объявление переменной имеет следующий формат:

[спецификатор_класса_памяти] спецификатор_типа идентификатор [=инициатор];

Спецификатор класса памяти определяется одним из 4 ключевых слов языка С: *auto, extern, register, static* и указывает, во-первых, каким образом будет распределяться память под объявляемую переменную и, во-вторых, область видимости этой переменной, т. е. из каких частей программы можно к ней обратиться.

Спецификатор типа - одно или несколько ключевых слов, определяющих тип объявляемой переменной. Инициатор задает начальное значение или список начальных значений, присваиваемых переменной при объявлении.

Примеры инициализации переменных: int i=5; float f=12.35; char ch='a';

Переменная любого типа может быть объявлена немодифицируемой, что достигается добавлением ключевого слова const к спецификатору типа. Объекты с типом const представляют собой данные, используемые только для чтения, т. е. этой переменной не может быть присвоено новое значение: например, const int a=5.

Отметим, что если после слова const отсутствует спецификатор типа, то подразумевается спецификатор типа int. Ключевое слово void означает отсутствие типа.

Данные целого типа

Для определения данных целого типа используются ключевые слова char, int, short, long, которые определяют диапазон значений и размер области памяти, выделяемой под переменные. Переменная типа char занимает в памяти 1 байт, short – 2 байта, long – 4 байта. Размер переменной типа int определяется типом процессора.

При объявлении целых типов можно использовать ключевые слова signed и unsigned, которые указывают, как интерпретируется старший бит объявляемой переменной. Если указано ключевое слово unsigned, то старший бит интерпретируется как часть числа, в противном случае старший бит интерпретируется как знаковый. В случае отсутствия ключевого слова unsigned целая переменная считается знаковой. В том случае, если спецификатор типа состоит из ключевого типа signed или unsigned и далее следует идентификатор переменной, то она будет рассматриваться как переменная типа int. Отметим, что ключевые слова signed и unsigned не обязательны. В памяти данные хранятся в двоичном коде. На рисунке 1.1 изображено внутреннее представление данных целого типа.

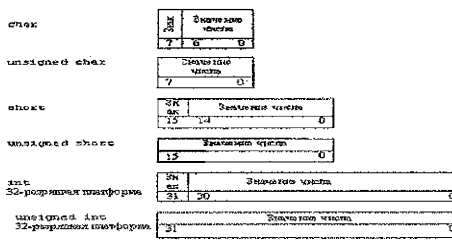

Рисунок 1.1 – Внутреннее представление данных целых типов

Переменная типа char(signed char) занимает в памяти 1 байт, при этом старший бит хранит информацию о знаке числа: 0 соответствует положительному числу, 1 – отрицательному. Биты с 0-го по 6-й используются для записи значения числа. Запись в каждый из этих битов значения 1 соответствует наибольшему положительному числу, равному 127, при этом старший бит установлен в 0. Такое представление целых чисел называется прямым кодом. Для хранения отрицательных чисел используется представление чисел, называемое дополнительным кодом.

Получить дополнительный код отрицательного числа можно по следующему правилу:

- в биты, предназначенные для хранения значения числа, записывается модуль отрицательного числа в прямом коде;

- в старший (знаковый) бит помещается 1;
- в битах, предназначенных для хранения значения числа, формируется обратный код, т. е. 1 заменяется на 0, а 0 на 1;
- к обратному коду числа прибавляется 1.

Рассмотрим дополнительный код для числа –1 (рисунок 1.2).

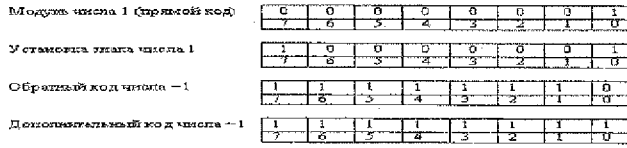

Рисунок 1.2 – Дополнительный код отрицательного числа

Минимальное по модулю отрицательное число представлено единицами во всех двоичных разрядах, предназначенных для хранения числа. Если в эти биты записать нули, получится наибольшее по модулю отрицательное число. Для переменной типа char это значение равно минус 128.

Переменная типа unsigned char хранит целые положительные значения, при этом все 8 бит используются для записи числа. Такое внутреннее представление числа позволяет записывать в переменную значения в диапазоне от 0 до 255.

Тип char по сравнению с другими целыми типами имеет особое назначение: он используется для представления символа или объявления строковых литералов. По умолчанию тип char или signed char интерпретируется как однобайтовая целая величина со знаком и с диапазоном значений от минус 128 до 127, хотя только значения в диапазоне 0 – 127 имеют символьные эквиваленты. Для представления символов русского алфавита модификатор типа идентификатора данных имеет вид unsigned char, так как коды русских букв превышают величину 127.

Данные вещественного типа

Для определения данных, представляющих число с плавающей точкой, используются ключевые слова float, double, long double.

Величина типа float занимает в памяти 4 байта, из которых 1 бит отводится для знака, 8 битов для порядка и 23 бита для мантиссы. Порядок хранится в виде двоичного положительного числа, которое получается при сложении величины порядка и числа 127 (1111111_2). Мантисса хранится в нормализованном виде, т. е. ее значение должно быть в диапазоне от 1 до 2, а старший бит равен 1. Если нормализация мантиссы нарушается, выполняется сдвиг мантиссы до тех пор, пока старшей цифрой не окажется единица, одновременно при каждой операции сдвига мантиссы происходит изменение величины порядка. Так как старшая цифра в нормализованной мантиссе – единица, она отбрасывается. Отброшенная единица называется неявной единицей. Диапазон значений переменной с плавающей точкой приблизительно равен от 3.4E–38 до 3.4E+38 , а точность (количество значащих цифр) равна 7.Рассмотрим, каким образом в ячейках памяти, выделенных под переменную типа float, хранится конкретное число.

Пусть float x=5.375; в двоичной системе $x_2=101.011$ или $x_2=1.01011 \cdot 2^{10+1111111}$, тогда с учетом того, что в нормализованной мантиссе первая цифра всегда 1, и эта единица отбрасывается, а порядок сдвигается на 127, получаем внутреннее представление числа.

0	1	0	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Знак	Порядок								Мантисса																							

Величина типа double занимает 8 байт в памяти, формат которой аналогичен формату float. Биты памяти распределяются следующим образом: 1 бит – знак, 11 битов – порядок и 52 бита – мантисса. Порядок сдвигается на величину 1023. С учетом опущенного старшего бита мантиссы диапазон значений равен от 1.7E–308 до 1.7E+308, а точность составляет 15 десятичных значащих цифр.

Выражения

Выражение - совокупность элементов данных (переменных, констант и вызовов функций), объединенных знаками операций и скобками. Каждое выражение должно иметь свое значение. Результат вычисления выражения зависит от расположения знаков операций и круглых скобок в выражении, а также от приоритета выполнения операций. Элементы данных, используемые в выражении, называются операндами. Каждый операнд имеет тип. При вычислении выражений тип каждого операнда может быть преобразован к другому типу. Преобразования типов бывают неявными (при вычислении выражений и вызовах функций) или явными (при выполнении операций преобразования типов).

Неявное преобразование из типа int в тип float: int a=5; float b=a;Явное преобразование типа int в тип float с помощью операции преобразования типов (тип в явном виде указывается в скобках перед преобразуемой переменной): int a=5; float b=(float)a;

Если в качестве операнда используется числовая константа, то тип операнда устанавливается в зависимости от значения константы. Константа, представленная целым числом, может быть типа int, long, unsigned int или unsigned long в зависимости от ее значения и от формы записи. По умолчанию компилятор приписывает константе тип наименьшего размера, в ячейку которого может уместиться константа. Но это правило имеет исключение: всем вещественным константам, даже самым маленьким, приписывается тип double.

Символьная константа требует один байт памяти и имеет тип char. Строковый литерал состоит из последовательности символов, заключенных в кавычки, и представляется в памяти как массив элементов типа char, инициализируемый указанной последовательностью символов. Выражение рассматривается как имеющее значение true, если результат вычислений не равен нулю, и false в противном случае.

Операции

Арифметические операции

К арифметическим операциям относятся операции сложения (+), вычитания (–), умножения (*), деления (/) и деления по модулю (%) – вычисление остатка от деления. В языке отсутствует операция возведения в степень.

В одном и том же выражении могут встретиться переменные и константы разных типов, в этом случае они преобразуются к старшему типу в соответствии с приоритетами типов, установленными в языке С: long double – double – float – unsigned long – long – unsigned int – int – unsigned char – char.

В этом перечислении наивысший приоритет имеет тип long double. Компилятор преобразует "меньший" тип в "больший". Например, если в выражении операнды имеют типы long double, double, float и т. д., то все они на время вычислений автоматически преобразуются к старшему типу. В данном случае это будет тип long double. Тип результата будет соответствовать старшему типу в выражении.

Отметим, что в соответствии с правилами выполнения вычислений в С при делении двух операндов целого типа результат также является целочисленным и формируется путем отбрасывания дробной части частного от деления: 4/7=0; 7/4=1.

Деление по модулю выполняется над операндами целого типа и результатом операции является остаток от деления: 7%4=3; 4%7=0.

Приоритеты арифметических операций:

- умножение, деление, деление по модулю;
- сложение, вычитание.

Операции одного уровня выполняются слева направо.

Помимо стандартных арифметических операций используются две дополнительных операции: инкремент (++) и декремент (– –). Результатом выполнения инкремента будет увеличение значения операнда на 1, результат декремента – уменьшение операнда на 1. Обе операции идентичны, поэтому рассмотрим на примере только операцию инкремента (увеличения).

Операция инкремента реализуется в двух видах: ++x; x++; аналогично декремент:– – x; x– –; где x – идентификатор переменной.

Результатом операции является увеличение значения переменной x на 1, причем в первом случае x сначала увеличивается на 1, а затем используется в дальнейших операциях, а во втором – значение x сначала используется, а потом увеличивается на 1. Различие операций сказывается при использовании их в сложных выражениях. Например, определить значения x, t, z, y после вычисления выражений при x=1, t=1: y=++x; z=t++; Ответ: x=2; t=2; z=1; y=2.

Операция присваивания

Операция присваивания определена в двух видах: простое и составное присваивание.

При простом присваивании, обозначаемом знаком (=), значение левого операнда заменяется значением правого операнда с соответствующим преобразованием типа правого операнда.

Структура оператора x=W, где x – переменная; W – выражение, например, x=2; y=k–8. В приведенной структуре оператора W является выражением, поэтому оно может содержать и операции присваивания. Если в выражении присутствуют несколько операций присваивания, то они выполняются справа налево, например, x=y=z=6.8;

*При составном присваивании используются следующие операции: +=, –=, *=, /=, %=.*

Формат операции составного присваивания: x OP W, где x – переменная, OP – операция, W – выражение. В составном присваивании вначале выполняется операция над x и W, а затем результат присваивается переменной x.

Например, пусть x=–4: x+=3; //результат x=–1; x–=3; //результат x=–7;

Если порядок выполнения операций не определен круглыми скобками, то их приоритеты приведены в таблице 1.3, при этом высший приоритет имеют операции инкремента и декремента, а наименьший – присваивания.

Таблица 1.3

++, —	Справа налево	
*, /, %	Слева направо	Порядок выполнения операций одного уровня
+, –	Слева направо	
=, *=, /=, %=, +=, -=	Справа налево	

Простейшая программа

Препроцессор и его функции

Препроцессор – программа, используемая для обработки исходного текста программы на языке С до компиляции и выполняющая следующие действия: поиск и включение в программу нужных внешних файлов; изменение условий компиляции; определение значений констант и т. д.

Препроцессор "общается" с программой при помощи директив. Директивы препроцессора представляют собой инструкции, записанные в тексте программы и выполняемые до ее трансляции. Директивы препроцессора отмечаются специальным маркером #. Знак # должен быть первым символом в строке, содержащей директиву. Между знаком # и первой буквой директивы могут находиться пробелы. После директив препроцессора точка с запятой не ставится.

Директивы препроцессора могут встречаться в любом месте программы, но обычно их стараются помещать в начале для удобства восприятия текста программы. Директивы, появляющиеся в любом месте исходного файла, применимы только к тексту, идущему после этой директивы.

Основные директивы препроцессора

Директива include

Директива включает в текст программы содержимое указанного файла и имеет две формы:

#include"имя файла"

#include <имя файла>

Имя файла должно соответствовать соглашениям операционной системы и может состоять только из имени файла либо из имени файла с указанием пути к этому файлу. Способ поиска файла зависит от того, заключено ли его имя в двойные кавычки или в угловые скобки. Если имя файла задано в угловых скобках, поиск файла проводится в специальном каталоге. Обычно таким каталогом является каталог INCLUDE интегрированной среды разработки (IDE). Если имя файла указано в кавычках, то поиск файла начинается с текущего каталога, а если файл не найден, поиск продолжается в каталоге INCLUDE.

Директива include широко используется для включения в программу так называемых заголовочных файлов, содержащих прототипы библиотечных функций.

Заголовочный файл (объект-заголовок) снабжает компилятор необходимой информацией о данных и функциях, которые могут использоваться в программе, имеет расширение .h (header) и представляет собой текстовый файл. Заголовочный файл может содержать определение типов, прототипы функций, объявление внешних переменных (extern), директивы препроцессора и комментарии. Любая С-программа должна содержать хотя бы одну директиву. Так, для реализации ввода/вывода должна присутствовать директива:

#include <stdio.h>

Файл stdio.h (standard input/output header) содержит необходимую информацию о средствах ввода/вывода информации:

- определение типа данных FILE (поток ввода/вывода идентифицируется указателем на переменную типа FILE);
- описание потоков stdin и stdout (ввод/вывод); по умолчанию эти файлы связаны с терминальным оборудованием (ввод с клавиатуры, вывод на дисплей);
- файл вывода сообщений об ошибках stderr;
- определение макроса NULL (нулевой указатель, т. е. значение, не указывающее ни на какой объект);
- определение макроса EOF (признак конца файла).

Для реализации других действий в программу с помощью данной директивы следует включать иные заголовочные файлы, например: преобразование строк #include <string.h>; работа с математическими функциями #include <math.h>; библиотечные функции #include <stdlib.h>

***Директива* define**

Директива define служит для замены часто использующихся констант, ключевых слов, операторов или выражений некоторыми идентификаторами - макросами. Идентификаторы, заменяющие текстовые или числовые константы, называют именованными или символическими константами. Идентификаторы, заменяющие фрагменты программ, называют макроопределениями, причем макроопределения могут иметь аргументы. Директива define имеет две синтаксические формы:

#define идентификатор текст
#define идентификатор (список параметров) текст

Эта директива заменяет все последующие вхождения идентификатора текстом. Такой процесс называется макроподстановкой. Текст может представлять собой любой фрагмент программы на С, а может и отсутствовать вовсе. В последнем случае все экземпляры идентификатора удаляются из программы. Идентификатор (макрос) в директиве define принято записывать прописными буквами:

```
#define WIDTH 80
#define LENGTH (WIDTH+10)
```

Во второй синтаксической форме в директиве define имеется список формальных параметров, который может содержать один или несколько идентификаторов, разделенных запятыми. Формальные параметры в тексте макроопределения отмечают позиции, на которые должны быть подставлены фактические аргументы макровызова. Каждый формальный параметр может появиться в тексте макроопределения несколько раз. При макровызове вслед за идентификатором записывается список фактических аргументов, количество которых должно совпадать с количеством формальных параметров.

Пример макроопределения:

```
#define SQUARE(X) (X)*(X)
```

Отметим, что скобки необходимы (как и в предыдущем примере) для обеспечения правильного порядка действий. Теперь при появлении в программе выражения Z=SQUARE(2); переменная Z получит значение, равное 4.Препроцессор не выполняет проверки синтаксической правильности текста подстановки, поэтому использование конструкции #define PI = 3.1415 приведет к тому, что вместо идентификатора PI везде будет подставляться текст "= 3.1415", что явно не соответствует желаемому. Ошибка будет обнаружена только на этапе компиляции.

Директива undef

Директива undef используется для отмены действия директивы define, синтаксис которой

#undef идентификатор

Директива отменяет действие текущего определения define для указанного идентификатора. Не является ошибкой использование директивы undef для идентификатора, который не был определен директивой define, например:

#undef WIDTH

#undef MAX.

Структура и правила составления программ

Структура функции

С-программа состоит из функций, которые являются строительными элементами языка. Каждая функция имеет заголовок и тело. Заголовок включает в себя имя функции, список аргументов, заключенный в круглые скобки, и тип возвращаемого функцией значения. За строкой, содержащей имя функции, идет тело функции, заключенное в фигурные скобки. В общем виде синтаксис функции выглядит следующим образом:

возвр_тип имя_функции (список_параметров)

{

тело функции

}

Тело функции содержит последовательность операторов языка, выполняющих некоторые действия.

Возвр_тип определяет тип значения (результата), возвращаемого функцией. Функция может возвращать значение любого типа, за исключением массивов. Имя_функции представляет собой идентификатор, после имени функции обязательно следуют круглые скобки. В скобках указывается список формальных параметров. Каждый элемент списка формальных параметров состоит из имени переменной и ее типа. Все параметры функции должны объявляться отдельно, причем для каждого из них надо указывать и тип, и имя. При вызове функции формальные параметры заменяются значениями фактических аргументов. Список параметров может быть и пустым. Такой пустой список можно указать в явном виде, поместив внутри скобок ключевое слово void, или оставить скобки пустыми: void MyFunction(void) {…..}.

Функции С, возвращающие значения, делают это с помощью оператора возврата return, за которым следует возвращаемое значение. В функциях типа void используется оператор return без значения или оператор return не используется вовсе.

Функцию перед использованием необходимо объявлять. Объявление функции называется прототипом функции. Например, для функции MyFunction(), приведенной ранее, прототип будет иметь следующий вид:

void MyFunction(void); или void MyFunction();

Прототипы дают компилятору возможность тщательно выполнить проверку типов аргументов, правильность их преобразования, обнаружить несоответствия в количестве аргументов, использованных при вызове функции, и в количестве параметров функции.

Функция main()

Одна функция в любой С-программе является главной и имеет имя main. Таким образом, любая программа должна содержать хотя бы одну функцию и обязательно одна из функций программы должна иметь имя main. Именно с этой функции начинается выполнение программы. В хорошо структурированной программе главная функция всегда содержит действия, отражающие сущность решаемой задачи, чаще всего это вызовы

функций. Простейшая программа состоит из функции main(). Вслед за именем в круглых скобках идет список аргументов. Функция main() может и не иметь аргументов, тогда у нее будет следующий заголовок: возвр_тип main()

Стандарт ISO/ANSI C требует наличия перед функцией main() типа возвращаемого результата. В качестве возвращаемого значения рекомендуется использовать тип int, тогда функция main() будет иметь вид

```
int main()
   {  return 0;  }
```

Допускается также указывать функции main() тип возвращаемого значения void:

```
void main() {  }
```

Структура простой программы

После рассмотрения отдельных составных частей программы определим общие правила, используемые при разработке С-программ:

- в начале программы перечисляются директивы препроцессора;
- программа состоит из совокупности функций, одна из которых должна называться main();
- описание функции состоит из заголовка и тела функции и называется определением функции или ее реализацией;
- заголовок функции содержит тип возвращаемого функцией значения, имя функции и список параметров;
- имя функции узнается по круглым скобкам, которые идут за именем и могут быть пустыми;
- тело функции заключено в фигурные скобки и состоит из ряда инструкций.

Итак, имеем простую программу:

```
#include ....//директивы препроцессора
int main (void)
{объявления переменных;
инициализация переменных;
вызовы функции;
return 0;}
```

Правила записи объявлений, операторов и комментариев

Объявление – это предварительное знакомство с объектом программы. При объявлении переменной указывается тип и имя переменной. Если надо объявить несколько переменных одного типа, они перечисляются через запятую: double x, y, z;. Можно одновременно объявить переменную и присвоить ей начальное значение (инициализировать или определить): float f=3.5;

Оператор – это инструкция для компьютера, обозначающая какие-либо действия над данными (например, присваивание) или управляющая выполнением программы (операторы if, for и т. д.). В языке С используются операторы двух видов: простые и составные.

Простой оператор всегда заканчивается символом ;. Символ ; является частью простого оператора. *Составной оператор* – это последовательность простых операторов, заключенная в фигурные скобки. Операторы могут размещаться по два и более в одной строке. Тело функции может начинаться сразу же в следующей позиции за открывающей фигурной скобкой.

Пример простейшей программы

Приведем в качестве примера простейшую программу вычисления выражения "дважды два равно четыре".

```
/* First C-program*/
#include <stdio.h>
int main()
{float x=2.,y;
y=x*x;}
```

#include <stdio.h> - с помощью этой строки производится подключение заголовочного файла stdio.h, предоставляющего доступ к средствам ввода/вывода информации. Содержимое этого файла копируется препроцессором в файл программы на место директивы include. Файл stdio.h содержится в специальном каталоге. Обычно это каталог INCLUDE , являющийся составной частью среды разработки. main() - в каждой программе должна присутствовать эта функция. При запуске программа "проходит" через каждую строку кода в функции main() и исполняет ее.

Можно выделить четыре неотъемлемых части в определении функции: тип возвращаемого результата; имя функции; список аргументов (параметров); тело функции. Итак: возвращаемый тип: int; имя функции: main(); список аргументов: (); тело функции: {...}.

Средства ввода/вывода

Ввод/вывод информации в языке С осуществляется с помощью функций, объявленных в заголовочном файле stdio.h:

- printf() – консольный вывод;
- scanf() – консольный ввод.

Функция форматированного вывода printf()

Структура (синтаксис) обращения к функции:

printf ("строка формата", арг1, арг2, ..., аргN);

В качестве *аргументов функции* арг1...аргN используются идентификаторы переменных или выражения. Список аргументов может быть пустым. *Строка формата* записывается в двойных кавычках и может содержать: любой текст; спецификаторы форматов (по количеству аргументов), обозначаются символом %, содержат информацию о типе выводимого значения и его модификации; управляющие символы.

Основные форматы

Каждому аргументу должен соответствовать только один спецификатор формата. Вид формата определяется типом аргумента.

Целые числа:

- %d – аргумент рассматривается как целое 10-тичное число со знаком;
- %u – целое 10-тичное без знака;
- %x – 16-ричное целое без знака;
- %o – 8-ричное целое без знака.

Вещественные числа:

- %f – аргумент записывается в форме вещественного числа с фиксированной точкой (например, 0.036);
- %e – аргумент записывается в форме числа с плавающей точкой.

Символы и строки:

- %c – аргумент рассматривается как значение типа char, и на экран выводится один символ;
- %s – аргумент рассматривается как строка.

Пример,

```
#include <stdio.h>
int main(void)
{  float x=2.,y;
   y=x*x;
   printf("\n y(%f)=%f", x, y);
   return 0;}
```

Управляющая строка содержит ESC-символ перевода строки '\n', текст и два спецификатора формата (%f): первый для аргумента x, второй – для аргумента y.

Модификации форматов

Спецификаторы формата целых чисел можно записать в виде: %Nd, %Nu, %No, %Nx, где N – натуральное число, определяющее количество позиций, отводимых под значения числа (ширина поля вывода).

Для форматов вещественных чисел спецификатор формата представляется в виде: %N.Mf или %N.Me, где N и M – натуральные числа, определяющие общее число позиций, отведенных под число, N (ширина поля вывода) и число знаков в дробной части M (точность отображения).

Модифицируем форматы в нашем примере:

```
#include <stdio.h>
int main(void)
{float x=2.;
printf("\n y(%4.1f)=%6.2f", x, x*x);
return 0;}
```

Результат на экране: y(2.0) = 4.00.

При неправильно заданном формате, когда ширина поля вывода оказывается меньше, чем необходимо для представления значения переменной, компилятор языка добавляет недостающие позиции, исправляя ошибку программиста. Аналогично можно форматировать вывод символьных строк: %N.Ms, здесь в спецификаторе преобразования N – ширина поля вывода строки, M – максимальное количество выводимых на экран символов. Примеры форматирования данных на экране с помощью функции printf() приведены в таблице 1.4

Таблица 1.4

Аргумент		Спецификатор формата	Результат
значение	тип		
123	int	%d	123
123	int	%10d	(7 пробелов) 123
123	int	%x	7B
64000L	long	%ld	64000
'x'	char	%c	x
'x'	char	%5c	(4 пробела) x
"поле"	Указатель строки	%s	поле
"поле"	Указатель строки	%10s	(6 пробелов) поле
123.45	float	%f	123.450000
123.45	float	%10.3f	(3 пробела) 123.450
123.45	float	%e	1.234500E+02
123.45	double	%10.3lf	(3 пробела) 123.450

Функция форматированного ввода scanf()

Структура (синтаксис) обращения к функции:

scanf("строка формата", список аргументов);

С помощью данной функции производится ввод с клавиатуры значений переменных, перечисленных в списке аргументов в формате, определенном строкой формата. Функция преобразует последовательность вводимых символов в различные формы: целые числа, числа с плавающей точкой, символы и строки С.

Особенности функции:

- в языке С в строке формата рекомендуется писать только спецификаторы форматов; количество спецификаторов должно быть равно числу аргументов в строке; каждый спецификатор имеет ту же структуру, что и в функции printf();
- список аргументов может состоять из одного или нескольких аргументов; разделителем в списке аргументов служит [,];
- в качестве аргументов функции используются только адреса переменных.

Задачей аргумента в данной функции является указание адреса ячейки памяти, куда должно быть помещено вводимое значение. Так, символ & обозначает операцию получения адреса переменной, т. е. конструкция &p обеспечивает ввод значения в ячейку памяти, где размещена переменная p.

При использовании функции scanf() необходимо помнить два правила:

- при считывании значений для переменных простого типа перед именем переменной ставится символ &;
- при считывании строки символ & не ставится, так как строковая переменная задается с помощью указателя.

В качестве примера приведем программу ввода с клавиатуры произвольного символа и воспроизведения на экране монитора как самого символа, так и его кода ASCII:

```
#include <stdio.h>
int main(void)
{char c;
scanf("%c",&c);
printf("\n Code of your symbol %c is equal %d",c, c);
return 0;}
```

Иногда во время выполнения программы наблюдается пропуск ввода в символьную переменную. Причина такой ошибки – коды символов, оставшиеся в буфере клавиатуры после предыдущего ввода данных, обычно это бывают управляющие коды. В этом случае функция scanf() не останавливает работу программы в ожидании ввода данных, а считывает в символьную переменную код, сохранившийся в буфере клавиатуры после предыдущего ввода. Чтобы избежать пропуска ввода, следует перед вводом в символьную переменную очистить буфер клавиатуры (входной поток stdin) с помощью функции fflush():

```
#include <stdio.h>
int main(void)
{char c,d;
printf("Введите первый символ->");
scanf("%c",&c);
printf ("Введите второй символ->");
fflush(stdin); /* Очистка буфера клавиатуры */
scanf("%c",&d);printf("Вы ввели: %c %c\n",c,d);}
```

Лабораторная работа №2

Тема: «Операторы»

Управляющие операторы

Оператор - конструкция языка программирования, обозначающая какие-либо действия в программе. Среди большого количества операторов можно выделить операторы, управляющие ходом вычислений. Такие операторы называют управляющими операторами (инструкциями). Другая группа операторов выполняет действия над данными, например, +, %, ++ и т. д. Такие операторы в языке С традиционно называют операциями.

В языке С можно выделить следующие группы операторов: условные; цикла; безусловного перехода; операторы-выражения; блоки (составные операторы).

К **условным** относятся операторы if и switch, к операторам **цикла** – for, while и do-while, к операторам **безусловного перехода** - break, continue, goto и return.

Операторы-выражения – это операторы, состоящие из операндов (переменных, констант, функций), знаков операций и круглых скобок, обозначающих порядок действий над операндами.

Блок или **составной оператор** представляет собой фрагмент текста программы, заключенный в фигурные скобки {}.

Условные операторы

Логические выражения

При выполнении многих операторов вначале анализируются данные (вычисляется некоторое логическое выражение), и в зависимости от полученного результата выбирается та или иная ветвь вычислительного процесса. Логическое выражение (ЛВ) – это выражение, принимающее одно из двух значений: истина или ложь.

В С (С89) нет специального типа для логических переменных и констант. Однако результат у ЛВ есть всегда. В языке С, если значение ЛВ равно нулю, то оно ложно, любое ненулевое ЛВ истинно. Простейшие ЛВ – любые выражения (в частности, переменная, арифметическое выражение). Отношения (следующий уровень ЛВ) позволяют сравнивать данные (таблица 2.1).

Отношение имеет структуру: V1 OP V2;

где V1, V2 – выражения, OP – знак операции отношения.

Операции <, <=, >, >= являются операциями одного уровня и выполняются в порядке их написания слева/направо; операции == и != имеют более низкий приоритет.

Следующий уровень ЛВ – логические выражения с использованием логических операций И(&&), ИЛИ(||), НЕ (!). В круглых скобках представлены обозначения этих операций в С.

Таблица 2.1.

Алгебраическая запись	Запись в С
<	<
≤	<=
>	>
≥	>=
=	==
≠	!=

Правила записи и результат обычные. Иерархия операций, если порядок их выполнения в ЛВ не полностью определен круглыми скобками, представлена в таблице 2.2.

Таблица 2.2.

Уровень иерархии	Операция
1	( )
2	!(НЕ)++ – –
3	*(умножение)/%
4	+ –
5	< <= > >=
6	== !=
7	&&(И)
8	||(ИЛИ)
9	= *= /= %= += –=

***Формы оператора* if**

В общем, виде синтаксис оператора if следующий:

```
if (ЛВ) OP1;
else OP2.
```

При истинном значении ЛВ выполняется оператор OP1, в противном случае – OP2, иногда используется и иная форма условного оператора

```
if (ЛВ) OP;
```

если ЛВ истинно - выполняется OP, в противном случае OP пропускается.

Если при ЛВ, имеющем значение «истина», необходимо выполнение нескольких операторов, то их надо заключить в фигурные скобки {}.

Пример,

```
#include <stdio.h>
int main(void)
{
        float x=2.;
        if (x==1) printf("\n x=%f",x);
        else printf("\n y=%f",x+4);
        return 0;}
```

Ответ: y=6.

В структуре операторов if действия OP1 и OP2 могут быть представлены любыми операторами, в том числе и условными. В этом случае, если порядок выполнения операций не полностью определен скобками {}, правило реализации следующее: к каждому if относится ближайший else.

***Оператор выбора* switch**

Оператор выбора switch (или переключатель) предназначен для выбора ветви вычислительного процесса исходя из значения управляющего выражения. Использование данного оператора целесообразно при сложном условном ветвлении.

Структура оператора следующая:

```
switch (выражение)
{
case константное выражение : оператор или группа операторов
break;
case константное выражение : оператор или группа операторов
break;
.....
default : оператор или группа операторов
}
```

Значение выражения оператора switch должно быть целочисленным. Это означает, что в выражении можно использовать переменные только целого или символьного типа. Результат вычисления выражения по очереди сравнивается с каждым из константных выражений. Если в какой-либо строке находится совпадение, управление передается на соответствующую метку case, и выполняется связанная с ней группа операторов. Выполнение продолжается до конца тела оператора switch или пока не встретится оператор break, который передает управление из тела switch оператору, следующему за закрывающей данную конструкцию фигурной скобкой.

Применение оператора break в контексте блока switch является обычным. Без него после выполнения варианта case, который соответствует значению управляющего выражения, оператор switch продолжит свою работу, при этом будут выполнены все последующие варианты case и ветка default.

Оператор, связанный с default, выполняется, если выражение не совпало ни с одним из константных выражений в case. Оператор default необязательно располагается в конце конструкции. Кроме того, он и сам необязателен. В этом случае при отсутствии совпадений не выполняется ни один оператор. Не допускается совпадение константных выражений.

Пример. По номеру дня недели, введенному с клавиатуры, вывести на экран название дня недели:

```
#include <stdio.h>
int main(void)
{
        char day;
        printf("Введите номер дня недели\n");
        scanf("%c",&day);
         switch (day)
        {
          case'1': printf("Понедельник\n");break;
          case'2': printf("Вторник\n");break;
          case'3': printf("Среда\n");break;
          case'4': printf("Четверг\n");break;
          case'5': printf("Пятница\n");break;
          case'6': printf("Суббота\n");break;
          case'7': printf("Воскресенье\n");break;
          default: printf("Ошибка ввода\n");}
        return 0;}
```

Ветвь default совсем не обязательно помещать последней в switch.

Операторы цикла

Операторы цикла служат для многократного повторения последовательности операторов до тех пор, пока выполняется некоторое условие. Цикл состоит из заголовка и тела цикла. Вход в цикл и выход из цикла осуществляются через заголовок цикла. В С существуют три вида циклов: for, while, do-while.

***Оператор* while**

Ключевое слово while позволяет выполнять оператор или группу операторов до тех пор, пока условие не перестанет быть истинным.

Синтаксис: while(ЛВ) ОР;
где ЛВ – логическое выражение, ОР – тело цикла (простой или составной оператор).
Итак:
если ЛВ – истинно, выполняется ОР, затем снова проверяется ЛВ и так далее;
если ЛВ – ложно, ОР пропускается, и управление передается на оператор, следующий за телом цикла.
Поскольку проверка ЛВ выполняется перед телом цикла, цикл while называют циклом с предусловием.
Если ЛВ изначально ложно, тело цикла while не выполнится ни разу.
Пример, Вычислить значение sin(x) для некоторого x с заданной точностью ε путем разложения в степенной ряд, при условии, что x>ε:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots = \sum_{k=0}^{\infty} \frac{x^{2k+1}(-1)^k}{(2k+1)!}$$

```
#include <stdio.h>
#include <math.h>
#define EPS 0.001
int main(void)
{
        float x,S=0,U;
        int i=0;
        printf("\n Input x→");
        scanf("%f",&x);
        U=x;
        while (fabs (U)>EPS) /*fabs – функция вычисления
        абсолютного значения, объявлена в math.h*/
        {
           S+=U;
           ++i;
           U*=(-x)*x;
           U/=2*i*(2*i+1);
        }
        printf("\n sin(%f)=%7.3f",x,S);
        return 0;}
```

***Оператор* for**

Оператор цикла for позволяет выполнять оператор или группу операторов заранее заданное количество раз.
Общая форма оператора

for (V1; ЛВ; V2) ОР;

где V1 – выражение, в котором производится присваивание переменной, называемой параметром цикла, начального значения (инициализация); ЛВ – условие, определяющее, следует ли в очередной раз выполнять оператор (тело цикла); V2 – выражение, в котором производится изменение переменной цикла (приращение); ОР – оператор или группа операторов.

Принцип работы:

- вычисляется V1;
- вычисляется ЛВ;
- если ЛВ истинно, выполняется ОР, затем вычисляется V2;
- проверка ЛВ → ОР → V2 и т. д.

Когда ЛВ становится ложным, осуществляется переход к оператору, следующему за ОР.
Если ОР – составной оператор, то его надо заключить в {}.
Если ЛВ сразу ложно, то ни ОР, ни V2 не выполняются ни разу.
Отметим две дополнительные возможности оператора for по сравнению с оператором while:

- возможность включения инициализирующего выражения V1, используемого один раз перед тем, как будет произведена оценка условия;
- возможность включения выражения V2, которое будет использоваться после каждой итерации оператора ОР.

Пример, Вычислить значения функции sin(x) в равноудаленных точках на интервале [0, 4π]. Количество расчетных точек вводится с клавиатуры:

```
#include <stdio.h>
#include <math.h>
int main(void)
{
        int n;
        float x, y;
        float x1=0, x2=4*M_PI;
        printf("Введите количество точек\n");
        scanf("%d",&n);
        x=x1;
        float step=(x2-x1)/(n-1);
        for (int i=1; i<=n; i++)
        {
           y = sin(x);
           printf("%3d %8.3f %8.3f\n",i,x,y);
           x+=step;}
         return 0;}
```

Следует иметь в виду, что все три выражения – V1, ЛВ, V2 являются необязательными. В качестве ОР может быть использован пустой оператор.
Возможны конструкции:

for(V1;;V2)OP (бесконечный цикл); for(;ЛВ;)OP и даже for(;;).

Допускается использование оператора for, когда имеется несколько начальных выражений вида V1, перечисленных через запятую, и несколько выражений вида V2, также перечисленных через запятую. Конструкция такого вида носит название – операция «запятая».

Оператор do-while

В цикле do-while проверка условия проводится после выполнения тела цикла:

do OP while(ЛВ)

Действие

- выполняется ОР;
- если ЛВ истинно, повторяется выполнение ОР, в противном случае осуществляется переход к оператору, следующему за while.

Если в цикле должно выполняться несколько операторов, они заключаются в {}. Данный цикл носит название цикла с постусловием, т. е. при любом ЛВ тело цикла выполняется хотя бы один раз.

С использованием ранее записанной формулы разложения вычислим y=sin(x) с точностью ε:

```
#include <stdio.h>
#include <math.h>
#define EPS 0.001
int main(void)
{
        float x,S=0,U;
        int i=0;
        printf("\n Input x→");
        scanf("%f",&x);
        U=x;
        do
        {
            S+=U;
            ++i;
            U*=(-x)*x;
            U/=2*i*(2*i+1);
        }
        while (fabs (U)>EPS);
        printf("\n sin(%6.3f)=%7.3f",x,S);
        return 0;}
```

Операторы перехода

В ряде случаев возникают ситуации, когда необходимо прервать выполнение блока операторов вне зависимости от каких-либо условий. Определены четыре оператора перехода: break, continue, return и goto. Операторы break и continue можно использовать в любом из операторов цикла. Заметим, что вход в тело цикла из оператора, расположенного вне этого цикла, невозможен. Возможны выход из цикла до его нормального завершения и обход части цикла при некоторых условиях. Оператор break, как отмечено ранее, можно также использовать в операторе switch. Операторы return и goto можно использовать в любом месте внутри функции.

***Оператор* break**

Оператор break применяется в двух случаях. Во-первых, в операторе switch с его помощью прерывается выполнение последовательности case. Во-вторых, оператор break используется для немедленного прекращения выполнения цикла без проверки его условия и передачи управления оператору, следующему после оператора цикла.

Например,

```
#include <stdio.h>
int main(void)
{       int num, count=0;
        for(;;count ++)
        {  printf("\n num→");
           scanf("%d",&num);
           if (num<0) break;
        }
        printf("\n count=%d",count);
        return 0;}
```

последовательно вводятся целые числа, и подсчитывается число введенных до первого отрицательного числа. После этого ввод чисел прекращается. Оператор break использован в условии оператора if для выхода из цикла.

***Оператор* continue**

Можно сказать, что оператор continue немного похож на оператор break. Оператор break вызывает прерывание цикла, а continue – прерывание текущей итерации цикла и переход к следующей итерации).

***Оператор* return**

Завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию. Управление передается в вызывающую функцию в точку, непосредственно следующую за вызовом. Синтаксис:

```
return (выражение);
```

Значение вычисленного в операторе return выражения возвращается в вызывающую функцию в качестве результата вызываемой функции. Если выражение опущено, то возвращаемое функцией значение не определено (функции типа void).

Если оператор return в вызываемой функции отсутствует, управление автоматически передается в вызывающую функцию после выполнения последнего оператора функции. Возвращаемое функцией значение в этом случае не определено.

Итак, оператор return используется в 2-х случаях:

- если надо немедленно выйти из функции;

если функция должна возвращать значение.

Например, *в первом случае:*

```
void print(char x) /*создание функции print*/
{
        if (x<32)
        {printf ("Это управляющий код \n");
        return;   /*return используется для выхода из функции*/}
        printf ("Введен символ %c \n",x);}
```

во втором случае:

```
int sum (int a,int b) /*создание функции sum*/
{return (a+b); /*return используется для возврата суммы */}
```

Применение оператора goto и меток

Оператор goto осуществляет безусловную передачу управления на метку в пределах текущей функции. Метка – это идентификатор с двоеточием, ставится перед оператором, которому надо передать управление. Синтаксис:

```
M: ...
...
goto M; /* M – метка*/
```

Необходимости в применении оператора goto практически нет. Он затрудняет понимание программы и нарушает ее структуру. Считается, что в программировании не существует ситуаций, в которых нельзя обойтись без оператора goto. В большинстве случаев его можно заменить оператором break или continue. Использование goto считается допустимым при организации выхода из глубоко вложенного цикла, хотя уже одно это (чрезмерная вложенность и неожиданный выход сразу из нескольких циклов) может свидетельствовать о плохой структуре программы. Как область действия goto, так и область действия метки ограничены текущей функцией.

Лабораторная работа №3

Тема: «Функции»

Определение и вызов функций

Функция - это совокупность объявлений и операторов, обычно предназначенная для решения определенной задачи. Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова. В любой программе на СИ должна быть функция с именем main (главная функция), именно с этой функции, в каком бы месте программы она не находилась, начинается выполнение программы.

При вызове функции ей при помощи аргументов (формальных параметров) могут быть переданы некоторые значения (фактические параметры), используемые во время выполнения функции. Функция может возвращать некоторое (одно) значение - результат выполнения функции, который при выполнении программы подставляется в точку вызова функции, где бы этот вызов ни встретился. Допускается также использовать функции не имеющие аргументов и функции, не возвращающие никаких значений.

С использованием функций в языке СИ связаны три понятия - *определение функции* (описание действий, выполняемых функцией), *объявление функции* (задание формы обращения к функции) и вызов функции. *Определение функции* задает тип возвращаемого значения, имя функции, типы и число формальных параметров, а также объявления переменных и операторы, называемые телом функции, и определяющие действие функции. В определении функции также может быть задан класс памяти.

Пример:

```
int rus (unsigned char r)
{ if (r>='A' && c<=' ')
  return 1;
  else
  return 0;}
```

Определения используемых функций могут следовать за определением функции main, быть перед ним, или находиться в другом файле. Для того, чтобы компилятор мог осуществить проверку соответствия типов передаваемых фактических параметров типам формальных параметров до вызова функции, нужно поместить объявление (прототип) функции.

Объявление функции имеет такой же вид, что и определение функции, с той лишь разницей, что тело функции отсутствует, и имена формальных параметров тоже могут быть опущены. Для функции, определенной в последнем примере, прототип может иметь вид : int rus (unsigned char r); или rus (unsigned char);

В программах на языке СИ широко используются библиотечные функции. Прототипы библиотечных функций находятся в специальных заголовочных файлах, поставляемых вместе с библиотеками в составе систем программирования, и включаются в программу с помощью директивы #include.

Если объявление функции не задано, то по умолчанию строится прототип функции на основе анализа первой ссылки на функцию, будь то вызов функции или определение. Однако такой прототип не всегда согласуется с последующим определением или вызовом функции. Рекомендуется всегда задавать прототип функции. Это позволит компилятору либо выдавать диагностические сообщения, при неправильном использовании функции, либо корректным образом регулировать несоответствие аргументов устанавливаемое при выполнении программы.

Например, функция rus из предыдущего примера может быть определена следующим образом:

```
int rus (r)
unsigned char r;
{       /*тело функции*/ }
```

В соответствии с синтаксисом языка СИ определение функции имеет следующую форму:

[спецификатор-класса-памяти] [спецификатор-типа] имя-функции
([список-формальных-параметров])
{ тело-функции }

Необязательный спецификатор-класса-памяти задает класс памяти функции, который может быть static или extern.

Спецификатор-типа функции задает тип возвращаемого значения и может задавать любой тип. Если спецификатор-типа не задан, то предполагается, что функция возвращает значение типа int. Функция не может возвращать массив или функцию, но может возвращать указатель на любой тип, в том числе и на массив и на функцию. Тип возвращаемого значения, задаваемый в определении функции, должен соответствовать типу в объявлении этой функции. Функция возвращает значение, если ее выполнение заканчивается оператором return, содержащим некоторое выражение. Указанное выражение вычисляется, преобразуется, если необходимо, к типу возвращаемого значения и возвращается в точку вызова функции в качестве результата. Если оператор return не содержит выражения или выполнение функции завершается после выполнения последнего ее оператора (без выполнения оператора return), то возвращаемое значение не определено. Для функций, не использующих возвращаемое значение, должен быть использован тип void, указывающий на отсутствие возвращаемого значения. Если

функция определена как функция, возвращающая некоторое значение, а в операторе return при выходе из нее отсутствует выражение, то поведение вызывающей функции после передачи ей управления может быть непредсказуемым.

Список-формальных-параметров - это последовательность объявлений формальных параметров, разделенная запятыми. Формальные параметры - это переменные, используемые внутри тела функции и получающие значение при вызове функции путем копирования в них значений соответствующих фактических параметров. Список-формальных-параметров может заканчиваться запятой (,) или запятой с многоточием (,...), это означает, что число аргументов функции переменно. Однако предполагается, что функция имеет, по крайней мере, столько обязательных аргументов, сколько формальных параметров задано перед последней запятой в списке параметров. Такой функции может быть передано большее число аргументов, но над дополнительными аргументами не проводится контроль типов. Если функция не использует параметров, то наличие круглых скобок обязательно, а вместо списка параметров рекомендуется указать слово void.

Порядок и типы формальных параметров должны быть одинаковыми в определении функции и во всех ее объявлениях. Типы фактических параметров при вызове функции должны быть совместимы с типами соответствующих формальных параметров. Тип формального параметра может быть любым основным типом, структурой, объединением, перечислением, указателем или массивом. Если тип формального параметра не указан, то этому параметру присваивается тип int.

Для формального параметра можно задавать класс памяти register, при этом для величин типа int спецификатор типа можно опустить.

Идентификаторы формальных параметров используются в теле функции в качестве ссылок на переданные значения. Эти идентификаторы не могут быть переопределены в блоке, образующем тело функции, но могут быть переопределены во внутреннем блоке внутри тела функции.

При передаче параметров в функцию, если необходимо, выполняются обычные арифметические преобразования для каждого формального параметра и каждого фактического параметра независимо. После преобразования формальный параметр не может быть короче чем int, т.е. объявление формального параметра с типом char равносильно его объявлению с типом int. А параметры, представляющие собой действительные числа, имеют тип double.

Преобразованный тип каждого формального параметра определяет, как интерпретируются аргументы, помещаемые при вызове функции в стек. Несоответствие типов фактических аргументов и формальных параметров может быть причиной неверной интерпретации.

Тело функции - это составной оператор, содержащий операторы, определяющие действие функции. Все переменные, объявленные в теле функции без указания класса памяти, имеют класс памяти auto, т.е. они являются локальными. При вызове функции локальным переменным отводится память в стеке и производится их инициализация. Управление передается первому оператору тела функции и начинается выполнение функции, которое продолжается до тех пор, пока не встретится оператор return или последний оператор тела функции. Управление при этом возвращается в точку,

следующую за точкой вызова, а локальные переменные становятся недоступными. При новом вызове функции для локальных переменных память распределяется вновь, и поэтому старые значения локальных переменных теряются.

Параметры функции передаются по значению и могут рассматриваться как локальные переменные, для которых выделяется память при вызове функции и производится инициализация значениями фактических параметров. При выходе из функции значения этих переменных теряются. Поскольку передача параметров происходит по значению, в теле функции нельзя изменить значения переменных в вызывающей функции, являющихся фактическими параметрами. Однако если в качестве параметра передать указатель на некоторую переменную, то используя операцию разадресации, можно изменить значение этой переменной.

Пример:

```
void change (int *x, int *y)
{ int k=*x;
*x=*y;
*y=k;
}
```

При вызове такой функции в качестве фактических параметров должны быть использованы не значения переменных, а их адреса

change (&a,&b);

Если требуется вызвать функцию до ее определения в рассматриваемом файле или определение функции находится в другом исходном файле, то вызов функции следует предварять объявлением этой функции. Объявление (прототип) функции имеет следующий формат:

[спецификатор-класса-памяти] [спецификатор-типа] имя-функции ([список-формальных-параметров]) [,список-имен-функций];

В отличие от определения функции, в прототипе за заголовком сразу же следует точка с запятой, а тело функции отсутствует. Если несколько разных функций возвращают значения одинакового типа и имеют одинаковые списки формальных параметров, то эти функции можно объявить в одном прототипе, указав имя одной из функций в качестве имени-функции, а все другие поместить в список-имен-функций, причем каждая функция должна сопровождаться списком формальных параметров. Правила использования остальных элементов формата такие же, как при определении функции. Имена формальных параметров при объявлении функции можно не указывать, а если они указаны, то их область действия распространяется только до конца объявления.

Прототип - это явное объявление функции, которое предшествует определению функции. Тип возвращаемого значения при объявлении функции должен соответствовать типу возвращаемого значения в определении функции.

Если прототип функции не задан, а встретился вызов функции, то строится неявный прототип из анализа формы вызова функции. Тип возвращаемого значения создаваемого прототипа int, а список типов и числа параметров функции формируется на основании типов и числа фактических параметров, используемых при данном вызове.

Таким образом, прототип функции необходимо задавать в следующих случаях: функция возвращает значение типа, отличного от int; требуется проинициализировать некоторый указатель на функцию до того, как эта функция будет определена.

Если прототип задан с классом памяти static, то и определение функции должно иметь класс памяти static. Если спецификатор класса памяти не указан, то подразумевается класс памяти extern.

Вызов функции имеет следующий формат:

адресное-выражение ([список-выражений])

Поскольку синтаксически имя функции является адресом начала тела функции, в качестве обращения к функции может быть использовано адресное-выражение (в том числе и имя функции или разадресация указателя на функцию), имеющее значение адреса функции.

Список-выражений представляет собой список фактических параметров, передаваемых в функцию. Этот список может быть и пустым, но наличие круглых скобок обязательно.

Фактический параметр может быть величиной любого основного типа, структурой, объединением, перечислением или указателем на объект любого типа. Массив и функция не могут быть использованы в качестве фактических параметров, но можно использовать указатели на эти объекты.

Выполнение вызова функции происходит следующим образом:

1. Вычисляются выражения в списке выражений и подвергаются обычным арифметическим преобразованиям. Затем, если известен прототип функции, тип полученного фактического аргумента сравнивается с типом соответствующего формального параметра. Если они не совпадают, то либо производится преобразование типов, либо формируется сообщение об ошибке. Число выражений в списке выражений должно совпадать с числом формальных параметров, если только функция не имеет переменного числа параметров. В последнем случае проверке подлежат только обязательные параметры. Если в прототипе функции указано, что ей не требуются параметры, а при вызове они указаны, формируется сообщение об ошибке.
2. Происходит присваивание значений фактических параметров соответствующим формальным параметрам.
3. Управление передается на первый оператор функции.
4. Выполнение оператора return в теле функции возвращает управление и, возможно, значение в вызывающую функцию. При отсутствии оператора return управление возвращается после выполнения последнего оператора тела функции, а возвращаемое значение не определено.

Адресное выражение, стоящее перед скобками определяет адрес вызываемой функции. Это значит, что функция может быть вызвана через указатель на функцию.

Пример: int (*fun)(int x, int *y);

Здесь объявлена переменная fun как указатель на функцию с двумя параметрами: типа int и указателем на int. Сама функция должна возвращать значение типа int. Круглые скобки, содержащие имя указателя fun и признак указателя *, обязательны, иначе запись

int *fun (intx,int *y);

будет интерпретироваться как объявление функции fun возвращающей указатель на int.

Вызов функции возможен только после инициализации значения указателя fun и имеет вид: (*fun)(i,&j);

В этом выражении для получения адреса функции, на которую ссылается указатель fun, используется операция разадресации * .

Указатель на функцию может быть передан в качестве параметра функции. При этом разадресация происходит во время вызова функции, на которую ссылается указатель на функцию. Присвоить значение указателю на функцию можно в операторе присваивания, употребив имя функции без списка параметров.

Пример:

```
double (*fun1)(int x, int y);
double fun2(int k, int l);
fun1=fun2;        /* инициализация указателя на функцию */
(*fun1)(2,7);     /* обращение к функции*/
```

Любая функция в программе на языке СИ может быть вызвана рекурсивно, т.е. она может вызывать саму себя. Компилятор допускает любое число рекурсивных вызовов. При каждом вызове для формальных параметров и переменных с классом памяти auto и register выделяется новая область памяти, так что их значения из предыдущих вызовов не теряются, но в каждый момент времени доступны только значения текущего вызова.

Переменные, объявленные с классом памяти static, не требуют выделения новой области памяти при каждом рекурсивном вызове функции, и их значения доступны в течение всего времени выполнения программы.

Классический пример рекурсии - это математическое определение факториала n! :

n! =1 при n=0;

n*(n-1)! при n>1 .

Функция, вычисляющая факториал, будет иметь следующий вид:

```
long fakt(int n)
{ return ( (n==1) ? 1 : n*fakt(n-1) );}
```

Хотя компилятор языка СИ не ограничивает число рекурсивных вызовов функций, это число ограничивается ресурсом памяти компьютера и при слишком большом числе рекурсивных вызовов может произойти переполнение стека.

Вызов функции с переменным числом параметров

При вызове функции с переменным числом параметров в вызове этой функции задается любое требуемое число аргументов. В объявлении и определении такой функции переменное число аргументов задается многоточием в конце списка формальных параметров или списка типов аргументов.

Все аргументы, заданные в вызове функции, размещаются в стеке. Количество формальных параметров, объявленных для функции, определяется числом аргументов, которые берутся из стека и присваиваются формальным параметрам. Программист отвечает за правильность выбора дополнительных аргументов из стека и определение числа аргументов, находящихся в стеке. Программист может разрабатывать свои функции с переменным числом параметров. Для обеспечения удобного способа доступа к аргументам функции с переменным числом параметров имеются три макроопределения (макросы) va_start, va_arg, va_end, находящиеся в заголовочном файле stdarg.h. Эти макросы указывают на то, что функция, разработанная пользователем, имеет некоторое число обязательных аргументов, за которыми следует переменное число необязательных аргументов. Обязательные аргументы доступны через свои имена как при вызове обычной функции. Для извлечения необязательных аргументов используются макросы va_start, va_arg, va_end в следующем порядке.

Макрос va_start предназначен для установки аргумента arg_ptr на начало списка необязательных параметров и имеет вид функции с двумя параметрами:

void va_start(arg_ptr,prav_param);

Параметр prav_param должен быть последним обязательным параметром вызываемой функции, а указатель arg_prt должен быть объявлен с предопределением в списке переменных типа va_list в виде: va_list arg_ptr;

Макрос va_start должен быть использован до первого использования макроса va_arg.

Макрокоманда va_arg обеспечивает доступ к текущему параметру вызываемой функции и тоже имеет вид функции с двумя параметрами type_arg va_arg(arg_ptr,type);

Эта макрокоманда извлекает значение типа type по адресу, заданному указателем arg_ptr, увеличивает значение указателя arg_ptr на длину использованного параметра (длина type) и таким образом параметр arg_ptr будет указывать на следующий параметр вызываемой функции. Макрокоманда va_arg используется столько раз, сколько необходимо для извлечения всех параметров вызываемой функции.

Макрос va_end используется по окончании обработки всех параметров функции и устанавливает указатель списка необязательных параметров на ноль (NULL).

Рассмотрим применение этих макросов для обработки параметров функции, вычисляющей среднее значение произвольной последовательности целых чисел. Поскольку функция имеет переменное число параметров, будем считать концом списка значение, равное -1. Поскольку в списке должен быть хотя бы один элемент, у функции будет один обязательный параметр.

Пример:

```
        #include
        int main()
        { int n;
        int sred_znach(int,...);
        n=sred_znach(2,3,4,-1);
/* вызов с четырьмя параметрами */
        printf("n=%d",n);
        n=sred_znach(5,6,7,8,9,-1);
/* вызов с шестью параметрами  */
        printf("n=%d",n);
        return (0);
        }
        int sred_znach(int x,...);
        {
        int i=0,  j=0,  sum=0;
        va_list uk_arg;
va_start(uk_arg,x);  /*  установка указателя uk_arg на  */
                     /*  первый необязятельный параметр */
    if (x!=-1) sum=x;    /*  проверка на пустоту списка     */
    else return (0);
    j++;
    while ( (i=va_arg(uk_arg,int))!=-1)
                        /* выборка очередного   */
```

```
{                    /* параметра и проверка */
 sum+=i;             /* на конец списка     */
 j++;
}
va_end(uk_arg);      /* закрытие списка параметров  */
return (sum/j);}
```

Передача параметров функции main

Функция main, с которой начинается выполнение СИ-программы, может быть определена с параметрами, которые передаются из внешнего окружения, например, из командной строки. Во внешнем окружении действуют свои правила представления данных, а точнее, все данные представляются в виде строк символов. Для передачи этих строк в функцию main используются два параметра, первый параметр служит для передачи числа передаваемых строк, второй для передачи самих строк. Общепринятые (но необязательные) имена этих параметров argc и argv. Параметр argc имеет тип int, его значение формируется из анализа командной строки и равно количеству слов в командной строке, включая и имя вызываемой программы (под словом понимается любой текст, не содержащий символа пробел). Параметр argv – это массив указателей на строки, каждая из которых содержит одно слово из командной строки. Если слово должно содержать символ пробел, то при записи его в командную строку оно должно быть заключено в кавычки.

Функция main может иметь и третий параметр, который принято называть argp, и который служит для передачи в функцию main параметров операционной системы (среды) в которой выполняется СИ-программа.

Заголовок функции main имеет вид:

```
int main (int argc, char *argv[], char *argp[])
```

Операционная система поддерживает передачу значений для параметров argc, argv, argp, а на пользователе лежит ответственность за передачу и использование фактических аргументов функции main.

Следующий пример представляет программу печати фактических аргументов, передаваемых в функцию main из операционной системы и параметров операционной системы.

```
Пример:
int main ( int argc, char *argv[], char *argp[])
{ int i=0;
  printf ("\n Имя программы %s", argv[0]);
  for   (i=1; i>=argc; i++)
  printf ("\n аргумент %d равен %s", argv[i]);
  printf ("\n   Параметры операционной системы:");
  while  (*argp)
   { printf ("\n %s",*argp);
     argp++;
   }
  return (0);}
```

Рекурсия

Функция называется рекурсивной, если во время ее обработки возникает ее повторный вызов, либо непосредственно, либо косвенно, путем цепочки вызовов других функций.

Прямой (непосредственной) рекурсией является вызов функции внутри тела этой функции.

```
int a()
{.....a().....}
```

Косвенной рекурсией является рекурсия, осуществляющая рекурсивный вызов функции посредством цепочки вызова других функций. Все функции, входящие в цепочку, тоже считаются рекурсивными.

Например:

```
a(){.....b().....}
b(){.....c().....}
c(){.....a().....}.
```

Все функции a, b, c являются рекурсивными, так как при вызове одной из них осуществляется вызов других и самой себя.

Рассмотрим задачу о Ханойских башнях. Имеются три стержня с номерами 1,2,3. На стержень 1 надето n дисков различного диаметра так, что они образуют пирамиду (см. рисунок 3.1). Написать программу для печати последовательности перемещений дисков со стержня на стержень, необходимых для переноса пирамиды со стержня 1 на стержень 3 при использовании стержня 2 в качестве вспомогательного. При этом за одно перемещение должен переноситься только один диск, и диск большего диаметра не должен помещаться на диск меньшего диаметра. Доказано, что для n дисков минимальное число необходимых перемещений равно 2^n-1.

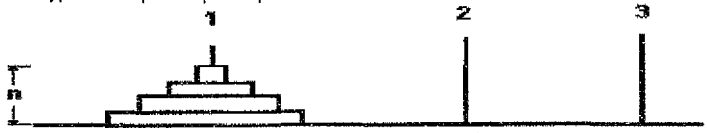

Рисунок 3.1 – Задача о Ханойских башнях

Для решения простейшего случая задачи, когда пирамида состоит только из одного диска, необходимо выполнить одно действие - перенести диск со стержня i на стержень j, что очевидно (этот перенос обозначается i -> j). Общий случай задачи изображен на рисунке, когда требуется перенести n дисков со стержня i на стержень j, считая стержень w вспомогательным. Сначала следует перенести n-1 диск со стержня i на стержень w при вспомогательном стержне j, затем перенести один диск со стержня i на стержень j и, наконец, перенести n-1 диск из w на стержень j, используя вспомогательный стержень i. Итак, задача о переносе n дисков сводится к двум задачам о переносе n-1 диска и одной простейшей задаче. Схематически это можно записать так: T(n,i,j,w) = T(n-1,i,w,j), T(1,i,j,w), T(n-1,w,j,i).

Ниже приведена программа, которая вводит число n и печатает список перемещений, решающая задачу о Ханойских башнях при количестве дисков n. Используется внутренняя рекурсивная процедура tn(n, i, j, w), печатающая перемещения, необходимые для переноса n дисков со стержня i на стержень j с использованием вспомогательного стержня w при {i, j, w} = {1,3,2}.

```
/*              ханойские башни              */
#include
main()                          /*  вызывающая  */
{  void tn(int, int, int, int);     /*  функция     */
   int n;
   scanf(" %d",&n);
   tn(n,1,2,3);
}
void tn(int n, int i, int j, int w)  /*  рекурсивная  */
{  if (n>1)                     /*  функция     */
   {  tn (n-1,i,w,j);
      tn (1,i,j,w);
      tn (n-1,w,j,i);
   }
   else printf(" \n %d -> %d",i,j);
   return ;
}
```

Последовательность вызовов процедуры tn при m=3 иллюстрируется древовидной структурой на рисунок 3.2. Каждый раз при вызове процедуры tn под параметры n, i, j, w выделяется память и запоминается место возврата. При возврате из процедуры tn память, выделенная под параметры n, i, j, w, освобождается и становится доступной память, выделенная под параметры n, i, j, w предыдущим вызовом, а управление передается в место возврата.

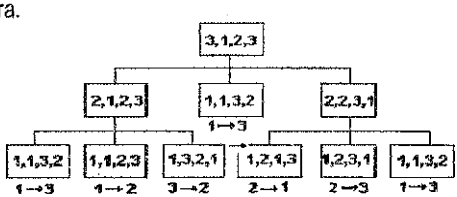

Рисунок 3.2 – Последовательность вызовов процедуры tn

Лабораторная работа №4

Тема: «Массивы. Статические массивы. Динамические массивы»

Общие сведения о массивах

Несколько переменных одного типа можно объединить под одним именем. Такая переменная будет представлять собой массив. Массив – это тип данных, представляющий собой ограниченный набор упорядоченных элементов одного и того же типа, имеющих одно и то же имя. Элементом массива является переменная. Количество элементов массива определено заранее при объявлении массива. Все элементы упорядочены – каждому присвоен порядковый номер, который называется индексом. Доступ к конкретному элементу массива осуществляется с помощью индекса. В языке С все массивы располагаются в отдельной непрерывной области памяти. Первый элемент массива имеет наименьший адрес, а последний – наибольший. Элементы массива могут

быть как простыми переменными, так и составными. Элемент массива может иметь несколько индексов. Количество индексов переменной определяет размерность массива.

Одномерные массивы

Общая форма объявления одномерного массива имеет следующий вид:

<класс> тип имя [размер],

где класс – необязательный элемент, определяющий класс памяти (extern, static, register);

тип – базовый тип элемента массива;

имя – идентификатор массива;

размер – количество элементов в массиве.

Доступ к элементу массива осуществляется с помощью имени массива и индекса. Индекс элемента массива помещается в квадратных скобках после имени. Нижнее значение индекса всегда нуль. Таким образом, элементами массива, состоящего из N элементов, являются переменные с индексами a[0],a[1],...,a[N–1]. В качестве N в описании должна стоять целая положительная константа.

Пример Найти $S = \sum_{i=1}^{N} x_i$, N = 15

```
# include <stdio.h>
# define N 15
int main()
{       float x[N],s;
        int i;
        for (i=0,s=0;i<N;++i)
        {printf("\n Input x[%d]",i);
          scanf("%f",&x[i]);
          s+=x[i];
        }
        printf("\n S=%f",s);
        return 0;}
```

Объем памяти, необходимый для хранения массива, определяется типом элемента массива и количеством элементов, входящих в массив. Для одномерного массива требуемый объем памяти вычисляется следующим образом:

Объем памяти в байтах = sizeof(базовый тип элемента)*длина массива.

Во время выполнения программы не проверяется ни соблюдение границ массива, ни его содержимое. Задача проверки корректности выполнения программы возложена на программиста. В языке С массивы при объявлении можно инициализировать:

<класс> тип имя [размер] = {список значений};

Список значений представляет собой список констант, перечисленных в фигурных скобках через запятую. Типы констант должны быть совместимы с типом массива. После закрывающей фигурной скобки точка с запятой обязательна:

```
static float x[5]={7,5,0,–3.2,0,4};
```

Если в списке инициализации задать количество элементов меньше, чем задано в объявлении массива, то остальные элементы принимаются равными нулю. Если в

списке инициализации задать количество элементов больше, чем задано в объявлении массива, это вызовет ошибку при компиляции программы.

При инициализации допустима и следующая форма объявления:

<класс> тип имя [] = {$a_0, a_1, \ldots, a_{M-1}$};

Компилятор сам создает массив из M элементов, присваивая им значения $a_0, a_1, \ldots, a_{M-1}$.

Двумерные массивы

Простейшая форма многомерного массива – двумерный массив, т. е. массив одномерных массивов.

Описание двумерного массива выглядит следующим образом:

<класс> тип имя [N1] [N2];

где N1,N2 – количество строк и столбцов.

Массив в памяти располагается по строкам:

ID[0][0],...,ID[0][N2–1],

...

ID[N1–[0],...,ID[N1–1][N2–1], где ID – имя массива.

Как и в случае одномерных массивов, возможна инициализация двумерных массивов:

```
int a[2][3]={{1,–2,7},{2,–3,9}};
```

Пример программы

```
# include <stdio.h>
int main()
{       int a[2][3]={{6,–1},{–3,2}};
        int i,j;
        for (i=0;i<2;++i)
        for (j=0;j<3;++j)
        printf("\n a[%d,%d]=%d",i,j,a[i][j]);}
```

Ответ: a[0,0]=6, a[0,1]=–1, a[0,2] = 0, a[1,0]=–3, a[1,1]=2, a[1,2]=0.

Массивы и указатели

В языке С массивы и указатели тесно связаны друг с другом. Например, когда объявляется массив в виде int a[25], то при этом не только выделяется память для 25 элементов массива, но и формируется указатель с именем a, значение которого равно адресу первого по счету (нулевого) элемента массива. Доступ к элементам массива может осуществляться через указатель с именем a. С точки зрения синтаксиса языка указатель a является константой, значение которой можно использовать в выражениях, но изменить это значение нельзя. Поскольку имя массива является указателем-константой, допустимо, например, такое присваивание:

```
int a[25];
int *ptr;
ptr=a;
```

В этом примере в переменную-указатель ptr записывается адрес начала массива a, т. е. адрес первого элемента массива. Также справедливы следующие соотношения: например, имеется массив a[N], тогда истинными будут следующие сравнения:

```
a==&a[0];
*a==a[0].
```

Указатели можно увеличивать или уменьшать на целое число:

```
ptr=a+1;
```

Теперь указатель ptr будет указывать на второй элемент массива a, что эквивалентно &a[1].

При увеличении указателя на единицу адрес, который он представляет, увеличивается на размер объекта связанного с ним типа, например:

```
int a[25];
int *ptr=a;
ptr+=3;
```

Первоначально указатель ptr указывал на начало массива a. После прибавления к переменной ptr числа 3 значение указателя увеличилось на 3*sizeof(int), а указатель ptr теперь будет указывать на четвертый элемент массива a. Указатель можно индексировать точно так же, как и массив. На самом деле компилятор преобразует индексацию в арифметику указателей, например, ptr[3]=10 представляется как *(ptr+3)=10.

К указателям типа void арифметические операции применять нельзя, так как им не ставится в соответствие размер области памяти.

Для доступа к элементам массива существует два различных способа. *Первый способ* связан с использованием обычных индексных выражений в квадратных скобках, например, a[7]=3 или a[i+2]=5. При таком способе доступа записываются два выражения, причем второе выражение заключается в квадратные скобки. Первое из этих выражений должно быть указателем, а второе – выражением целого типа. Указатель, используемый в индексном выражении, необязательно должен быть константой, указывающей на какой-либо массив, это может быть и переменная-указатель. В частности, после выполнения присваивания ptr=a доступ к седьмому элементу массива можно получить как с помощью константы-указателя a в форме a[7], так и переменной-указателя ptr в форме ptr[7]. *Второй способ* доступа к элементам массива связан с использованием адресных выражений и операции косвенной адресации в форме *(a+3)=10 или *(a+i+2)=5.При реализации на компьютере первый способ приводится ко второму, т. е. индексное выражение приводится к адресному. Для приведенных примеров обращение к элементу массива a[3] преобразуется в *(a+3).

Для доступа к начальному элементу массива, т. е. к элементу с нулевым индексом, можно использовать просто значение указателя a или ptr, поэтому любое из присваиваний

```
*a=2;
a[0]=2;
*(a+0)=2;
*ptr=2;
ptr[0]=2;
*(ptr+0)=2;
```

присваивает начальному элементу массива значение 2.

Многомерные массивы в языке С – это массивы массивов, т. е. массивы, элементами которых, в свою очередь, являются массивы. При объявлении таких массивов в памяти компьютера создается несколько различных объектов. Например, при выполнении объявления двумерного массива int a2[4][3] в программе создается указатель a2, который определяет в памяти местоположение первого элемента массива и, кроме того, является указателем на массив из четырех указателей. Каждый из этих четырех указателей содержит адрес одномерного массива, представляющего собой строку двумерного массива и состоящего из трех элементов типа int, и позволяет обратиться к соответствующей строке массива.

Таким образом, объявление a2[4][3] порождает в программе три разных объекта: указатель с идентификатором a2, безымянный массив из четырех указателей и безымянный массив из двенадцати чисел типа int. Для доступа к безымянным массивам используются адресные выражения с указателем a2. Доступ к элементам массива указателей осуществляется с указанием одного индексного выражения в форме a2[2] или *(a2+2). Для доступа к элементам двумерного массива чисел типа int должны быть использованы два индексных выражения в форме a2[1][2] или эквивалентных ей *(*(a2+1)+2) и (*(a2+1))[2]. Следует учитывать, что с точки зрения синтаксиса языка С указатель a2 и указатели a2[0], a2[1], a2[2], a2[3] являются константами, и их значения нельзя изменять во время выполнения программы.

При размещении элементов многомерных массивов они распюлагаются в памяти подряд по строкам, т. е. быстрее всего изменяется последний индекс, а медленнее – первый. Такой порядок дает возможность обращаться к любому элементу многомерного массива, используя адрес его начального элемента и только одно индексное выражение.

Например, обращение к элементу a2[1][2] можно осуществить при помощи указателя ptr2, объявленного в форме int *ptr2=a2[0], как обращение ptr2[1×3+2] (здесь 1 и 2 – это индексы используемого элемента, а 3 – число элементов в строке) или как ptr2[5]. Заметим, что внешне похожее обращение a2[6] выполнить невозможно, так как указателя с индексом 6 не существует.

Массивы и функции

Использование указателей в качестве формальных параметров функции дает возможность передавать из вызывающей подпрограммы в функцию не само значение или массив значений, а адрес переменной (адреса фактических аргументов). При работе с массивами не требуется использовать в качестве формальных аргументов массив. Это может быть указатель того же типа, что и тип элемента массива. В подпрограмму в этом случае можно передавать адрес начального элемента массива.

Пример:Задан одномерный массив из N элементов. Найти значение максимального элемента массива. Поиск максимума оформить в виде функции max: b– указательна целое, k–количество элементов в массиве.

```
# include <stdio.h>
# define N 3
int max(int k,int* b)
{       int i,m1;
        m1=*b; //*b – значение 1-го эл-та массива (с индексом 0)
        for(i=1;i<k;i++)
        {
          b++;//Переход к следующему элементу массива
          if (m1<*b)m1=*b; //*b – значение текущего эл-та массива
        }
        return(m1);
}
int main()
{
        static int A[N]={1,7,3};
        printf("\n max=%d",max(N,&A[0]));
        return 0;
}
```

Рассмотрим теперь пример программы с двумерным массивом. Дан двумерный массив, состоящий из трех строк и четырех столбцов. Найти максимальный элемент каждой строки, используя ранее созданную функцию max:

```
int main()
{
        static int B[3][4]={3,5,1,6,8,3,7,3,2,6,9,3};
        printf("\n 1 row: max=%d",max(4,B[0]));
        printf("\n 2 row: max=%d",max(4,B[1]));
        printf("\n 3 row: max=%d",max(4,B[2]));
        return 0;
}
```

При вызове функции max использовались указатели на начало строк массива, каждая строка массива содержит 4 значения.

Результат работы программы:

1 row: max=6
2 row: max=8
3 row: max=9.

Рассмотрим передачу двумерного массива в функцию. В качестве параметров функции передается адрес начала массива и количество элементов в строке массива, соответствующее объявлению массива в вызывающей функции (int A2[][N]), а также два целых числа m и n, имеющие смысл фактического количества строк и столбцов, используемых в массиве. Они не должны превышать значения количества строк и столбцов соответственно, заданные при объявлении массива.

В примере используются следующие функции: ввод двумерного массива input_mas(), вывод на экран двумерного массива в виде таблицы output_mas(), вычисление суммы элементов массива, имеющих одинаковые номера столбцов и строк int diagonal():

```
#include <stdio.h>
#define N 4
#define M 3
void input_mas(int A2[][N],int m,int n);
....
void main()
{
        int mas[M][N];
        input_mas(mas,M,N);
        ....
        return 0;
}
void input_mas(int A2[][N],int m,int n)
{
        int i,j;
        printf("Wwedite chisla->");
        for(i=0;i<M;i++)
        for(j=0;j<N;j++)
        scanf("%d",&A2[i][j]);}
```

Методы доступа к элементам массивов

В языке СИ между указателями и массивами существует тесная связь. Например, когда объявляется массив в виде int array[25], то этим определяется не только выделение памяти для двадцати пяти элементов массива, но и для указателя с именем array, значение которого равно адресу первого по счету (нулевого) элемента массива, т.е. сам массив остается безымянным, а доступ к элементам массива осуществляется через указатель с именем array. С точки зрения синтаксиса языка указатель array является константой, значение которой можно использовать в выражениях, но изменить это значение нельзя. Поскольку имя массива является указателем допустимо, например, такое присваивание:

```
int array[25];
int *ptr;
ptr=array;
```

Здесь указатель ptr устанавливается на адрес первого элемента массива, причем присваивание ptr=array можно записать в эквивалентной форме ptr=&array[0].

Для доступа к элементам массива существует два различных способа. *Первый способ* связан с использованием обычных индексных выражений в квадратных скобках, например, array[16]=3 или array[i+2]=7. При таком способе доступа записываются два выражения, причем второе выражение заключается в квадратные скобки. Одно из этих выражений должно быть указателем, а второе - выражением целого типа. Последовательность записи этих выражений может быть любой, но в квадратных скобках записывается выражение следующее вторым. Поэтому записи array[16] и 16[array] будут эквивалентными и обозначают элемент массива с номером шестнадцать. Указатель, используемый в индексном выражении, необязательно должен быть константой, указывающей на какой-либо массив, это может быть и переменная. В частности после выполнения присваивания ptr=array доступ к шестнадцатому элементу массива можно получить с помощью указателя ptr в форме ptr[16] или 16[ptr]. *Второй способ* доступа к элементам массива связан с использованием адресных выражений и операции разадресации в форме *(array+16)=3 или *(array+i+2)=7. При таком способе доступа адресное выражение, равное адресу шестнадцатого элемента массива, тоже может быть записано разными способами *(array+16) или *(16+array).

При реализации на компьютере первый способ приводится ко второму, т.е. индексное выражение преобразуется к адресному. Для приведенных примеров array[16] и 16[array] преобразуются в *(array+16). Для доступа к начальному элементу массива (т.е. к элементу с нулевым индексом) можно использовать просто значение указателя array или ptr. Любое из присваиваний

```
*array = 2;
array[0] = 2;
*(array+0) = 2;
*ptr = 2;
```

присваивает начальному элементу массива значение 2, но быстрее всего выполнятся присваивания *array=2 и *ptr=2, так как в них не требуется выполнять операции сложения.

Указатели на многомерные массивы

Указатели на многомерные массивы в языке СИ - это массивы массивов, т.е. такие массивы, элементами которых являются массивы. При объявлении таких массивов в памяти компьютера создается несколько различных объектов. Например, при выполнении объявления двумерного массива int arr2[4][3] в памяти выделяется участок для хранения значения переменной arr, которая является указателем на массив из четырех указателей. Для этого массива из четырех указателей тоже выделяется память. Каждый из этих четырех указателей содержит адрес массива из трех элементов типа int, и, следовательно, в памяти компьютера выделяется четыре участка для хранения четырех массивов чисел типа int, каждый из которых состоит из трех элементов. Такое выделение памяти показано на схеме на рисунке 4.1.

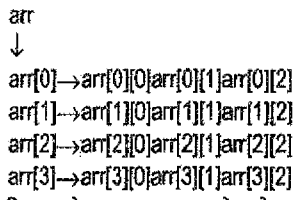

Рисунок 4.1 – Распределение памяти для двумерного массива

Таким образом, объявление arr2[4][3] порождает в программе три разных объекта: указатель с идентификатором arr, безымянный массив из четырех указателей и безымянный массив из двенадцати чисел типа int. Для доступа к безымянным массивам используются адресные выражения с указателем arr. Доступ к элементам массива указателей осуществляется с указанием одного индексного выражения в форме arr2[2] или *(arr2+2). Для доступа к элементам двумерного массива чисел типа int должны быть использованы два индексных выражения в форме arr2[1][2] или эквивалентных ей *(*(arr2+1)+2) и (*(arr2+1))[2]. Следует учитывать, что с точки зрения синтаксиса языка СИ указатель arr и указатели arr[0], arr[1], arr[2], arr[3] являются константами, и их значения нельзя изменять во время выполнения программы.

При размещении элементов многомерных массивов они располагаются в памяти подряд по строкам, т.е. быстрее всего изменяется последний индекс, а медленнее - первый. Такой порядок дает возможность обращаться к любому элементу многомерного массива, используя адрес его начального элемента и только одно индексное выражение.

Например, обращение к элементу arr2[1][2] можно осуществить с помощью указателя ptr2, объявленного в форме int *ptr2=arr2[0] как обращение ptr2[1*4+2] (здесь 1 и 2 – это индексы используемого элемента, а 4 – это число элементов в строке) или как ptr2[6]. Заметим, что внешне похожее обращение arr2[6] выполнить невозможно, так как указателя с индексом 6 не существует.

Операции с указателями

Над указателями можно выполнять унарные операции: инкремент и декремент. При выполнении операций ++ и -- значение указателя увеличивается или уменьшается на длину типа, на который ссылается используемый указатель.

Пример:

```
int *ptr,
a[10];
ptr=&a[5];
ptr++;    /* равно адресу элемента a[6] */
ptr--;    /* равно адресу элемента a[5] */
```

В бинарных операциях сложения и вычитания могут участвовать указатель и величина типа int. При этом результатом операции будет указатель на исходный тип, а его значение будет на указанное число элементов больше или меньше исходного.

В операции вычитания могут участвовать два указателя на один и тот же тип. Результат такой операции имеет тип int и равен числу элементов исходного типа между уменьшаемым и вычитаемым, причем если первый адрес младше, то результат имеет отрицательное значение. Пример:

```
int *ptr1, *ptr2, a[10];
int i;
  ptr1=a+4;
  ptr2=a+9;
  i=ptr1-ptr2; /*  равно 5  */
  i=ptr2-ptr1; /*  равно -5  */
```

Значения двух указателей на одинаковые типы можно сравнивать в операциях ==, !=, <, <=, >, >= при этом значения указателей рассматриваются просто как целые числа, а результат сравнения равен 0 (ложь) или 1 (истина).

Пример:

```
int *ptr1, *ptr2, a[10];
  ptr1=a+5;
  ptr2=a+7;
  if (prt1>ptr2) a[3]=4;
```

В данном примере значение ptr1 меньше значения ptr2, и поэтому оператор a[3]=4 не будет выполнен.

Массивы указателей

В языке СИ элементы массивов могут иметь любой тип и, в частности, могут быть указателями на любой тип. Рассмотрим несколько примеров с использованием указателей. Следующие объявления переменных

```
int a[]={10,11,12,13,14,};
int *p[]={a, a+1, a+2, a+2, a+3, a+4};
int **pp=p;
```

порождают программные объекты, представленные на схеме на рисунке 4.2.

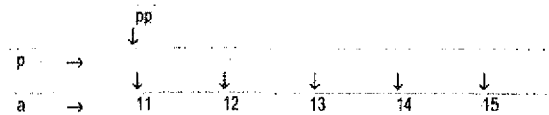

Рисунок 4.2 – Схема размещения переменных при объявлении

При выполнении операции pp-p получим нулевое значение, так как ссылки pp и p равны и указывают на начальный элемент массива указателей, связанного с указателем p (на элемент p[0]). После выполнения операции pp+=2 схема изменится и примет вид, изображенный на рисунок 4.3

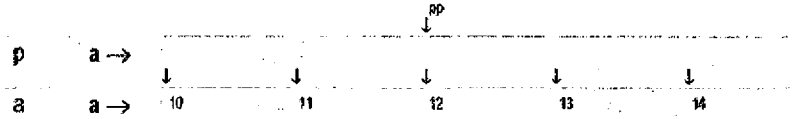

Рисунок 4.3 – Схема размещения переменных после выполнения операции pp+=2

Результатом выполнения вычитания pp-p будет 2, так как значение pp есть адрес третьего элемента массива p. Ссылка *pp-a тоже дает значение 2, так как обращение *pp есть адрес третьего элемента массива a, а обращение a есть адрес начального элемента массива a. При обращении с помощью ссылки **pp получим 12 - это значение третьего элемента массива a. Ссылка *pp++ даст значение четвертого элемента массива p, т.е. адрес четвертого элемента массива a. Если считать, что pp=p, то обращение *++pp это значение первого элемента массива a (т.е. значение 11), операция ++*pp изменит содержимое указателя p[0] таким образом, что он станет равным значению адреса элемента a[1]. Сложные обращения раскрываются изнутри. Например, обращение *(++(*pp)) можно разбить на следующие действия: *pp дает значение начального элемента массива p[0], далее это значение инкрементируется ++(*p) в результате чего указатель p[0] станет равен значению адреса элемента a[1], и последнее действие – это выборка значения по полученному адресу, т.е. значение 11. Рассмотрим теперь пример с многомерным массивом и указателями. Следующие объявления переменных

```
int a[3][3]={ { 11,12,13 },{ 21,22,23 },{ 31,32,33 }  };
int *pa[3]={ a,a[1],a[2] };
int *p=a[0];
```

порождают в программе объекты, представленные на схеме на рисунке 4.4

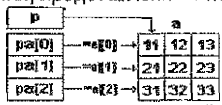

Рисунок 4.4 – Схема размещения указателей на двумерный массив

Согласно этой схеме доступ к элементу a[0][0] можно получить по указателям a, p, pa при помощи следующих ссылок: a[0][0], *a, **a[0], *p, **pa, *p[0].

Динамическая память. Распределение памяти

Часто возникают ситуации, когда заранее неизвестно, сколько объектов – чисел, строк текста и прочих данных – будет хранить программа. В этом случае используется динамическое выделение памяти, когда память занимается и освобождается в процессе исполнения программы. При динамическом выделении памяти для хранения данных используется специальная область памяти, так называемая «куча» (heap). Объем «кучи» и ее местоположение зависят от модели памяти, которая определяет логическую структуру памяти программы. Функции, выполняющие динамическое распределение памяти в «куче», и заголовочные файлы, в которых эти функции объявлены, пред-

ставлены в таблице 4.1.При каждом обращении к функции распределения памяти выделяется запрошенное число байт. Адрес начала выделенной памяти возвращается в точку вызова функции и записывается в переменную-указатель. Созданная таким образом переменная называется динамической переменной.

Таблица 4.1

Функция управления памятью	Действие	Заголовочный файл в BC++ 3.1	Заголовочный файл в VC++ 6.0
malloc()	Распределение	stdlib.h alloc.h	stdlib.h malloc.h
calloc()	Распределение		
realloc()	Перераспределение		
free()	Освобождение		

Если выделенный участок памяти больше не требуется, он может быть освобожден. При высокой активности по динамическому распределению памяти «куча» фрагментируется. Для смягчения отрицательных последствий фрагментации служат функции повторного распределения памяти. Они пытаются либо расширить, либо уменьшить размер ранее выделенного блока памяти.

Динамическая память. Функции управления памятью

Рассмотрим функции управления памятью:

- malloc() – предназначена для выделения непрерывной области памяти заданного размера, например, void * malloc(size_t size),где size_t – тип результата операции sizeof, определяемый в различных объектах-заголовках (stdio.h,stdlib.h,string.h и др.) и соответствующий типу unsigned long; size – размер выделяемой памяти в байтах. Функция malloc возвращает указатель без типа. Если выделить память не удалось, функция возвращает значение NULL.
- calloc() – предназначена для выделения памяти под заданное количество объектов, каждый из которых имеет размер size байт, всего n × size байт. Возвращает указатель на первый байт выделенной область памяти. Если выделить память не удалось, функция возвращает значение NULL: void * calloc(size_t n, size_t size);
- realloc() – предназначена для изменения размера динамически выделенной области, адресуемой указателем ptr, при этом размер области может как увеличиваться, так и уменьшаться:void* realloc(void* ptr, size_t size);где ptr – указатель на первоначальную область памяти, size – новый размер области памяти. Функция возвращает адрес новой области памяти, при этом старая область освобождается. Данные из освобождаемой области копируются в новую, но не более size байт. В некоторых случаях область, адресуемая первоначально указателем ptr, может сохраниться, только изменится ее размер. Если выделить память не удалось, функция возвращает значение NULL. Для того чтобы избежать ошибок в связи с перемещением данных в динамической памяти, функцию realloc() следует использовать так:

  ptr2=realloc(ptr1,new_size);

  if(ptr2!=NULL) /*Проверка выделения новой памяти*/ ptr1=ptr2; /*Запись адреса новой области памяти в исходный указатель*/

Таким образом, указатель ptr1 всегда будет хранить текущий адрес размещения данных, независимо от того, были данные перемещены функцией realloc() или нет.

- free() – предназначена для освобождения памяти, выделенной функциями malloc(), calloc(), realloc(). После выполнения функции освобожденная память может быть выделена вновь под другие данные:void free(void* ptr); где prt – указатель на область памяти, созданной только функциями динамического управления памятью malloc(), calloc(), realloc(). Использование других указателей в функции free делает ее поведение неопределенным и может привести к потере управления памятью.

Динамические массивы

Рассмотрим пример создания динамического числового массива. Функция calloc() выделяет память для 10 элементов типа int. Указатель ptr хранит адрес первого элемента массива. Значения элементов вводятся с клавиатуры. Затем вычисляется сумма всех элементов. После вывода результата на экран память, распределенная под массив, освобождается:

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{       int* ptr=(int*)calloc(10,sizeof(int));
        int s=0;
        for(int i=0;i<10;i++) scanf("%d", &ptr[i]);
        for(i=0;i<10;i++)  s+=ptr[i];
        printf("Summa=%d\n",s);
        free(ptr);
        return 0;}
```

После освобождения памяти значение адреса, присвоенное указателю ptr, не следует использовать, так как эта область может быть выделена под другую переменную. Рекомендуется в такой указатель записать значение пустого указателя NULL : ptr=NULL;

Для создания двумерного массива вначале нужно распределить память для массива указателей на одномерные массивы, а затем распределять память для одномерных массивов. Пусть, например, требуется создать массив a[n][m], это можно сделать при помощи следующего фрагмента программы:

```
        main ()
        { double **a;
        int n,m,i;
        scanf("%d %d",&n,&m);
        a=(double **)calloc(m,sizeof(double *));
        for (i=0; i<=m; i++)
        a[i]=(double *)calloc(n,sizeof(double));}
```

Указатель на массив не обязательно должен показывать на начальный элемент некоторого массива. Он может быть сдвинут так, что начальный элемент будет иметь индекс отличный от нуля, причем он может быть как положительным, так и отрицательным.

Лабораторная работа №5

Тема: «Строки»

В языке С отсутствует специальный строковый тип данных. Строка представляет собой последовательность (одномерный массив) из одного или более символов, последним из которых является нулевой символ таблицы ASCII '\0'. Это единственный вид строки, определенный в С.

Язык С поддерживает строковые константы, называемые строковыми литералами. Строковый литерал – это любая последовательность символов, заключенная в двойные кавычки ("..."). В конце литерала компилятор автоматически добавляет нулевой символ.

Не следует путать понятия строки и символа. Символьная константа заключается в одинарные кавычки, а строковая – в двойные. Например, 'c' – символьная константа, а "c" – строковая константа.

Литерал можно задать с помощью директивы препроцессора define: # define STR "..."

Строковая переменная может быть сформирована как одномерный массив типа char либо с помощью указателя на переменную типа char. Количество символов в массиве (объем выделяемой памяти) должно быть не меньше, чем количество символов в строке плюс один символ для хранения символа нуль.

Представление символьной строки при помощи одномерного массива

Синтаксис объявления имеет вид: char ID [N];

где ID – идентификатор массива, N – длина массива, при этом в памяти для хранения строки выделяется N байт.

Идентификатор массива – константа типа указатель, значение которой равно адресу первого элемента массива.

Инициализация возможна двумя способами:

- посимвольная инициализация char st[10]={'y','e','s','\0'};
  при этом оставшиеся 6 позиций не будут заполнены;
- инициализация на основе строковой константы char st [10]="Yes";

при этом в выделенную для строки память будут помещены 3 символа и добавлен четвертый – символ '\0'.

Инициализация и объявление возможны без указания длины массива char st[]={'y','e','s','\0'};

в этом случае будет создан массив из четырех элементов.

Указатель на символьную строку

По форме записи данная конструкция ничем не отличается от указателя на символьную переменную: char *S1;

где char – тип указателя на символ, S1 – переменная-указатель.

Для инициализации указателя требуется указать область памяти, где уже находится или будет находиться строка, при этом для строки должен быть выделен необходимый объем памяти.

Существует ряд способов инициализации указателя на строку:

- инициализация строковым литералом char *S1="Yes";
- присваивание значение другого указателя char *S1=S;

где S – идентификатор массива или указатель на другую строку символов.

Указателю можно присваивать значение другого указателя: S1=S;
где S1 – переменная типа указатель; S – строковая константа, идентификатор массива или указатель на другую строку символов. Например, char *S1, S[10]="Yes"; S1=S;
При этом создаются переменная-указатель S1 и массив символов S, под который выделяется поле длиной 10 символов, 4 из которых будут заполнены. Затем в переменную-указатель записывается адрес символьного массива (рисунок 5.1).

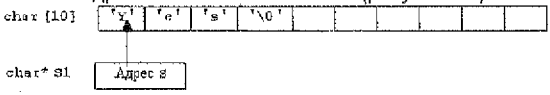

Рисунок 5.1 – Строковая константа и указатель

Ввод/вывод символьных строк

Ввод символьных строк с клавиатуры в языке С осуществляется с помощью функций scanf() и gets(), объявленных в заголовочном файле stdio.h. При работе с этими функциями следует помнить, что для строк предварительно должна быть выделена память. Конечный нуль добавляется после завершения ввода строки автоматически.
Существует несколько способов ввода строк:

- scanf ("%s",S1); где S1 – указатель типа char* на предварительно выделенную область. При использовании формата %s аргумент рассматривается как строка. Ввод строки реализуется до пробела или нажатия [Enter]. Несколько слов вводить в одну переменную с помощью scanf нельзя;
- scanf ("%Ns",S1); где N – максимальное число символов, записываемых в строку S1. Заполнение строки заканчивается при вводе N непробельных символов или выполняется до первого пробельного символа.
- gets (S1); при помощи этой функции можно вводить строку, содержащую пробелы. Ввод прекращается при нажатии [Enter] или при заполнении буфера клавиатуры.

Вывод символьных строк на экран в С осуществляется с помощью функций printf() и puts():

- printf("%s",S1); где S1 – указатель типа char*
- puts (S1);

Вывод строки продолжается до символа '/0'.
Пример

```
#include <stdio.h>
#define STR "What is your name?"
int main()
{char *ch;
  static char ch1[]="My name is George\n";
  ch=STR;
  printf("\n%s \t %s",ch,ch1);
  /*\t – табуляция – сдвиг на заданное количество позиций*/
  for (int i=0;i<6;i++)
  {printf("\n %c", *(ch+i));
  printf("   %c", *(ch1+i));}
  return 0;}
```

Массивы символьных строк

Массив строк есть не что иное, как массив данных определенного типа. Строка сама уже является массивом символов, поэтому массив символьных строк является двумерным.

При определении массива строк используются два индекса:

- первый индекс определяет максимальное количество строк в массиве;
- второй – максимальную длину каждой строки.

Например, static char massiv [3][8]= {"Ivanov", "Petrov", "Sidorov"};

Под переменную massiv выделяется область размером 3×8 байт памяти, по 8 байт для каждой строки. Таким образом, указатель на начало массива massiv и указатели на начало строк massiv[i] представляют собой указатели-константы.

Можно объявить массив символьных строк с помощью указателей: char *list[N];

где N – количество строк в массиве.

Переменная list представляет собой массив из N указателей, память при этом выделяется для хранения N значений указателей. Таким образом, имя массива list является константой-указателем. Элементы массива list[i] используются для хранения адресов строк и являются переменными величинами.

Инициализация символьных строк с помощью указателей:

```
char *list[3]={"Ivanov","Petrov","Sidorov"};
```

Доступ к строке:

```
list [0] – ссылка на "Ivanov";
```

Доступ к символу указанной строки:

```
*list[0] – определяет символ "I" в строке "Ivanov";
*(list[1]+2) – символ 't' в строке "Petrov".
```

Функции работы со строками

Язык С определяет строки как особый вид массивов, позволяя осуществлять их ввод/вывод как единого целого. Встроенные средства обработки строк в языке С отсутствуют, но строки настолько широко используются в программировании, что большинство компиляторов имеет специальные функции для работы со строками. Эти функции входят в состав заголовочного файла <string.h>.

Отметим ряд основных функций.

Определение длины строки:

strlen (S1) – длина строки S1, исключая нулевой символ.

Копирование строк:

strcpy (S1,S2), где S1 – указатель, S2 – указатель или константа.

Строка S2 копируется посимвольно в строку S1. Необходимо иметь в виду, что размер строк компилятором не сравнивается, и это возлагается на программиста.

Слияние строк (конкатенация):

strcat (S1,S2).

К концу строки S1 подсоединяется строка S2. Нулевой символ в конце строки S1 отбрасывается. Компилятор не следит, хватит ли в S1 места для S1 плюс S2.

Сравнение строк:

strcmp (S1,S2).

Сравниваются коды символов, находящихся на одинаковых позициях в строках S1 и S2, начиная с нулевой. В зависимости от компилятора результат работы функции следующий:

- возвращается нуль, если строки одинаковы, и значение отличное от нуля, если строки не совпадают;
- отрицательное число, если строка S1 «меньше» строки S2 с точки зрения алфавита (ASCII-кодов), и положительное, если «больше».

Данная функция используется для упорядочения по какому-либо критерию.

Лабораторная работа №6

Тема: «Структуры»

Структурой, называется совокупность логически связанных переменных различных типов, сгруппированных под одним именем для удобства дальнейшей обработки. Структура – это способ связать воедино данные разных типов и создать пользовательский тип данных.

Определение структуры

Структура – тип данных, задаваемый пользователем.

Определение типа структуры представляется в виде

```
struct ID
    {
      <тип> <имя 1-го элемента>;
      <тип> <имя 2-го элемента>;
      .....
      <тип> <имя последнего элемента>;
    };
```

Определение типа структуры начинается с ключевого слова struct и содержит список объявлений, заключенных в фигурные скобки. За словом struct следует имя типа, называемое телом структуры. Элементы списка объявлений называются членами структуры или полями. Каждый элемент списка имеет уникальное для данного структурного типа имя. Однако следует заметить, что одни и те же имена полей могут быть использованы в различных структурных типах.

Определение типа структуры представляет собой шаблон (template), предназначенный для создания структурных переменных.

Объявление переменной структурного типа имеет следующий вид:

```
struct ID var1;
```

при этом в программе создается переменная с именем var1 типа ID. Все переменные, использующие один шаблон (тип) структуры, имеют одинаковый набор полей, однако различные наборы значений, присвоенные этим полям. При объявлении переменной происходит выделение памяти для размещения переменной. Шаблон структуры позволяет определить

размер выделяемой памяти. В общем случае, под структурную переменную выделяется область памяти не менее суммы длин всех полей структуры, например,

```
struct list
{  char name[20];
   char first_name[40];
   int;
} L;
```

Определение типа структуры может быть задано в программе на внешнем уровне, при этом имя пользовательского типа имеет глобальную видимость (при этом память не выделяется). Определение типа структуры также может быть сделано внутри функции, тогда имя типа структуры имеет локальную видимость. Создание структурной переменной возможно двумя способами: с использованием шаблона (типа) или без него.

Создание структурной переменной pt на основе шаблона выполняется следующим образом:

```
struct point       //Определение типа структуры
{int x;int y;};
.....
struct point pt;       //Создание структурной переменной
```

Структурная переменная может быть задана уникальным образом:

```
struct       //Определение анонимного типа структуры
{char name[20];
  char f_name[40];
  char s_name[20];
  } copymy;       //Создание структурной переменной
```

При размещении в памяти структурной переменной можно выполнить ее инициализацию. Неявная инициализация производится для глобальных переменных, переменных класса static. Структурную переменную можно инициализировать явно при объявлении, формируя список инициализации в виде константных выражений.

Формат: struct ID name_1={значение1, ... значениеN};

Внутри фигурных скобок указываются значения полей структуры, например,

```
struct point pt={105,17};
```

при этом первое значение записывается в первое поле, второе значение – во второе поле и т. д., а сами значения должны иметь тип, совместимый с типом поля.

Над структурами возможны следующие операции:

- присваивание значений одной структурной переменной другой структурной переменной, при этом обе переменные должны иметь один и тот же тип;
- получение адреса переменной с помощью операции &;
- осуществление доступа к членам структуры.

Присваивание значения одной переменной другой выполняется путем копирования значений соответствующих полей, например:

```
struct point pt={105,15},pt1;
pt1=pt;
```

В результате выполнения этого присваивания в pt1.x будет записано значение 105, а в pt1.y – число 15.

Работа со структурной переменной обычно сводится к работе с отдельными полями структуры. Доступ к полю структуры осуществляется с помощью операции. (точка) посредством конструкции вида: имя_структуры.имя_поля_структуры;
при этом обращение к полю структуры представляет собой переменную того же типа, что и поле, и может применяться везде, где допустимо использование переменных такого типа. Например: struct list copy = {"Ivanov","Petr",1980};
Обращение к "Ivanov" имеет вид copy.name. И это будет переменная типа указатель на char. Вывод на экран структуры copy будет иметь вид:
printf("%s%s%d\n",copy.name,copy.first_name,copy.i);
Структуры нельзя сравнивать. Сравнивать можно только значения конкретных полей.

Структуры и функции

Структуры могут быть переданы в функцию в качестве аргументов и могут служить в качестве возвращаемого функцией результата.
Существует три способа передачи структур функциям:

- передача компонентов структуры по частям;
- передача целиком структуры;
- передача указателя на структуру.

Например, в функцию передаются координаты двух точек:

```
void showrect(struct point p1,struct point p2)
{printf("Левый верхний угол прямоугольника:%d %d\n",p1.x, p1.y);
  printf("Правый нижний угол прямоугольника:%d %d\n", p2.x, p2.y);
}
```

При вызове такой функции ей надо передать две структуры:

```
struct point pt1={5,5}, pt2={50,50};
showrect(pt1,pt2);
```

Теперь рассмотрим функцию, возвращающую структуру:

```
struct point makepoint (int x,int y) /*makepoint – формирует точку по компонентам x и
y*/
{struct point temp;
  temp.x = x;
  temp.y = y;
  return temp;}
```

Результат работы этой функции может быть сохранен в специальной переменной и выведен на экран:

```
struct point buf;
buf=makepoint(10,40);
printf("%d %d\n",buf.x,buf.y);
```

После выполнения этого фрагмента на экран будут выведены два числа: 10 и 40.

Указатели на структуру

Если функции передается большая структура, то эффективнее передать указатель на эту структуру, нежели копировать ее в стек целиком. Указатель на структуру по виду ничем не отличается от указателей на обычные переменные.

Формат: struct point *pp;
где pp – указатель на структуру типа struct point, *pp – сама структура, (*pp).x и (*pp).y – члены структуры.
Скобки (*pp).x необходимы, так как приоритет операции (.) выше приоритета операции (*). В случае отсутствия скобок *pp.x понимается как *(pp.x).

Инициализация указателя на структуру выполняется так же, как и инициализация указателей других типов: struct point var1, *S1; здесь var1 – структурная переменная, *S1 – указатель на структуру.
Для определения значения указателя ему нужно присвоить адрес уже сформированной структуры:

```
S1 = &var1;
```

Теперь возможно еще одно обращение к элементам структуры:

```
(*S1).name.
```

Указатели на структуры используются весьма часто, поэтому для доступа к ее полям была введена короткая форма записи. Если p – указатель на структуру, то p → <поле структуры>позволяет обратиться к указанному полю структурной переменной.
Знак → (стрелка) вводится с клавиатуры с помощью двух символов: '–' (минус) и '>' (больше). Например, pp → x; pp → y.
Операторы доступа к полям структуры (.) и (→) вместе с операторами вызова функции () и индексами массива [] занимают самое высокое положение в иерархии приоритетов операций в языке С.
Указатели на структуру используются в следующих случаях:

- доступ к структурам, размещенным в динамической памяти;
- создание сложных структур данных – списков, деревьев;
- передача структур в качестве параметров в функции.

Массивы структур

При необходимости хранения некоторых данных в виде нескольких массивов одной размерности, но разного типа, возможна организация массива структур. Наглядно массив структур можно представить в виде таблицы, где столбцы таблицы представляют собой поля структуры, а строки – элементы массива структур.
Например, имеется два разнотипных массива:

```
char c[N];
int i[N];
```

Объявление

```
struct key
{char c;
      int i;
} keytab[N];
```

создает тип структуры key и объявляет массив keytab[N], каждый элемент которого есть структура типа key.
Возможна запись:

```
struct key
{     char c;
      int i;
};
struct key keytab[N];
```

Инициализация массива структур выполняется следующим образом:

```
struct key
{       char c;
        int i;
      } keytab[]={'a', 0,'b', 0};
```

В этом примере создается массив на две структуры типа key с именем keytab. Рассмотрим обращение к полю структуры в этом случае – для примера выведем на экран содержимое массива:

```
for(int i=0;i<2;i++)
  printf("%c %d\n",keytab[i].c,keytab[i].i);
```

При обращении к полю структуры сначала происходит обращение к элементу массива (keytab[i]), а затем только обращение к полю структуры (keytab[i].c).

Доступ к полю элемента массива структур может быть получен через константу-указатель на массив и смещение внутри массива, например, доступ к полю элемента массива с номером i следует записать следующим образом:

(*(keytab+i)).c или (keytab+i)→ c.

Массив структур также может быть передан в функцию в качестве аргумента, передача массива происходит через указатель на массив.

Вложенные структуры

Полем структурной переменной может быть переменная любого типа, в том числе другая структурная переменная. Поле, представляющее собой структуру, называется вложенной структурой.

Тип вложенной структуры должен быть объявлен раньше. Кроме того, структура не может быть вложена в структуру того же типа.

Объявление вложенной структуры:

```
struct point
{int x,y;};
struct rect
{struct point LUPoint, RDPoint;
        char BorderColor[20];
};
struct rect Rect;
```

В переменной Rect два поля LUPoint (точка, соответствующая левому верхнему углу прямоугольника) и RDPoint (точка, соответствующая правому нижнему углу) представляют собой вложенные структуры. Для доступа к полю вложенной структуры следует сначала обратится к внешней структуре, затем к вложенной: Rect.LUPoint.x.

Пример: создание и использование вложенной структуры:

```
struct rect Rect={10,5,50,25,"White"};
printf("Параметры прямоугольника:\n");
printf("Координаты левого верхнего угла %d %d\n", Rect.LUPoint.x, Rect.LUPoint.y);
printf("Координаты левого верхнего угла %d %d\n", Rect.RDPoint.x, Rect.RDPoint.y);
printf("Цвет границы: %s\n", Rect.BorderColor);
```

В качестве поля структуры также можно использовать указатели на любые структуры, в том числе на структуры того же типа:

```
struct PointList
{       int x,y;
        struct PointList* LastPoint;
};
```

Структуры, имеющие в своем составе поля-указатели на такую же структуру, используются для создания сложных структур данных – списков, деревьев.

Использование синонима типа

Ключевое слово typedef позволяет в программе создать синоним типа, который может использоваться для объявления переменных, параметров функций. Синоним можно создать для любого существующего типа (int, float и т. д.), в том числе для пользовательского типа – структуры или массива.

Пример, создание синонима структуры:

```
typedef struct point
{int x,y;
} POINT;
```

Идентификатор POINT представляет собой синоним типа point. С помощью синонима POINT можно объявить переменную:

```
POINT pt1;
```

или передать переменную в функцию:

```
void ShowRect(POINT pt1,POINT pt2);
```

Объединения

Объединение – это тип данных, позволяющий переменным различного типа использовать одну и ту же область памяти. Для объявления объединения используется ключевое слово union. Объединение, так же как и структура, содержит несколько полей. Однако в любой момент времени доступно только одно поле. Под объединение выделяется столько памяти, сколько требуется для размещения самого большого поля. Все поля поочередно используют выделенную память для хранения своих значений.

Определение типа:

```
union tag
{       тип1 переменная1;
        тип2 переменная2;
        ......};
```

где tag – имя типа.

Объявление переменной:

```
union tag u1;
```

или

```
union tag
{       тип1 переменная1;
        тип2 переменная2;
    ......
} u1;
```

Инициализация объединения может быть выполнена при объявлении, при этом тип инициализирующего значения должен соответствовать первому полю объединения:

```
union tag
{  int i;
   double d;
} u={10};
```

Доступ к полю объединения осуществляется с помощью операций – . (точка) и → (стрелка).

```
printf("%d",u.i);
```

Результат: 10.

```
union tag *p=&u;
printf("%d", p→i);
```

Результат: 10.

При обращении к полям объединения следует помнить, какое поле размещалось в памяти последним. Если пытаться извлекать данные с помощью поля другого типа, значение может быть прочитано неправильно.

Пример

```
union tag
{       int i;
        double d;
} u={1.2};
printf("%d\n",u.i); //Вывод на экран числа 1
printf("%lf\n",u.d); //Вывод на экран числа 0
```

Ответ «1» получен потому, что инициализация должна соответствовать типу первого поля – int; ответ «0» – следствие того, что данные типа int прочитаны полем типа float неправильно.

Лабораторная работа №7

Тема: «Файлы»

Работа с файлами

Файл – это информация, размещенная на каком-либо носителе (диске) или в буфере ввода/вывода устройства (клавиатура). Для обмена данными файл должен быть открыт, по завершении этого процесса – закрыт. Поток– это логический канал, предназначенный для выполнения операций ввода/вывода. Каждому файлу при его открытии ставится в соответствие поток.

В языке С существуют стандартные потоки:

stdin – стандартный консольный ввод (клавиатура по умолчанию);

stdout – стандартный консольный вывод (монитор по умолчанию);

Стандартные потоки открываются при каждом запуске программы.

Для работы с файлами в программах на С используется заголовочный файл stdio.h, в котором объявлен специальный тип данных – структура FILE, предназначенная для хранения атрибутов (параметров) файлов (указатель текущей позиции файла, признак

конца файла, флаги индикации ошибок, сведения о буферизации и др.). Поля структуры типа FILE доступны с помощью специальных С-функций. Для организации работы с файлом используется определенная последовательность действий.

Объявление потока – переменной-указателя на структуру типа FILE, в которой будут храниться атрибуты файла

```
FILE *fl,
```

где *fl – указатель на файл.

Открытие файла

```
fl=fopen("путь к файлу","режим работы файла");
```

Параметр "путь к файлу" указывает размещение файла на диске и обязательно содержит имя файла и может содержать имя логического диска и путь.

Параметр "режим работы файла" показывает, как будет использоваться файл:

"w" – для записи данных (вывод);

"r" – для чтения данных (ввод);

"a" – для добавления данных к существующим записям.

Примеры открытия файлов:

```
FILE *f_in,*_out;
f_in=fopen("My_file1","r");
f_out=fopen("My_file2","w");
```

Функция fopen() возвращает значение указателя на структуру типа файл. Если файл по каким-либо причинам не открывается, функция fopen() возвращает значение NULL.

Рассмотрим особенности режимов открытия файлов. Если файл открывается в режиме записи данных "w", то указатель текущей позиции устанавливается на начало файла. Если указанный в функции fopen() файл не существует, то он создается. Необходимо помнить, что открытие существующего файла в режиме "w" приводит к уничтожению его старого содержания.

Открытие файла для чтения в режиме "r" возможно только для созданного ранее файла, при этом указатель текущей позиции устанавливается на начало файла. Если открываемый на чтение файл не существует, функция fopen() возвращает пустой указатель со значением NULL.

Если файл открывается в режиме добавления данных "a", то указатель текущей позиции устанавливается на конец файла. Данные, ранее помещенные в файл, остаются без изменений. Если указывается несуществующий файл, то он создается заново.

Также можно указать дополнительные условия режима открытия файла:

"b" – двоичный поток;

"t" – текстовый поток;

"+" – обновление файла.

Пример

"r+" – чтение файла с обновлением, т. е. возможна перезапись данных с усечением;

"w+" – запись в файл и одновременно чтение;

"a+" – добавление данных и чтение.

Для поочередного выполнения чтения и записи в режиме "+" необходимо ручное позиционирование курсора.

Обработка открытого файла

Специальные функции:

Чтение (ввод)	Запись (вывод)
getc()	putc()
fscanf()	fprintf()
fgets()	fputs()
fread()	fwrite()

При каждой операции ввода/вывода указатель текущей позиции файла смещается на одну позицию в сторону конца файла.

Проверка признака конца файла

Так как при каждой операции ввода/вывода происходит перемещение указателя текущей позиции в файле, в какой-то момент указатель достигает конца файла. Структура типа FILE имеет поле – индикатор конца файла. Функция feof() проверяет состояние индикатора конца файла и возвращает значение 0, если конец файла не был достигнут, или значение, отличное от нуля, если был достигнут конец файла. Функция имеет единственный аргумент – указатель на поток типа FILE. Вызов функции:

```
if (! feof(f_in))...
```

проверяет, что конец файла еще не достигнут.

Закрытие файла

После завершения обработки файла его следует закрыть с помощью функции fclose(). При этом разрывается связь указателя на файл с внешним набором данных. Освободившийся указатель можно использовать для другого файла. Формат вызова функции:

```
fclose(f_in);
```

При нормальном завершении программы в большинстве операционных систем все открытые файлы закрываются автоматически, но рекомендуется закрывать все файлы, дальнейшая обработка которых в программе не предполагается, при помощи функции fclose().

Функции ввода/вывода

Простейший способ выполнить чтение из файла или запись в файл – использовать функции getc() или putc().

Функция getc() выбирает из файла очередной символ; ей нужно только знать указатель на файл, например, char Symb=getc(f_in);

Если при обработке достигается конец файла, то функция getc() возвращает значение EOF(end of file).

Функция putc() заносит значение символа Symb в файл, на который указывает f_out. Формат вызова функции: putc(Symb,f_out);

Пример. Записать в файл буквы, вводимые с клавиатуры. Ввод продолжается до нажатия клавиши F6 или CTRL/z (ввод символа EOF – конца файла):

```
#include <stdio.h>
int main(void)
{char c;
  FILE *out;
  out=fopen("Liter","w");
  while ((c=getchar( ) )!=EOF)
     fputc(c,out);
  fclose(out);
  return 0;}
```

Функции fscanf() и fprintf() выполняют форматированный ввод/вывод. Чтение из файла выполняет функция fscanf():

fscanf(f_in,[строка формата],[список адресов переменных]);

Функция возвращает количество введенных значений или EOF.

Запись в файл осуществляет функция fprintf():

fprintf(f_out,[строка формата],[список переменных, констант]);

Возвращает количество выведенных байт (символов) или EOF.

Строка формата функций fscanf() и fprintf() формируется так же, как для консольного ввода/вывода и функциям printf() и scanf().

Следует заметить, что вызов функции

fscanf(stdin,[строка формата],[список адресов переменных]);

эквивалентен вызову

scanf([строка формата],[список адресов переменных]);

Аналогично,

fprintf(stdout, [строка формата],[список переменных, констант]);

эквивалентно

printf([строка формата],[список переменных, констант]);

Рассмотрим примеры программ, использующих эти функции.

Пример В программе создается массив, состоящий из четырех целых чисел. Вывести массив в файл:

```
#include <stdio.h>
#define n 4
int main()
{  int i=0;
   int array[n]={4,44,446,4466};
   FILE *out;
   out=fopen("num_arr.txt","w");
   for(;i<n;i++)
      fprintf(out,"%6.2d",array[i]);
   fclose(out);
   return 0;}
```

В результате выполнения программы в файл num_arr.txt будет помещена следующая информация:

			0	4			4	4			4	4	6		4	4	6	6

Лабораторная работа № 8

Тема: «Динамические структуры»

Организация списков и их обработка. Линейные списки

Линейный список - это конечная последовательность однотипных элементов (узлов), возможно, с повторениями. Количество элементов в последовательности называется длиной списка, причем длина в процессе работы программы может изменяться. Линейный список F, состоящий из элементов D1,D2,...,Dn, записывают в виде последовательности значений заключенной в угловые скобки F=, или представляют графически (см.рисунок 8.1).

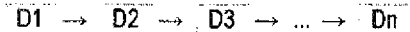

Рисунок 8.1 – Изображение линейного списка

Например, F1=< 2,3,1>,F2=< 7,7,7,2,1,12 >, F3=< >. Длина списков F1, F2, F3 равна соответственно 3,6,0.

При работе со списками на практике чаще всего приходится выполнять следующие операции: найти элемент с заданным свойством; определить первый элемент в линейном списке; вставить дополнительный элемент до или после указанного узла; исключить определенный элемент из списка; упорядочить узлы линейного списка в определенном порядке. Методы хранения линейных списков разделяются на методы последовательного и связанного хранения. Рассмотрим простейшие варианты этих методов для списка с целыми значениями F=<7,10>.

При последовательном хранении элементы линейного списка размещаются в массиве d фиксированных размеров, например, 100, и длина списка указывается в переменной l, т.е. в программе необходимо иметь объявления вида: float d[100]; int l;

Размер массива 100 ограничивает максимальные размеры линейного списка. Список F в массиве d формируется так: d[0]=7; d[1]=10; l=2;

Полученный список хранится в памяти согласно схеме на рисунок 8.2

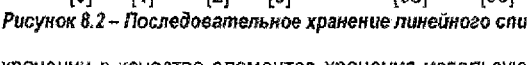

Рисунок 8.2 – Последовательное хранение линейного списка.

При связанном хранении в качестве элементов хранения используются структуры, связанные по одной из компонент в цепочку, на начало которой (первую структуру) указывает указатель dl. Структура, образующая элемент хранения, должна кроме соответствующего элемента списка содержать и указатель на соседний элемент хранения.

Описание структуры и указателя в этом случае может иметь вид:

```
typedef struct snd/* структура элемента хранения*/
{ float  val;          * элемент списка*/
struct snd *n ;      /* указатель на элемент хранения */
} DL;
DL *p;   /* указатель текущего элемента*/
DL *dl;  /* указатель на начало списка*/
```

Для выделения памяти под элементы хранения необходимо пользоваться функцией malloc(sizeof(DL)) или calloc(l,sizeof(DL)). Формирование списка в связанном хранении может осуществляется операторами:

```
p=malloc(sizeof(DL));
p->val=10;  p->n=NULL;
dl=malloc(sizeof(DL));
dl->val=7;  dl->n=p;
```

В последнем элементе хранения (конец списка) указатель на соседний элемент имеет значение NULL. Получаемый список изображен на рисунке 8.3

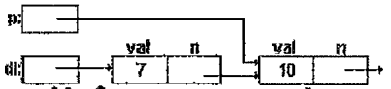

Рисунок 8.3 – Связное хранение линейного списка

Операции со списками при последовательном хранении

При выборе метода хранения линейного списка следует учитывать, какие операции будут выполняться и с какой частотой, время их выполнения и объем памяти, требуемый для хранения списка.

Пусть имеется линейный список с целыми значениями и для его хранения используется массив d (с числом элементов 100), а количество элементов в списке указывается переменной l. Реализация указанных ранее операций над списком представляется следующими фрагментами программ, которые используют объявления:

```
float d[100];
int i,j,l;
```

1) печать значения первого элемента (узла)

```
if (i<0 || i>l) printf("\n нет элемента");
else printf("d[%d]=%f ",i,d[i]);
```

2) удаление элемента, следующего за i-м узлом

```
if (i>=l) printf("\n Net soseda ");
l--;
for (j=i+1;j<="1" ||="" i=""">=l) printf("\n Net soseda");
else printf("\n %d  %d",d[i-1],d[i+1]);
```

4) добавление нового элемента new за i-м узлом

```
if (i==l || i>l) printf("\n Nelza dobavit");
else
{   for (j=l; j>i+1; j--) d[j+1]=d[j];
    d[i+1]=new; l++;}
```

5) частичное упорядочение списка с элементами K1,K2,...,Kl в список K1',K2',...,Ks,K1,Kt",...,Kt", s+t+1=l так, чтобы K1'=K1;

после упорядочения указатель v указывает на элемент K1' (см. рисунок 8.4)

Рисунок 8.4 – Схема частичного упорядочения списка

```
ND *v;
float k1;
k1=dl->val;
r=dl;
while( r->n!=NULL )
{ v=r->n;
  if (v->val<k1) {r->n=v->n;
     v->n=dl;
     dl=v;}
   else r=v; }
```

Количество действий, требуемых для выполнения указанных операций над списком в связанном хранении, оценивается соотношениями: для операций 1 и 2 - Q=l; для операций 3 и 4 - Q=1; для операции 5 - Q=l.

Организация двусвязных списков

Связанное хранение линейного списка называется списком с двумя связями или двусвязным списком, если каждый элемент хранения имеет два компонента указателя (ссылки на предыдущий и последующий элементы линейного списка). В программе двусвязный список можно реализовать с помощью описаний:

```
typedef struct ndd
  { float val;     /* значение элемента          */
    struct ndd * n; /* указатель на следующий элемент  */
    struct ndd * m; /* указатель на предыдущий элемент */
    } NDD;
NDD * dl, * p, * r;
```

Графическая интерпретация метода связанного хранения списка F=< 2,5,7,1 > как списка с двумя связями приведена на рисунке 8.5.

Рисунок 8.5 – Схема хранения двусвязного списка

Вставка нового узла со значением new за элементом, определяемым указателем p, осуществляется при помощи операторов:

```
r=malloc(NDD);
r->val=new;
r->n=p->n;
(p->n)->m=r;
p->=r;
```

Удаление элемента, следующего за узлом, на который указывает p

```
p->n=r;
p->n=(p->n)->n;
( (p->n)->n )->m=p;
free(r);
```

Связанное хранение линейного списка называется циклическим списком, если его последний указывает на первый элемент, а указатель dl - на последний элемент списка. Схема циклического хранение списка F=< 2,5,7,1 > приведена на рисунке 8.6

Рисунок 8.6 – Схема циклического хранения списка

При решении конкретных задач могут возникать разные виды связанного хранения.

Пусть на входе задана последовательность целых чисел B1,B2,...,Bn из интервала от 1 до 9999, и пусть Fi (1<i по возрастанию. Составить процедуру для формирования Fn в связанном хранении и возвращения указателя на его начало.

При решении задачи в каждый момент времени имеем упорядоченный список Fi и при вводе элемента Bi+1 вставляем его в нужное место списка Fi, получая упорядоченный

список Fi+1. Здесь возможны три варианта: в списке нет элементов; число вставляется в начало списка; число вставляется в конец списка. Чтобы унифицировать все возможные варианты, начальный список организуем как связанный список из двух элементов <0,1000>.

Рассмотрим программу решения поставленной задачи, в которой указатели dl, r, p, v имеют следующее значение: dl указывает начало списка; p, v – два соседних узла; r фиксирует узел, содержащий очередное введенное значение in.

```
#include
#include
typedef struct str1
 { float val;
   struct str1 *n; } ND;
main()
 { ND *arrange(void);
   ND *p;
   p=arrange();
   while(p!=NULL)
    {
     printf("\n %f ",p->val);
     p=p->n;
    }
 }
ND *arrange() /*  формирование упорядоченного списка */
 { ND *dl, *r, *p, *v;
   float in=1;
   char *is;
   dl=malloc(sizeof(ND));
   dl->val=0;              /* первый элемент   */
   dl->n=r=malloc(sizeof(ND));
   r->val=10000; r->n=NULL;   /* последний элемент */
   while(1)
   {
     scanf(" %s",is);
     if(* is=='q') break;
     in=atof(is);
     r=malloc(sizeof(ND));
     r->val=in;
     p=dl;
     v=p->n;
     while(v->vain;
     }
     r->n=v;
     p->n=r;}
   return(dl);}
```

Стеки и очереди

В зависимости от метода доступа к элементам линейного списка различают разновидности линейных списков, называемые стеком, очередью и двусторонней очередью.

Стек - это конечная последовательность некоторых однотипных элементов -скалярных переменных, массивов, структур или объединений, среди которых могут быть и одинаковые. Стек обозначается в виде: S= и представляет динамическую структуру данных; ее количество элементов заранее не указывается и в процессе работы, как правило, изменяется. Если в стеке элементов нет, то он называется пустым и обозначается S=< >.

Допустимыми операциями над стеком являются:

- проверка стека на пустоту S=< >,
- добавление нового элемента Sn+1 в конец стека – преобразование < S1,...,Sn> в < S1,...,Sn+1>;
- изъятие последнего элемента из стека - преобразование < S1,...,Sn-1,Sn> в < S1,...,Sn-1>;
- доступ к его последнему элементу Sn, если стек не пуст.

Таким образом, операции добавления и удаления элемента, а также доступа к элементу выполняются только в конце списка. Стек можно представить как стопку книг на столе, где добавление или взятие новой книги возможно только сверху.

Очередь - это линейный список, где элементы удаляются из начала списка, а добавляются в конце списка (как обыкновенная очередь в магазине).

Двусторонняя очередь - это линейный список, у которого операции добавления и удаления элементов и доступа к элементам возможны как в начале, так и в конце списка. Такую очередь можно представить как последовательность книг, стоящих на полке, так что доступ к ним возможен с обоих концов.

Одной из форм представления выражений является польская инверсная запись, задающая выражение так, что операции в нем записываются в порядке выполнения, а операнды находятся непосредственно перед операцией.

Например, выражение (6+8)*5-6/2 в польской инверсной записи имеет вид

6 8 + 5 * 6 2 / -

Особенность такой записи состоит в том, что значение выражения можно вычислить за один просмотр записи слева направо, используя стек, который до этого должен быть пуст. Каждое новое число заносится в стек, а операции выполняются над верхними элементами стека, заменяя эти элементы результатом операции. Для приведенного выражения динамика изменения стека будет иметь вид

S = < >; <6>; <6,8>; <14>; <14,5>; <70>;
<70,6>; <70,6,2>; <70,3>; <67>.

Ниже приведена функция eval, которая вычисляет значение выражения, заданного в массиве m в форме польской инверсной записи, причем m[i]>0 означает неотрицательное число, а значения m[i]<0 - операции. В качестве кодировки операций сложения, вычитания, умножения и деления выбраны отрицательные числа -1, -2, -3, -4. Для органи-

зации последовательного хранения стека используется внутренний массив stack. Параметрами функции являются входной массив a и его длина l.

```
float eval (float *m, int l)
{ int p,n,i;
  float stack[50],c;
  for(i=0; i < l ;i++)
  if ((n=m[i])<0)
    {  c=st[p--];
       switch(n)
       { case -1:  stack[p]+=c;  break;
         case -2:  stack[p]-=c;  break;
         case -3:  stack[p]*=c;  break;
         case -4:  stack[p]/=c;}}
  else stack[++p]=n;
  return(stack[p]);}
```

ЛИТЕРАТУРА

1. Белецкий, Я. Энциклопедия языка Си. – М.: Мир, 1992.
2. Берри, Р. Язык С: введение для программистов / Р. Берри, Б. Микинз. – М.: Финансы и статистика, 1988.
3. Бочков, С.О. Язык программирования Си для персонального компьютера / С.О. Бочков, Д.М. Субботин. – М: Радио и связь, 1990.
4. Власов, С.Е. Разработка программ в интегрированной среде Турбо Си. – Н.Новгород: НКЦП, 1992.
5. Джамса, К. 1001 совет по C/C++. Настольная книга программиста. – М.: Март, БИНОМ, Универсал, 1997.
6. Керниган, Б. Язык программирования Си / Б. Керниган, Д. Ритчи; пер. с англ. – 3-е изд., испр. – СПб.: «Невский диалект», 2001.
7. Подбельский, В.В. Практикум программирования на языке Си: учебное пособие / В.В. Подбельский, С.В. Фомин. – 2-е изд. – М.: Финансы и статистика, 2002.
8. Прата, С. Язык программирования С. Лекции и упражнения. – Киев: ДиаСофт, 2000.
9. Трой, Д. Программирование на языке Си для персонального компьютера IBM PC. – М.: Радио и связь, 1991.
10. Уэйт, М. Язык Си. Руководство для начинающих / М. Уэйт, С. Прата, Д. Мартин. – М.: Мир, 1988.

УЧЕБНОЕ ИЗДАНИЕ

Составитель: Хацкевич Мария Викторовна

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ СИ

Методические указания к выполнению
лабораторных работ по дисциплине

«Основы алгоритмизации и программирования в традиционных и интеллектуальных компьютерах»

для студентов специальности
1-40 03 01 «Искусственный интеллект»

Ответственный за выпуск: Хацкевич М.В.
Редактор: Боровикова Е.А.
Компьютерная верстка: Боровикова Е.А.
Корректор: Никитчик Е.В.

Подписано к печати 21.03.2013 г. Формат 60x84 1/16. Бумага «Снегурочка».
Усл. печ. л. 4,0. Уч. изд. л. 4,25. Тираж 50 экз. Заказ № 253.
Отпечатано на ризографе учреждения образования
«Брестский государственный технический университет».
224017, г.Брест, ул.Московская, 267.